МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ
ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»

Гордієнко Ю.Г., Кочура Ю.П.

НЕЙРОННІ МЕРЕЖІ

Конспект лекцій

Навчальний посібник
для здобувачів магістра ступеня
за освітньою програмою «Інженерія програмного забезпечення комп'ютерних систем»
спеціальності 121 «Інженерія програмного забезпечення» за
освітньою програмою «Комп'ютерні системи та мережі»
спеціальності 123 «Комп'ютерна інженерія»
за освітньою програмою «Інформаційні керуючі системи та технології»
спеціальності 126 «Інформаційні системи та технології»

Електронне мережеве навчальне видання

ЗАТВЕРДЖЕНО
на засіданні кафедри навчальної техніки
протокол № 10 від 25.05.2022

2022 рік

Нейронні мережі

Лекція_01

****

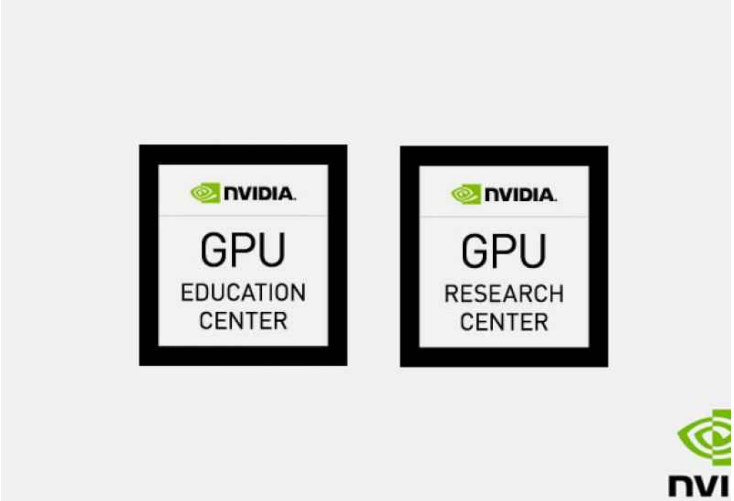
* Слайди лекції:
https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

****

Лекція 01 - Вступ

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) у рамках заг

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

НЕЙРОМЕРЕЖІ

ЛЕКЦІЯ 1: ВСТУП

Юрій Гордієнко, сертифікований інструктор DLI

NVIDIA | DEEP LEARNING INSTITUTE

Які поточні виклики в IT?

Давайте подивимося на сучасні тренди:

- **Великі дані**
- **Інтернет речей**
- **Розподілені обчислення**
- **Хмари**

+

- **Машинне навчання**
- **Глибоке навчання**
- **Штучний інтелект**
- **...**

Поточні виклики –Великі дані

Великі дані - проблема

Перша проблема великих даних виникла в 1880-х роках.

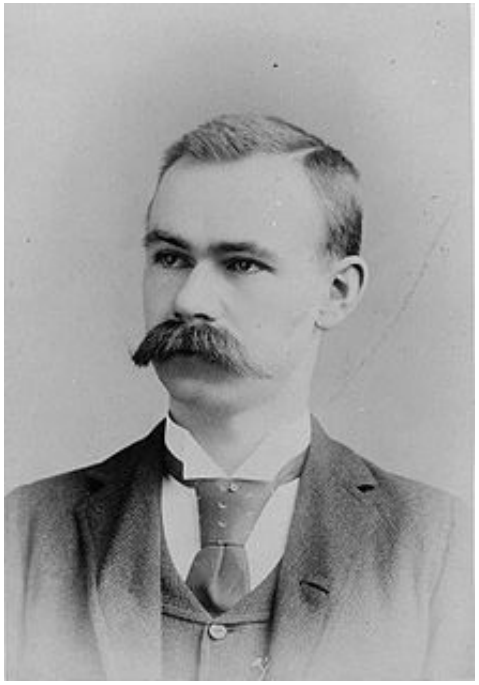
Наприкінці 1800-х років обробка перепису населення США була починає займати близько 10 років. Проходить перепис **кожні 10 років**і населення, а отже, кількість інформації збільшувалася —**проблема**!

У 1886 році Герман Голлеріт відкрив бізнес з оренди машин, які могли зчитувати та зводити в таблиці дані перепису населення.

перфокарти. Перепис 1890 року тривав менше 2 років, він охопив більшу кількість населення (62 мільйони людей) і більше даних, ніж перепис 1880 року.

Пізніше бізнес Холлеріта об'єднався з трьома іншими формувати те, що стало IBM!

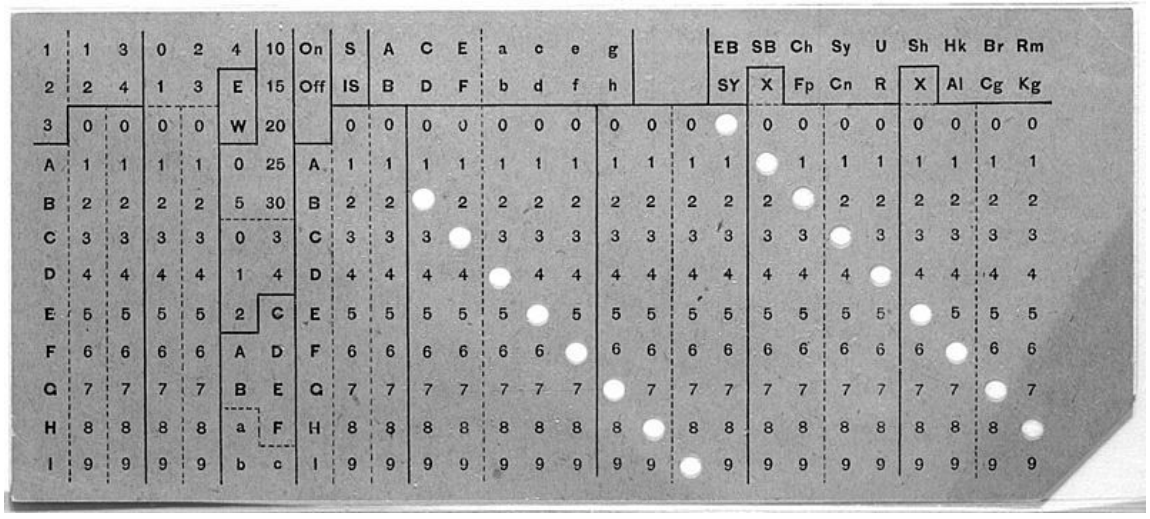
Великі дані - проблема



Герман Голлеріт
(1888-1929)

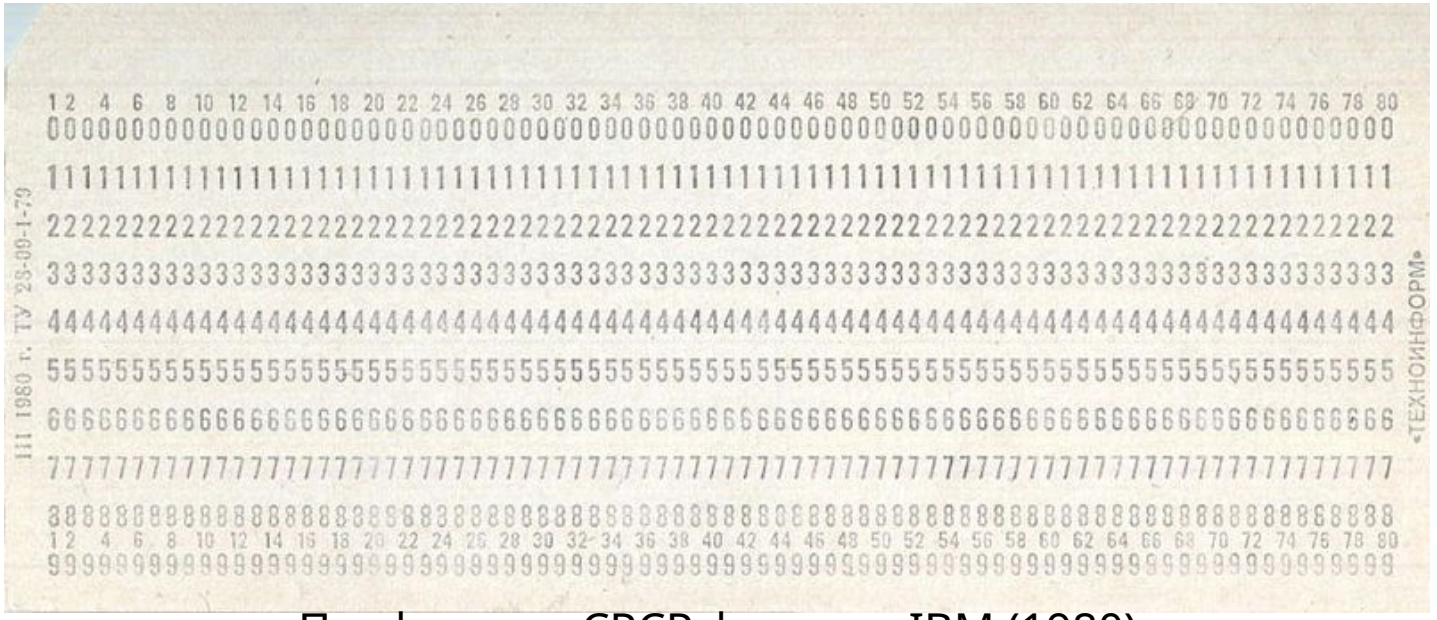
Табулююча машина Холлеріта
з коробкою для сортування
(1890)

Перфоратор картки Холлеріта, який використовувався Бюро перепису населення в США (1940)

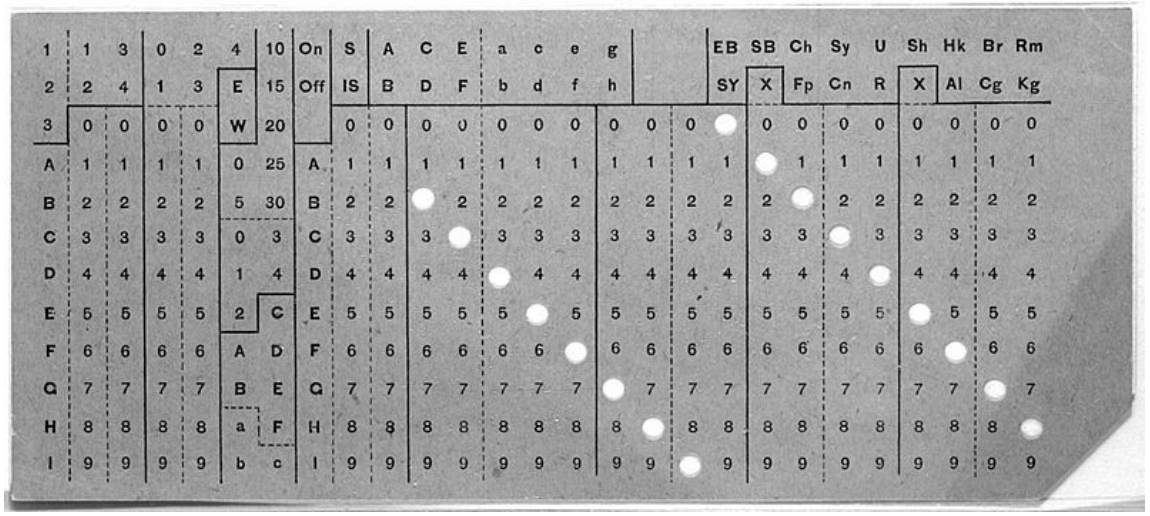


Перфокарта Холлеріта (1895)

Великі дані - проблема

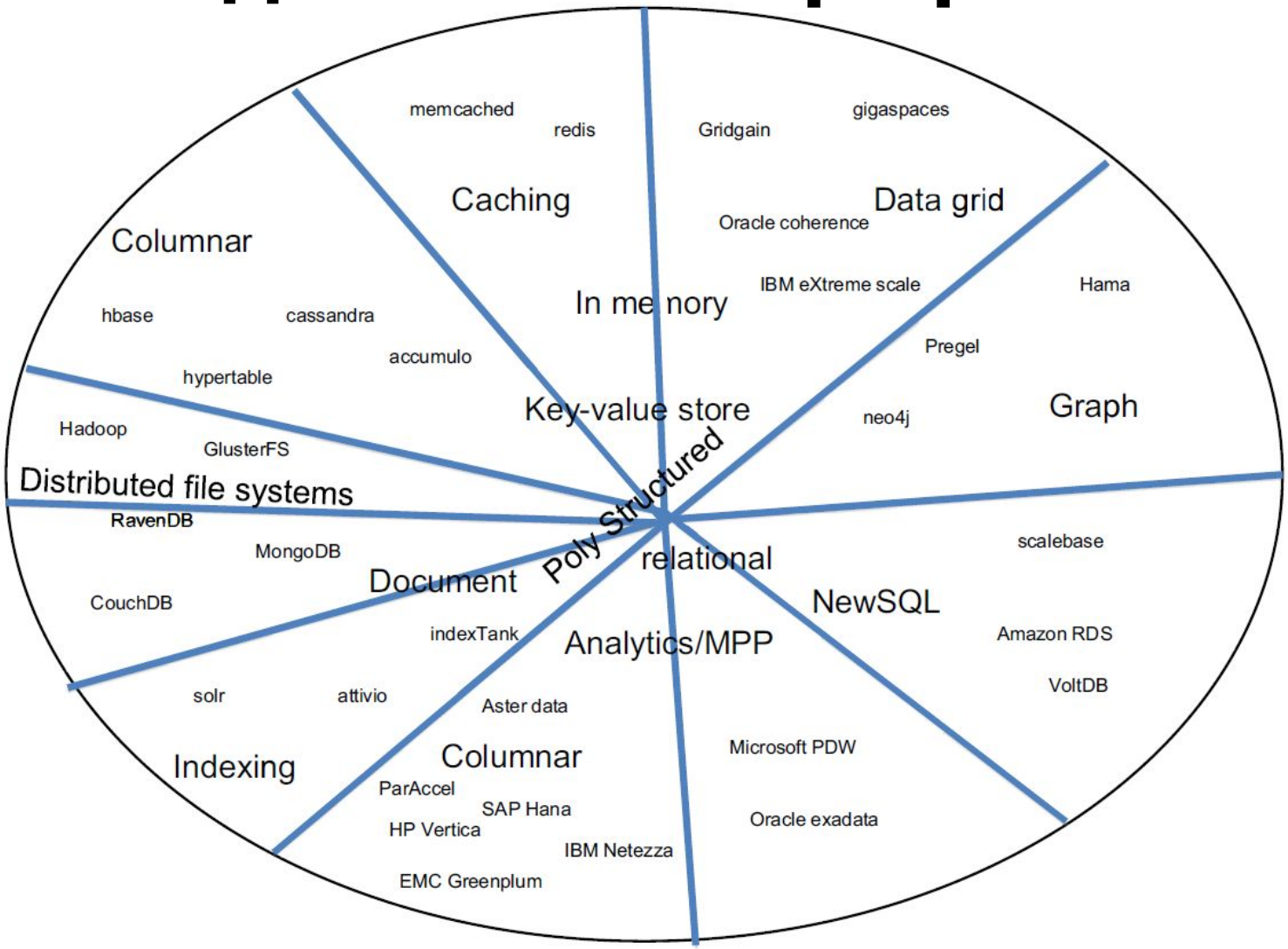


Перфокарта СРСР формату IBM (1980)



Перфокарта Холлеріта (1895)

Великі дані – категорії рішень



Категорії рішень з деякими прикладами,

але … НЕМАЄ вбивчого рішення (!), за винятком …

Великі дані – як вибрати найкраще рішення для вашої системи?

Візьмемо до уваги деякі критерії:

- **Тип вашої організації:**

Підприємствавіддають перевагу**фірмовий**продавці, але **стартапи**—дешевий**відкрите джерело**параметри.

- **Шаблони доступу до даних:**
  - більше читає АБО більше пише,
  - доступ на основі первинного ключа АБО**спеціальні запити**,
  - **прості відносини**(RDBMS?) АБО вперед і назад**поперечні відносини**як прогулянка соціальним графом (базами даних графів?)

Великі дані – як вибрати найкраще рішення для вашої системи?

- **Тип збережених даних:**
  - **структурований**дані (добре для реляційних моделей),
  - **напівструктурований**даних (XML/JSON добре підходить для документів і колонні магазини)
  - **неструктурований**даних (добре для параметрів на основі файлів, таких як Hadoop).

Частота зміни схеми даних:

  - в основному**фіксований**схеми (параметри відношення?),
  - постійно змінюється**схеми**(рішення документів?)

- **Необхідна затримка:**
  - швидкий доступ до даних (**я**п-**М**Еморі**БД**->**IMDB**рішення),
  - звичайний доступ до даних (стандарт**на диску БД**рішення)

Поточні виклики –

Інтернет речей

…

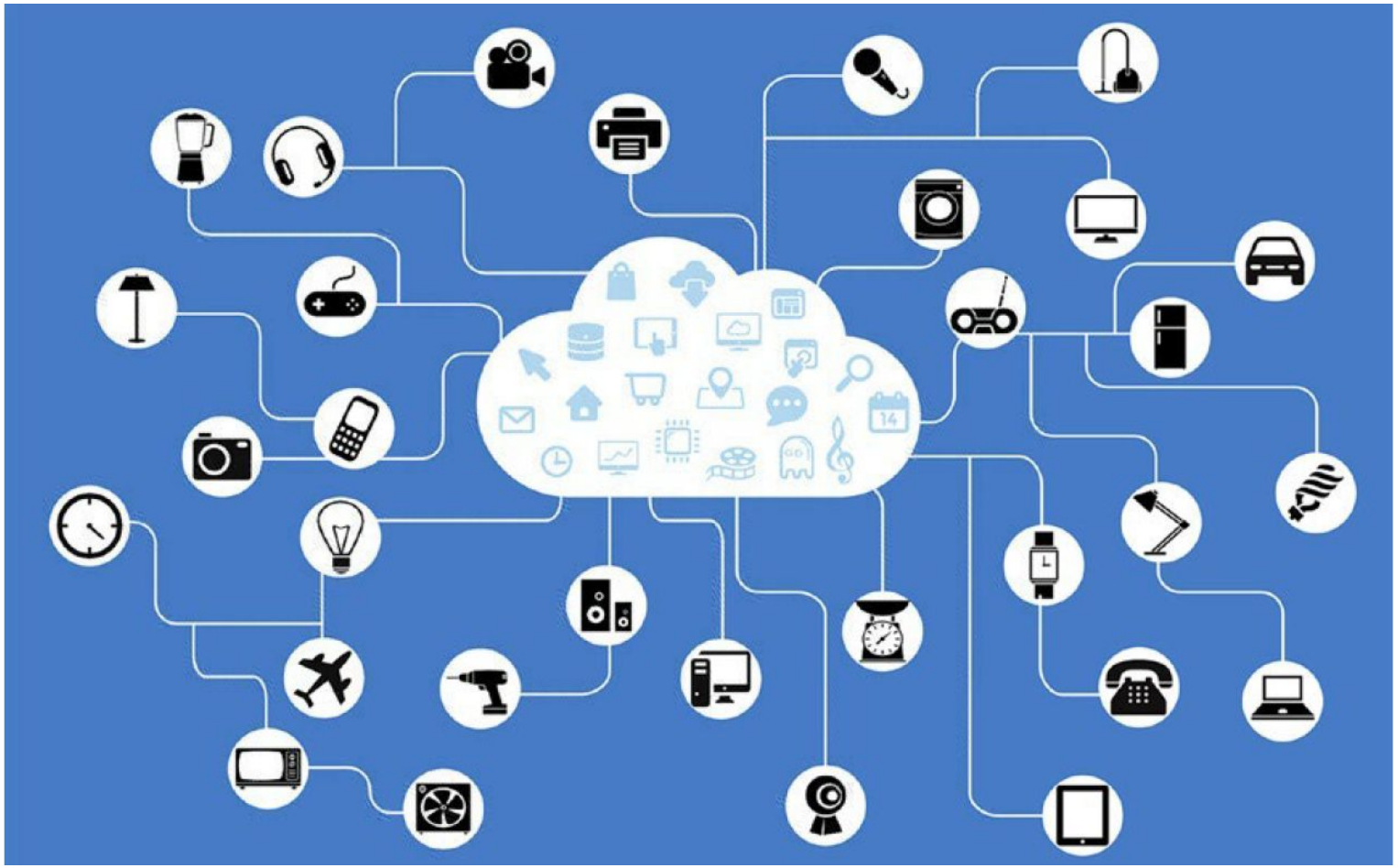
де набагато більше великих даних

з'явиться

…

генерується понад 50 мільярдами пристроїв Edge Layer

Що таке IoT-рішення?

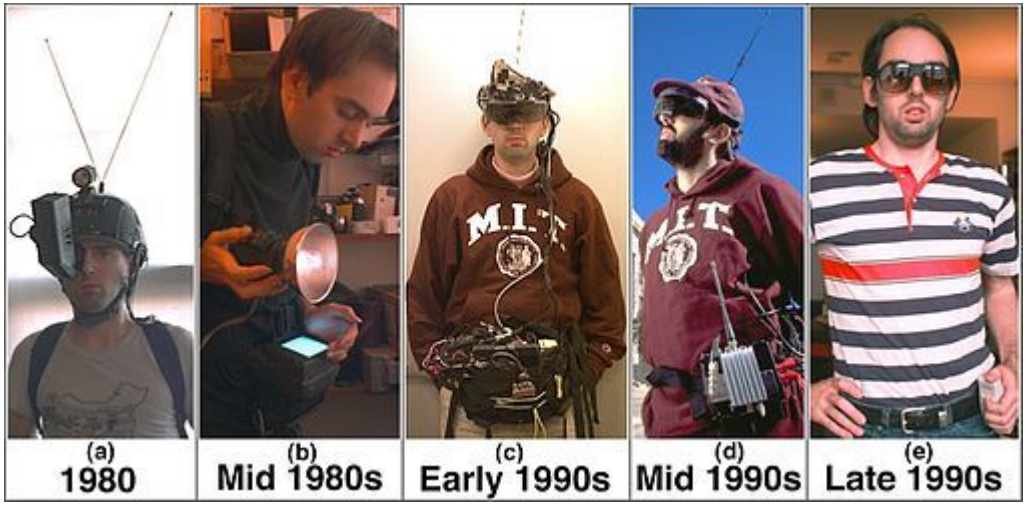


Розроблено IoT**для отримання даних і виконання дій**

приблизнощо завгодно**від**де завгодно**в**будь-який**час**

обчислювальна техніка

Королева Єлизавета I приймає
наручний годинник від Роберта Дадлі (1571)



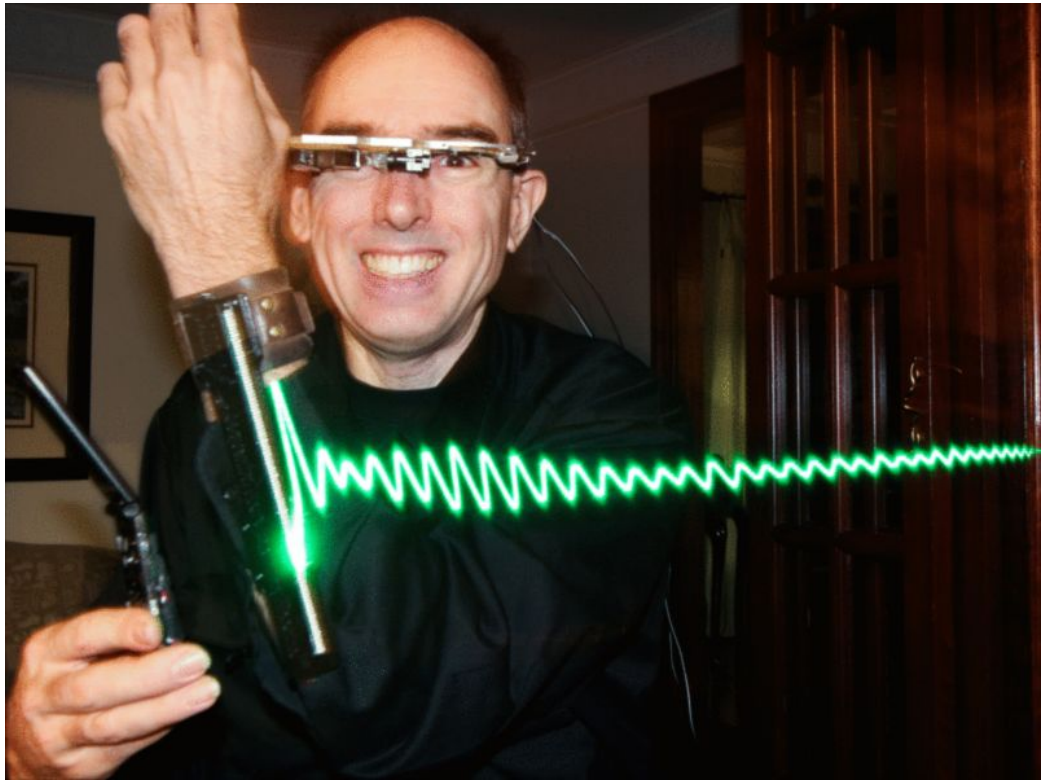
Еволюція переносного комп’ютера WearComp Стіва Манна
від рюкзака до нинішньої прихованої системи

Носимі комп'ютериє мініатюрні електронні пристрої, які носять під, з або поверх одягу.

Застосування:

- сенсорна інтеграція,
- поведінкове моделювання,
- системи моніторингу охорони здоров'я,
- управління обслуговуванням,
- мобільні телефони,
- електронний текстиль,
- дизайн одягу,
- фібертроніка

Приклад сучасного Інтернету речей: Стів Манн – піонер мобільних пристроїв



Стів Манн із трьома своїми винаходами:

- **Цифрові окуляри EyeTap**,
- **Розумний годинник**,
- **SWIM (машина для друку послідовних хвиль)**

Феноменологічна візуалізація доповненої реальності радіохвилі зі смартфона.

Стів Манн(1962)

- Доктор філософії медіа-мистецтва (1997);
- зародок групи Wearable Computing в MIT;
- тепер професор Університету Торонто.

Він є «батьком одягу».

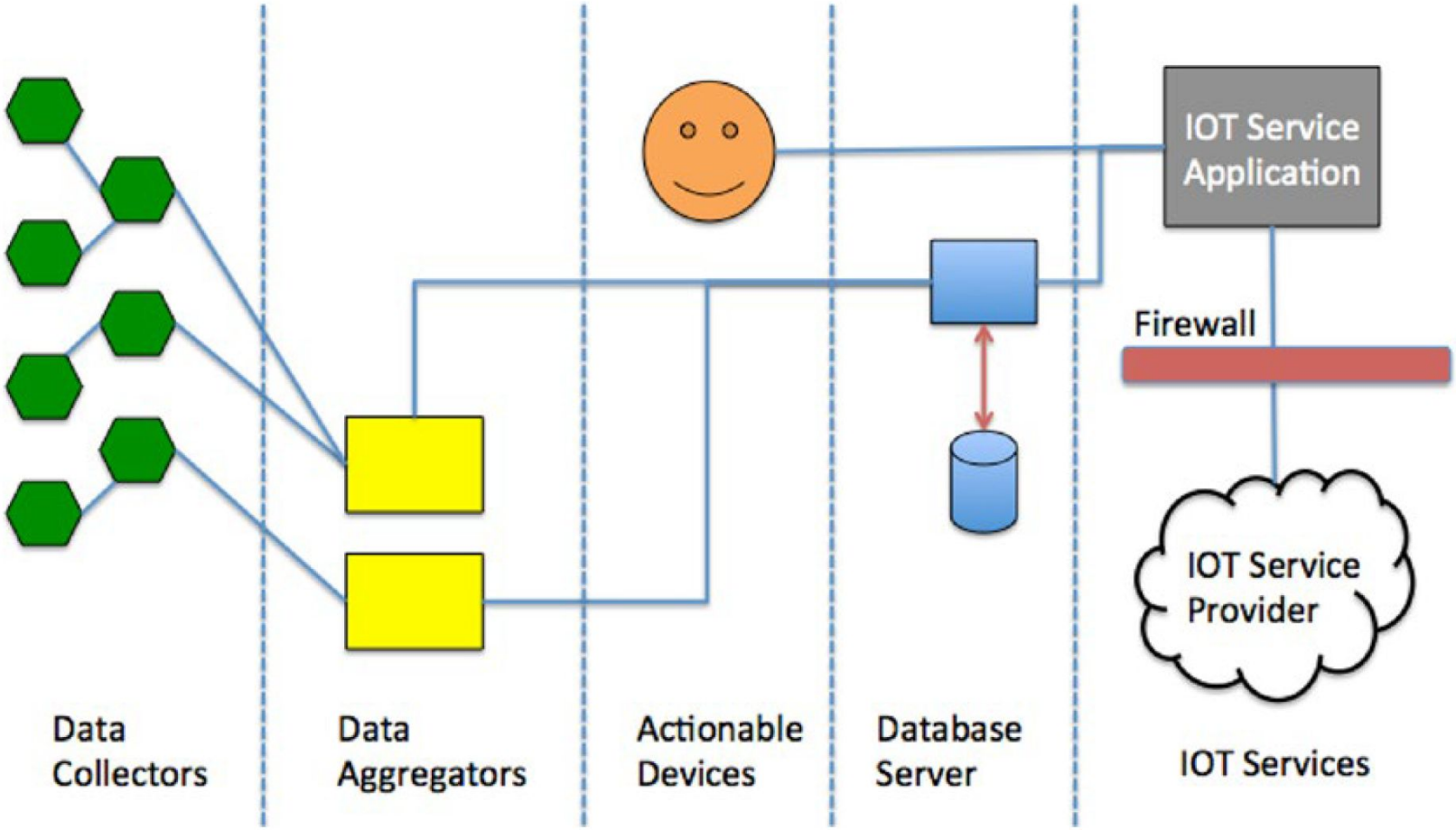
обчислення“.

Він створив перший універсальний переносний комп’ютер,

на відміну від носимих пристроїв, які виконують 1 спец функціонувати як хронометраж

(наприклад, наручний годинник)

Архітектура пристрою IOT



Архітектура пристрою IOT

- **Колектор даних**: датчик, пристрій IOT тощо, який створює дані з якоїсь події чи спостереження.
- **Агрегатор даних**: вузол (вбудований контролер, мікроконтролер, невеликий комп’ютер тощо), який отримує інформацію від одного чи кількох збирачів даних. Його мета — агрегувати та доповнювати дані для зберігання на наступному шарі.
- **Дієвий пристрій**: IOT-пристрій, який надає деякі функції, якими керує користувач, наприклад переміщення датчика, керування замками тощо.
- **Сервер бази даних**: вузол, як правило, сервер, який зберігає дані зібрані для подальшого пошуку та аналізу.
- **Сервіси IoT**: система SO, яка забезпечує рівень доступу до сервера бази даних і робочих пристроїв. Це можуть бути системи SO, розташовані всередині чи поза брандмауером рішення. Системи SO зазвичай є Інтернет-серверами або хмарні служби, які дозволяють користувачам переглядати дані та керувати ними дієві пристрої.

Поточні виклики –

Прийняття рішень

на базі Big Data

Прийняття рішень – проблема

проблема:система працює з величезною кількістю мультимедійних даних і повинна сортувати багатоканальні взаємодії (голос, електронні пошта, чати, пости, ...); взаємодії відбуваються пакетами або в режимах реального часу; їх має обробляти бізнес логіку відповідно до їх категорії.

рішення:ми повинні створити SO-систему на основі відомі шаблони для класифікації цих багатоканальних взаємодій у поданні великих даних (наприклад, застосувати Hadoop)

Прийняття рішень -

"There are no-longer any experts except Cambridge Analytica. They were Trump's digital team who figured out how to win"

Frank Luntz, Political Pollster

Прийняття рішень -

ПОЛИТИКА

Расследование Das Magazin: как Big Data и пара ученых обеспечили победу Трампу и Brexit

By The Insider *@the_ins_ru* · *On 06.12.2016*

184707 просмотров

http://theins.ru/politika/38490

Michael W. Bader

REIGN OF THE ALGORITHMS

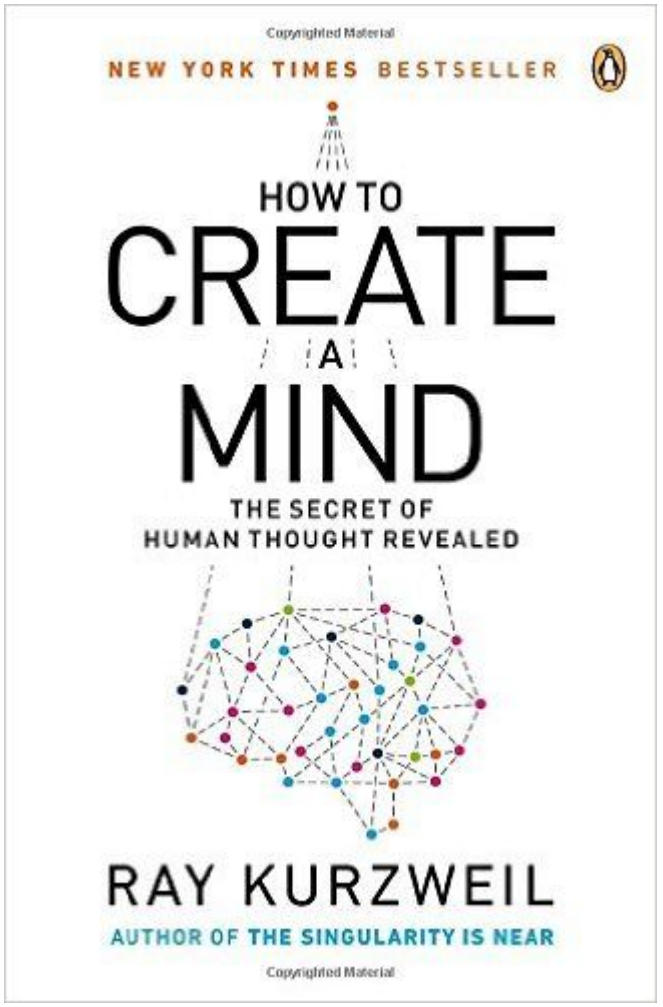
How "artificial intelligence" is threatening our freedom

Michael W. Bader deals with the topic of "artificial intelligence" (AI) from the point of view of the incapacitating effect these new technologies have on society. Apart from the highly alarming implementation of "artificial intelligence" for autonomous weapons systems or the manipulation of elections, a closer look is taken at the conception of humankind and society that underlies these developments. In the author's opinion, creating a socially responsible AI that caters for the common good and serves the benefit of the many rather than of the few is critically important.

I. Introduction

The author's previous essay, under the title "Against monopolism and libertarianism", dealt with the dangers of information capitalism that arose from Silicon Valley. It was observed that more and more power was being accumulated in the hands of a few companies outside of democratic control and old libertarian perspectives and new ideologies of the improvement of the world were coalescing for this purpose. Without question, the new tycoons, such as Google, Facebook, Airbnb and Uber are changing our world in their own image.[1]

Monopolism is back in fashion and democracy seems to be an outdated and cumbersome technology that gets in the way of the highly motivated entrepreneurs and their freedom. When discussing this topic, it is important always to maintain a certain critical distance in spite of too much enthusiasm for new exiting gimmicks and innovations of the large internet companies. This is important because their activities are giving rise to a completely unbridled form of information capitalism that is doing business by constantly collecting personal data on citizens - without asking for permission.



df

Прийняття рішень – Ринок

«... попит на спеціалістів з обробки даних перевищує пропозицію. Ці професіонали отримують високі зарплати та великі пакети опціонів на акції...»

Емілі Вальц. *Чи Data Scientist найсексуальніша робота нашого часу?* Спектр IEEE (2012)

«... лише Сполучені Штати стикаються з нестачею від 140 000 до 190 000 даних вчені з відповідними навичками..."

Джеймс Маніка, Майкл Чуй, Бред Браун, Жак Бугін, Річард Доббс, Чарльз Роксбург, Анджела Ханг Баєрс. *Великі дані: наступний рубіж для інновацій, конкуренції та продуктивності.* Глобальний інститут McKinsey (2011)

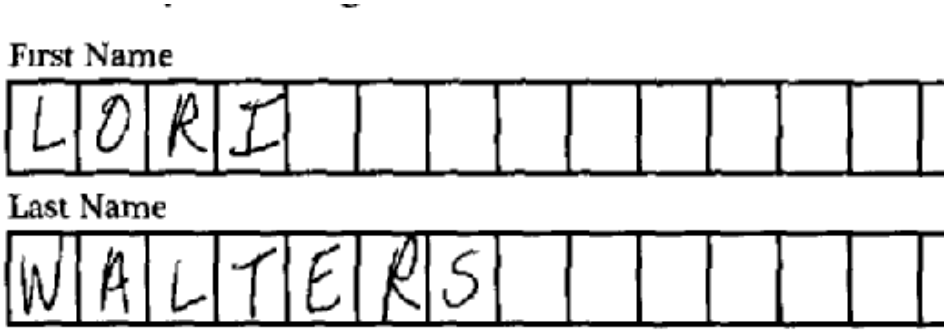
«... наука про дані — це найсексуальніша робота 21-го століття... нове покоління професіонал володіє ключем до використання можливостей великих даних. Але цих спеціалістів нелегко знайти — і конкуренція вони жорстокі"

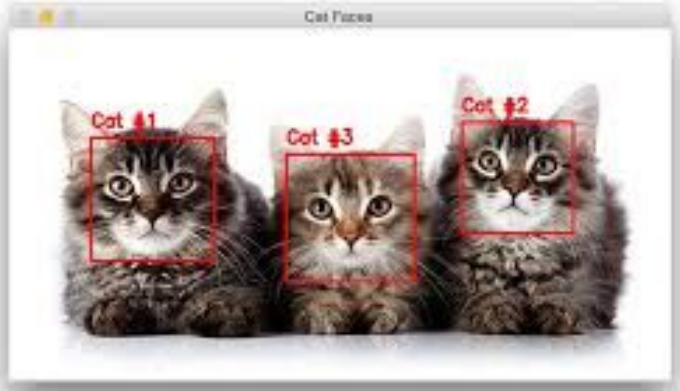
Девенпорт, Томас Х. і Діджей Патіл. «Науковий спеціаліст: найсексуальніша робота 21 століття». Harvard Business Review (2012)

Прийняття рішень – рішення: машинним навчанням

- **Машинне навчання — це здатність навчати комп’ютер без його явного програмування**

- **Приклади використовуються для навчання комп’ютерів виконувати завдання, які було б важко запрограмувати**





Машинне навчання: типи

- Контрольоване навчання
  - Дані тренування позначені
  - Ціль — правильно позначити нові дані
- Навчання з підкріпленням
  - Дані навчання не позначені
  - Система отримує зворотний зв’язок за свої дії
  - Мета – виконувати кращі дії
- Навчання без контролю
  - Дані навчання не позначені
  - Мета полягає в тому, щоб класифікувати спостереження

Машинне навчання: програми

- Розпізнавання рукописного тексту
  - перетворювати написані букви в цифрові
- Мовний переклад
  - перекладати усну та/або письмову мови (наприклад, Google Translate)
- Розпізнавання мови
  - конвертувати голосові фрагменти в текст (наприклад, Siri, Cortana та Alexa)
- Класифікація зображень
  - позначати зображення відповідними категоріями (наприклад, Google Photos)
- Автономне водіння
  - дозволити їздити автомобілям

Машинне навчання: особливості

- Особливості – це спостереження, які використовуються для формування прогнозів
  - Для класифікації зображень пікселі є характеристиками
  - Для розпізнавання голосу функціями є висота та гучність зразків звуку
  - Для автономних автомобілів доступні дані з камер, датчиків дальності та GPS

- Вилучення релевантних функцій є важливим для побудови моделі
  - Час доби не має значення при класифікації зображень
  - Час доби має значення при класифікації електронних листів, оскільки СПАМ часто виникає вночі

- Поширені типи функцій у робототехніці
  - Пікселі (дані RGB)
  - Дані про глибину (ехолот, лазерний далекомір)
  - Рух (значення кодера)
  - Орієнтація або прискорення (гіроскоп, акселерометр, компас)

Машинне навчання: заходи для Успішність класифікації

- **Справжній позитивний результат:**
  - **Правильно визначено як релевантне**
- **Справжній негатив:**
  - **Правильно визначено як неактуальне**
- **Хибно позитивний:**
  - **Неправильно позначено як релевантне**
- **Помилково негативний:**
  - **Неправильно позначено як нерелевантне**

Машинне навчання – сценарій використання:

Визначте котів

Прогноз: 

зображення: 

правда	**правда**	**помилковий**	**помилковий**
Позитивний	**Негативний**	**Негативний**	**Позитивний**

Машинне навчання –Метрики:

Точність, запам'ятовування та точність

- **Точність**
  - **Відсоток позитивних міток, які є правильними**
  - **Точність= (# справжніх позитивних результатів) / (# справжніх позитивних результатів + # помилкових позитивних результатів)**
- **Відкликати**
  - **Відсоток позитивних прикладів, які правильно позначені**
  - **Відкликати= (# справжніх позитивних) / (# справжніх позитивних + # помилкових негативних)**
- **Точність**
  - **Відсоток правильних міток**
  - **Точність= (# істинно позитивних + # істинно негативних) / (кількість зразків)**

Машинне навчання –Дані: Навчання та тестування … Переобладнання

- **Дані навчання**

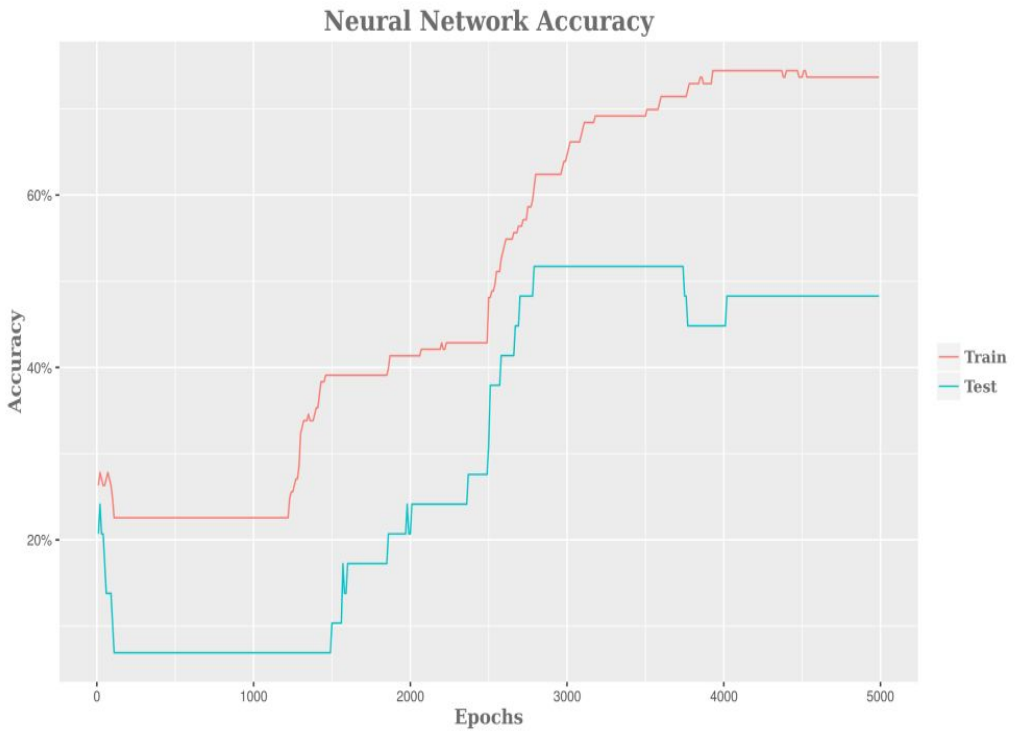
  дані, які використовуються для вивчення моделі

- **Тестові дані**

  дані, які використовуються для оцінки точність моделі

- **Переобладнання**

  Модель добре показує себе на тренуваннях даних, але погано на тестових даних



Машинне навчання –Модель: Упередження/відхилення та сценарії

- **Упередженість: очікувана різниця між прогнозом моделі та правдою**
- **Дисперсія: наскільки модель відрізняється між навчальними наборами**

- **Модельні сценарії**
  - **Високе зміщення: модель робить неточні прогнози щодо даних навчання**
  - **Висока дисперсія: модель не поширюється на нові набори даних**
  - **Низьке зміщення: модель робить точні прогнози на основі даних навчання**
  - **Низька дисперсія: модель узагальнюється для нових наборів даних**

Машинне навчання –Алгоритми:

Контрольоване навчання

- Лінійна регресія
- Дерева рішень
- Підтримуйте векторні машини
- К-найближчий сусід
- Нейронні мережі

Машинне навчання –Каркаси:

Контрольоване навчання

Інструмент	Використання	Мову
Scikit-Learn	класифікація, Регресія, кластеризація	Python
Spark MLlib	класифікація, Регресія, кластеризація	Scala, R, Java
Weka	класифікація, Регресія, кластеризація	Java
Кафе	Нейронні мережі	C++, Python
TensorFlow	Нейронні мережі	Python
H2O	класифікація, Регресія, кластеризація, нейронні мережі	R, Python

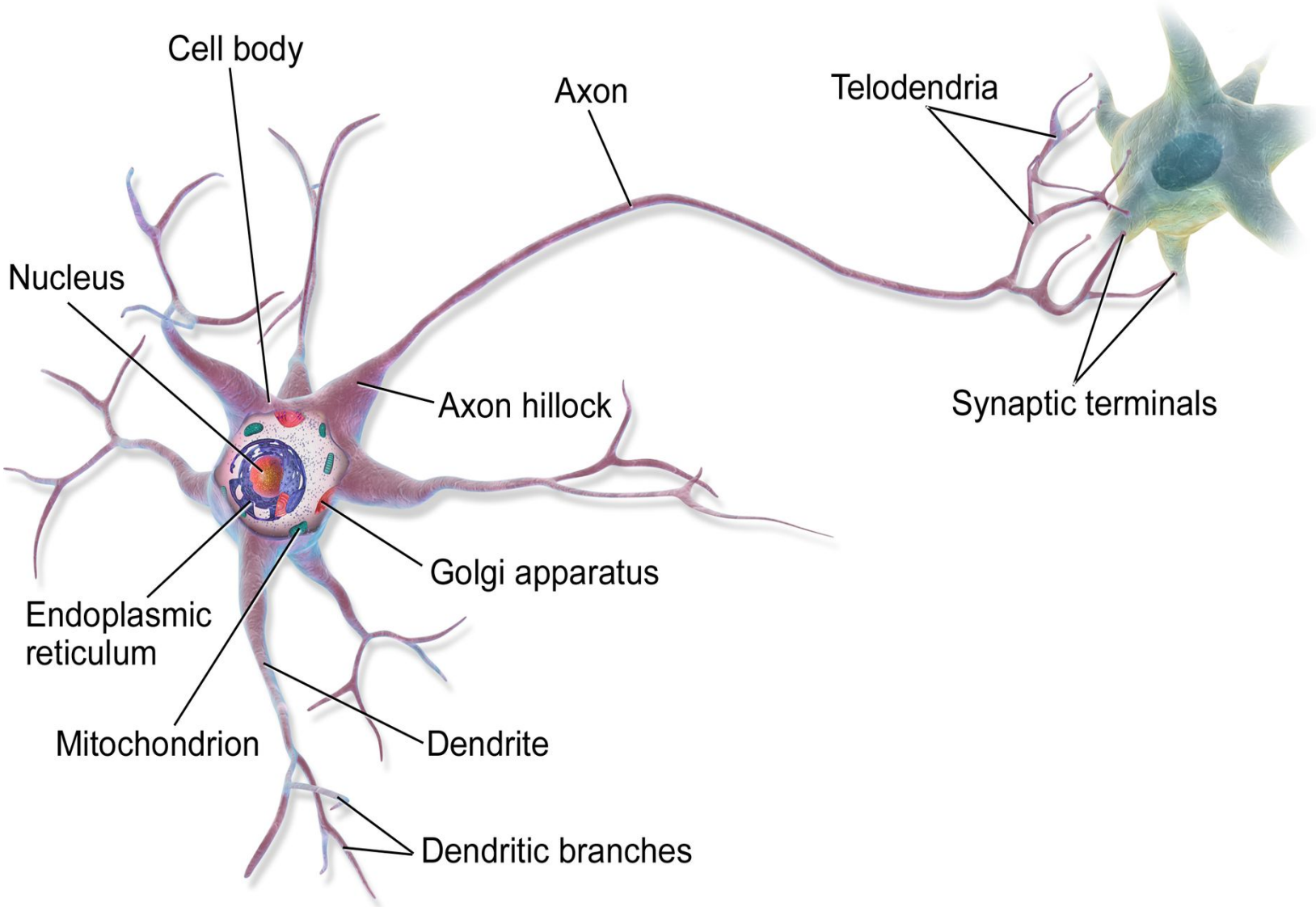
Поточні виклики –

Прийняття рішень

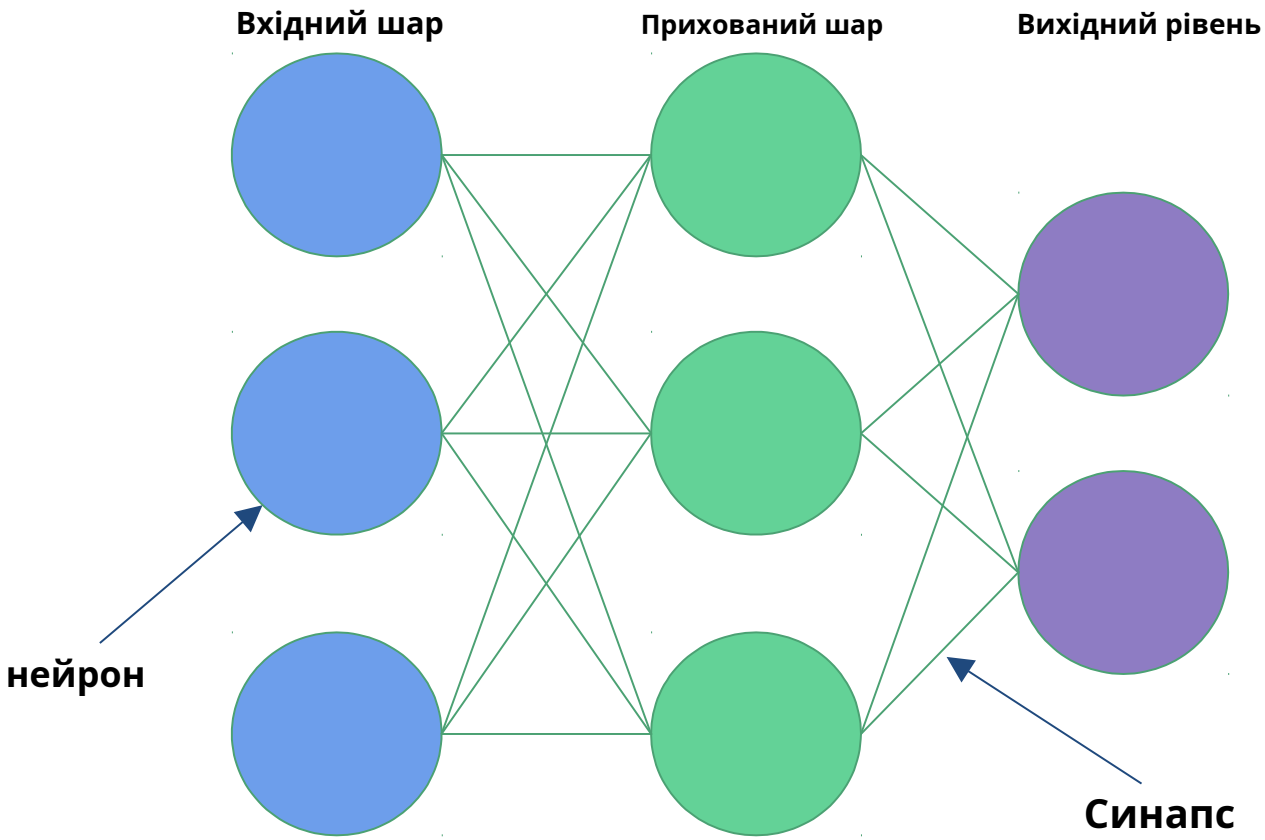
від Neural Networks

Нейронні мережі:

Біологічне натхнення



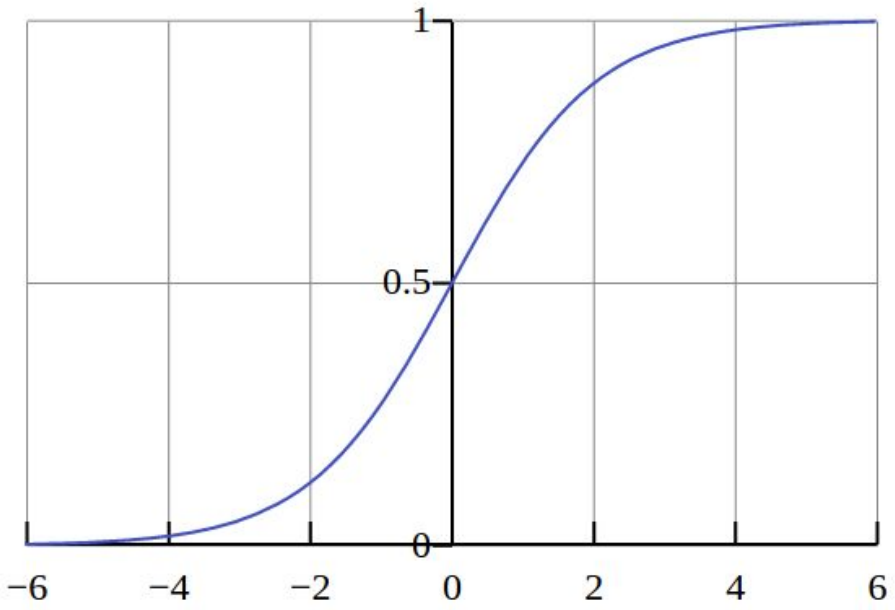
Нейронні мережі:
Архітектура



Нейронні мережі:
Функції активації

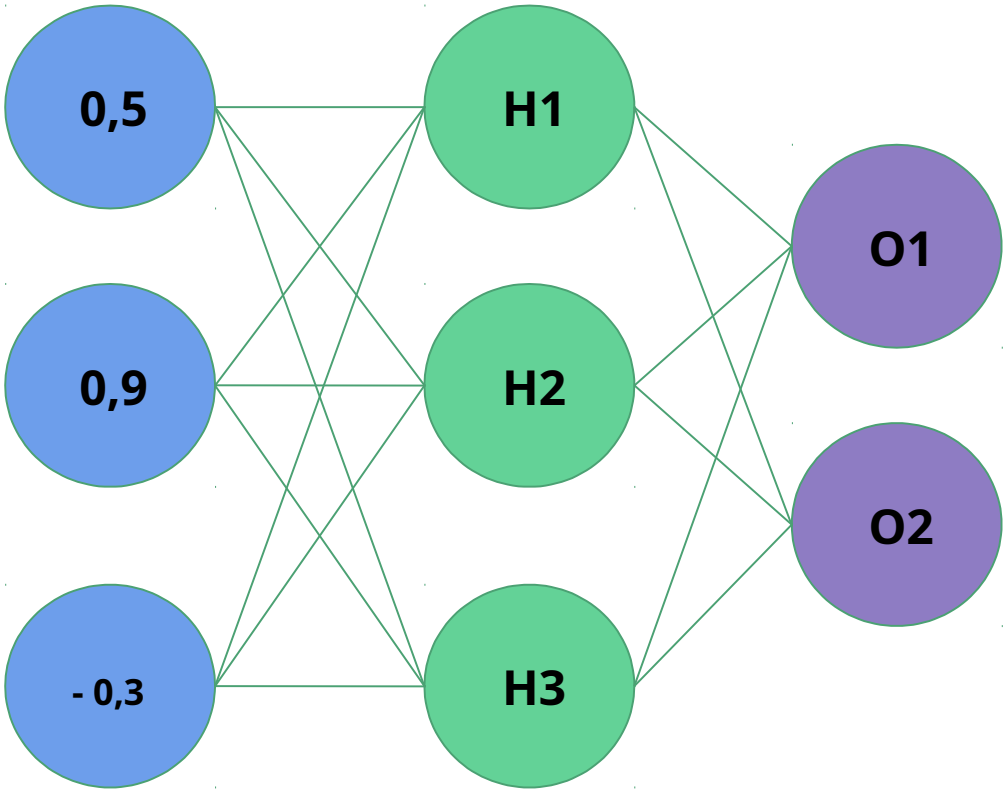
- Функції активації застосовуються до входів кожного нейрона
  - Загальною функцією активації є сигмоїда

$$S(t) = \frac{1}{1 + e^{-t}}$$



Нейронні мережі:

Приклад висновку – 1

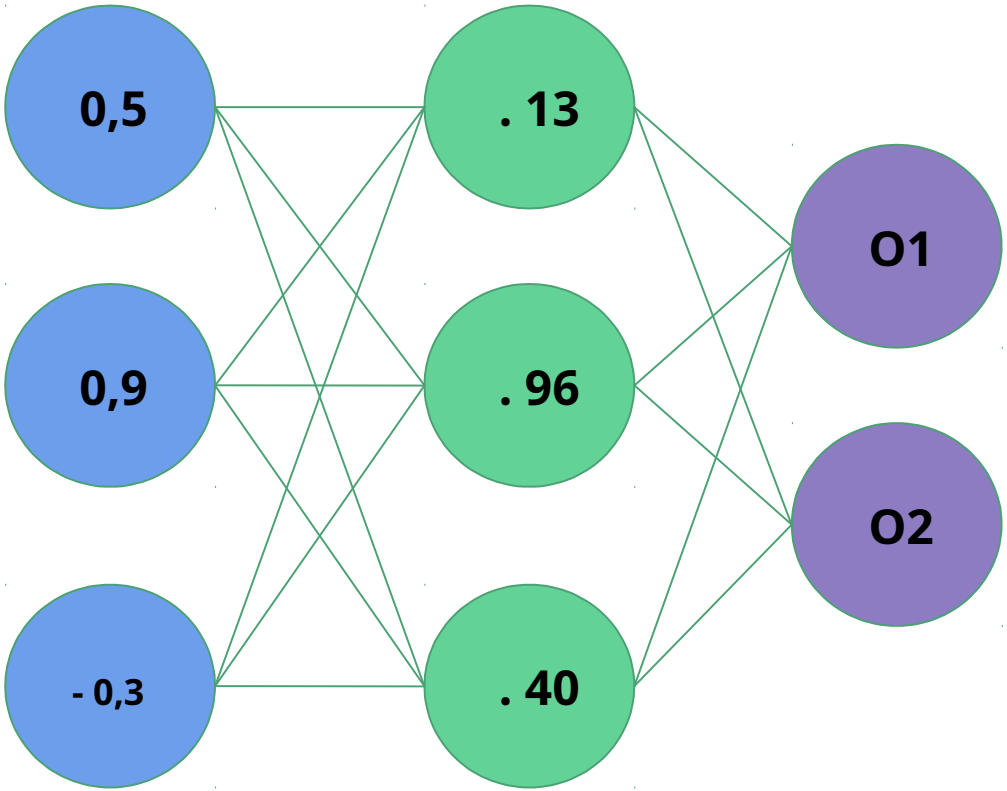


Вага H1 = (1,0, -2,0, 2,0)
Вага H2 = (2,0, 1,0, -4,0)
Вага H3 = (1,0, -1,0, 0,0)

Вага O1 = (-3,0, 1,0, -3,0)
Вага O2 = (0,0, 1,0, 2,0)

Нейронні мережі:

Приклад висновку – 2



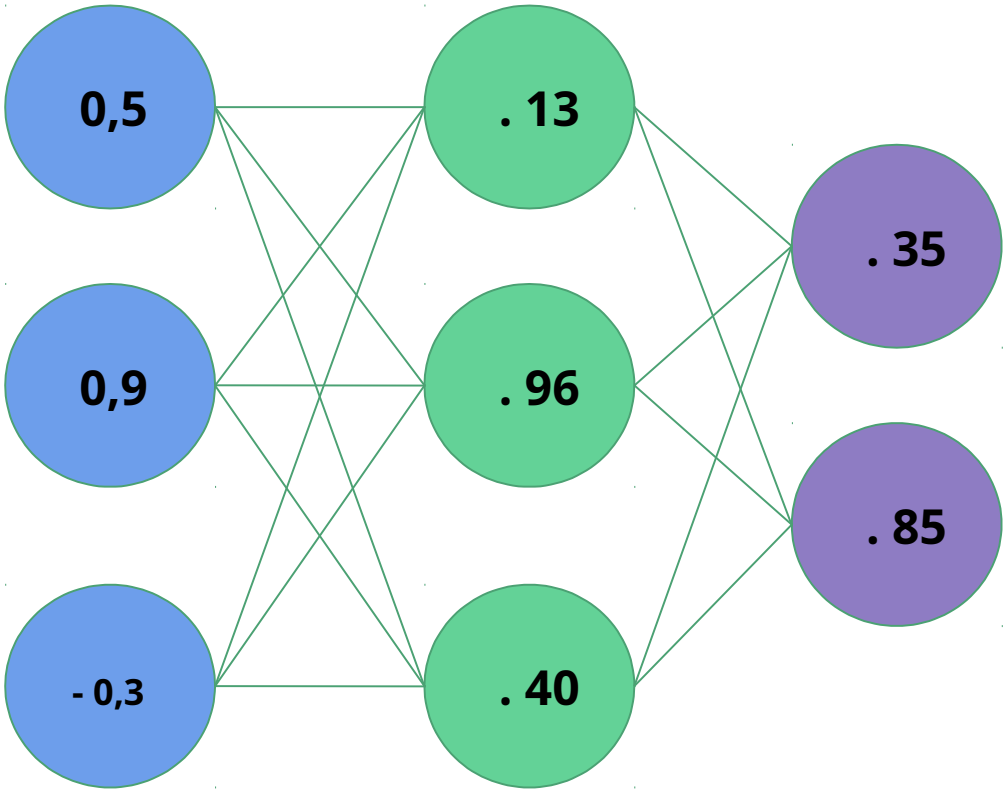
Вага H1 = (1,0, -2,0, 2,0)
Вага H2 = (2,0, 1,0, -4,0)
Вага H3 = (1,0, -1,0, 0,0)

Вага O1 = (-3,0, 1,0, -3,0)
Вага O2 = (0,0, 1,0, 2,0)

H1 = S(0,5 * 1,0 + 0,9 * -2,0 + -0,3 * 2,0) = S(-1,9) = 0,13
H2 = S(0,5 * 2,0 + 0,9 * 1,0 + -0,3 * -4,0) = S(3,1) = 0,96 H3
= S(0,5 * 1,0 + 0,9 * -1,0 + -0,3 * 0,0) = S(-0,4) = 0,40

Нейронні мережі:

Приклад висновку – 3



Вага H1 = (1,0, -2,0, 2,0)
Вага H2 = (2,0, 1,0, -4,0)
Вага H3 = (1,0, -1,0, 0,0)

Вага O1 = (-3,0, 1,0, -3,0)
Вага O2 = (0,0, 1,0, 2,0)

O1 = S(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = S(-.63) = .35 O1 =
S(0,13 * 0,0 + 0,96 * 1,0 + 0,40 * 2,0) = S(1,76) = 0,85

Нейронні мережі:

Приклад висновку – 4 (за допомогою матричних операцій)

Вага H1 = (1,0, -2,0, 2,0)
Вага H2 = (2,0, 1,0, -4,0)
Вага H3 = (1,0, -1,0, 0,0)

Ваги прихованих шарів **Вхідні дані**

$$S\begin{pmatrix} 1.0 & -2,0 & 2.0 \\ 2.0 & 1.0 & -4.0 \\ 1.0 & -1,0 & 0,0 \end{pmatrix} * \begin{pmatrix} 0,5 \\ 0,9 \\ -0,3 \end{pmatrix} = S\begin{pmatrix} -1,9 & 3.1 & -0,4 \end{pmatrix} =$$

Виходи прихованого шару

. 13 ,96 0,4

Нейронні мережі:Навчання – зворотне поширення та градієнтний спуск

- Порядок проведеннянавчанняНейронні мережі
  - Виконайте висновок на навчальній множині
  - Обчисліть похибку між прогнозами та фактичними мітками навчального набору
  - Визначте внесок кожного нейрона в помилку
  - Змініть ваги нейронної мережі, щоб мінімізувати помилку

- Внески помилок розраховуються за

  Зворотне поширення

- Мінімізація помилок досягається за рахунокГрадієнтний спуск

Нейронні мережі:Навчання – Зворотне поширення – Проблема

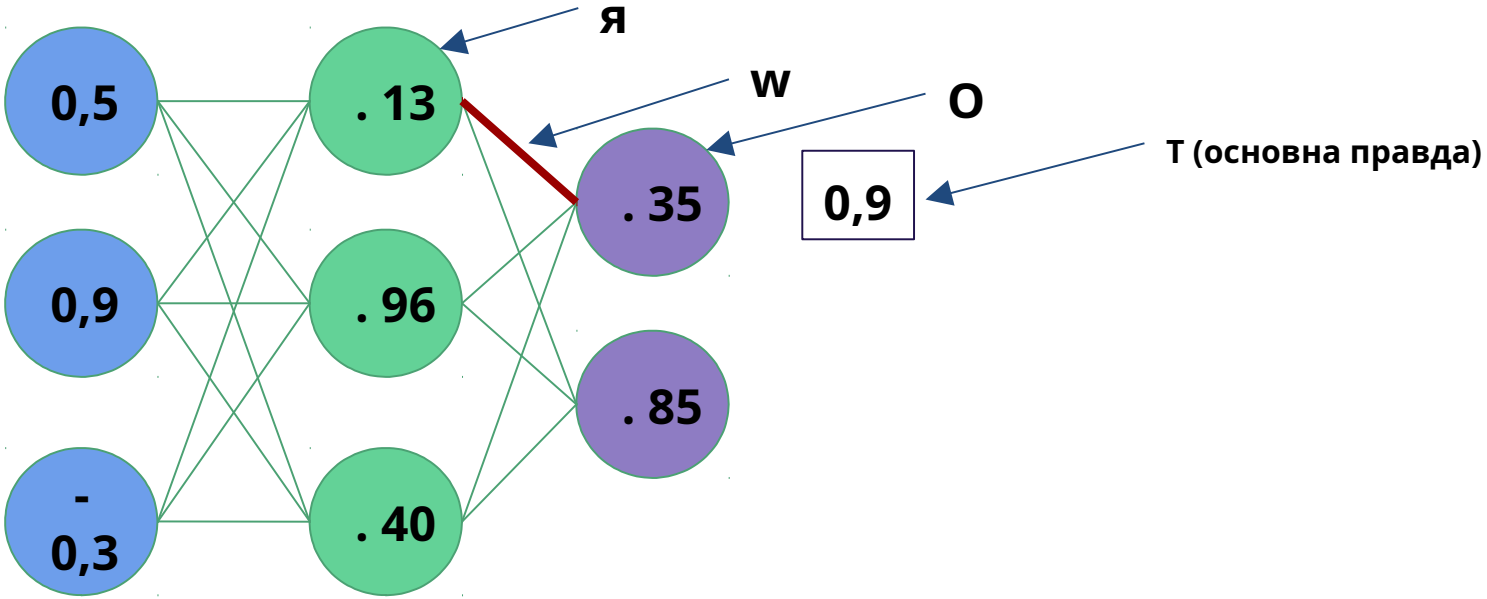
проблема:

Які ваги слід оновити і на скільки?

В полі зору:

Використовуйте похідну від помилки щодо ваги, щоб призначити «винуватця»

Нейронні мережі:Навчання – Зворотне поширення – Приклад



$$\frac{\partial E}{\partial w} = I \cdot (O - T) \cdot O \cdot (1 - O)$$

$$\frac{\partial E}{\partial w} = .13 \cdot (.35 - .9) \cdot .35 \cdot (1 - .35)$$

Нейронні мережі:Навчання – Градієнтний спуск – Проблема

- Gradient Descent мінімізує помилку нейронної мережі
  - На кожному часовому кроці похибка мережі обчислюється за навчальними даними
  - Потім ваги змінюються, щоб зменшити похибку
- Градієнтний спуск завершується, коли
  - Похибка досить мала
  - Перевищено максимальну кількість кроків за часом

Поточні виклики –

Прийняття рішень

шляхом глибокого навчання

Добре, на сьогодні досить…

Нейронні мережі

Лекція_02

Доброго дня!
Сьогодні лекція 2 пропонується Вам для самостійного так званого самостійного навчання на основі цих Jupyter Notebooks (із логами матеріалів виконаних завдань). Вони є основними:
1 - теоретичні відомості,
2 - прості приклади практичного використання і

дають Вам можливість "погратися" з параметрами та способами реалізації певних методів.
Під час наступної лекції за тиждень ми розберемо їх разом.
У разі проблеми із доступом - пишіть у Телеграм-групу предмету.

****

Слайди лекцій - як ДЕМО-зошити за адресою:
https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

****

Лекція 02 - Основи NN - DEMO від Jupyter Notebooks

****

* Довідник:
Lecture_02_DEMO - 2 демо:

DEMO_01_Activation Functions_EXAMPLE.ipynb https://drive.google.com/file/d/1xwNUPXlHezqR_2oo4XLn-8dHul91y356/view?usp=sharing

DEMO_02_Нейронна мережа з нуля_EXAMPLE.ipynb https://drive.google.com/file/d/1cPXcJfVI35lsmw-o_l60xyflJNzogWfJ/view?usp=sharing

****

* Довідник:
Lecture_02_DEMO_NG - 2 демонстрації на основі (С) курсу Coursera (з деякими рішеннями!):

01_LR_BuildNN_EXAMPLE.ipynb https://drive.google.com/file/d/1nPFleXe__PwNQjuXPXl-hgGlh3wayNDu/view?usp=sharing

02_NN_with_1-N_layers_EXAMPLE.ipynb https://drive.google.com/file/d/1RWgwrSMi4rl_sr_EC2IRQG64m7frfwCj/view?usp=sharing

****

Попередні умови: монтувати Google Drive + скопіювати необхідні файли (бібліотеки + набори даних)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
! pwd
```

```
/content
```

```
! ls /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DEMO
```

```
'DEMO_01_Activation Functions_EXAMPLE.ipynb'
'DEMO_01_Activation Functions.ipynb'
'DEMO_01_Activation Functions_UA.ipynb'
'DEMO_02_Neural Network from scratch_EXAMPLE.ipynb'
'DEMO_02_Neural Network from scratch.ipynb'
 images
```

```
! cp -r /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DE
```

```
! ls
```

```
'DEMO_01_Activation Functions_EXAMPLE.ipynb'
'DEMO_01_Activation Functions.ipynb'
'DEMO_01_Activation Functions_UA.ipynb'
'DEMO_02_Neural Network from scratch_EXAMPLE.ipynb'
'DEMO_02_Neural Network from scratch.ipynb'
 drive
 images
 sample_data
```

```
# Хитрість для використання статичних зображень у Google Colaboratory
path = '/usr/local/share/jupyter/nbextensions/google.colab'
!cp -r {path}/* .
!rm -r {path}
!ln -s /content {path}
# змінити базовий тег
from google.colab.output import eval_js
def change_base_url():
  eval_js("""
  var base = document.createElement('base')
  base.href = 'https://localhost:8080/nbextensions/google.colab/'
  document.head.prepend(base)
  """)
# зробити так, щоб він запускався автоматично в кожній клітинці
get_ipython().events.register('pre_run_cell', change_base_url)
```

```
DATA_HOME = "/content/drive/MyDrive/2022_COLAB_ML"
from IPython.core.display import HTML
def css_styling():
    styles = open(DATA_HOME + "/styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Функції активації

Функція активації, також відома як функція передачі, відіграє життєво важливу роль у нейронних мережах. Він використовується для введення нелінійності в нейронні мережі. Як ми знали раніше, ми застосовуємо функцію активації до вхідних даних (input), які множаться на ваги (weights) та додаються до зміщення (bias), тобто, $f(z)$, де z = (input * weights) + bias і $f(\cdot)$ є функція активації. Якщо ми не застосовуємо функцію активації, то нейрон просто нагадує лінійну регресію. Метою функції активації є запровадження нелінійного перетворення для вивчення складних основних закономірностей у даних.

Тепер давайте розглянемо деякі з цікавих функцій активації, які часто використовуються.

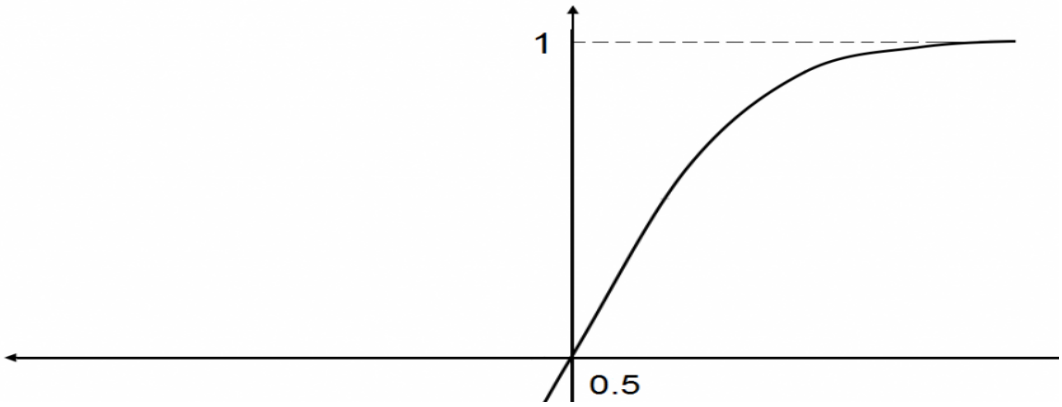
Сигмоїда (sigmoid function)

Сигмоїда (або сигмоїдна функція) є однією з найбільш часто використовуваних функцій активації. Він масштабує значення від 0 до 1. Сигмоїдну функцію можна визначити наступним чином:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Це S-подібна крива, як показано нижче:

```
%%html
<img src="images/6.png" style="width:650px;height:400px;">
```



Вона диференційована, тобто ми можемо знайти нахил кривої в будь-яких двох точках. Він монотонний, що означає, що він або зовсім не зростає, або не спадає. Сигмоїдна функція також відома як логістична функція. Як ми знаємо, що ймовірність лежить між 0 і 1, і оскільки сигмоїдна функція здавлює значення між 0 і 1, вона використовується для прогнозування ймовірності результату. Функцію sigmoid можна визначити в Python наступним чином:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

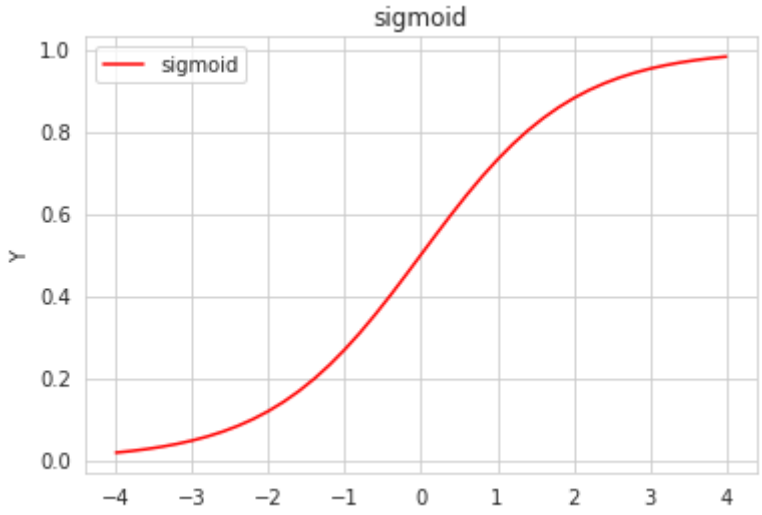
```
x_list = np.linspace(-4, 4, 41)
#x_list = range(-5,5)
```

```
sigmoid_list = []

print('x_list=',list(x_list))

for x in x_list:
    y = sigmoid(x)
    sigmoid_list.append(y)
print('sigmoid_list=',sigmoid_list)
```

```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999
sigmoid_list= [0.01798620996209156, 0.021881270936130476, 0.0265969
```

```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(x_list, sigmoid_list, color='red', label='sigmoid')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('sigmoid')
```

```
plt.legend()
plt.show()
```



Функція tanh

Функція гіперболічного тангенса (tanh) виводить значення від -1 до +1 і виражається таким чином:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Він також нагадує S-подібну криву. На відміну від сигмоїдної функції, яка має центр на 0,5, функція tanh має центр на 0, як показано на наступній діаграмі:

```
%%html
<img src="images/7.png" style="width:650px;height:400px;">
```

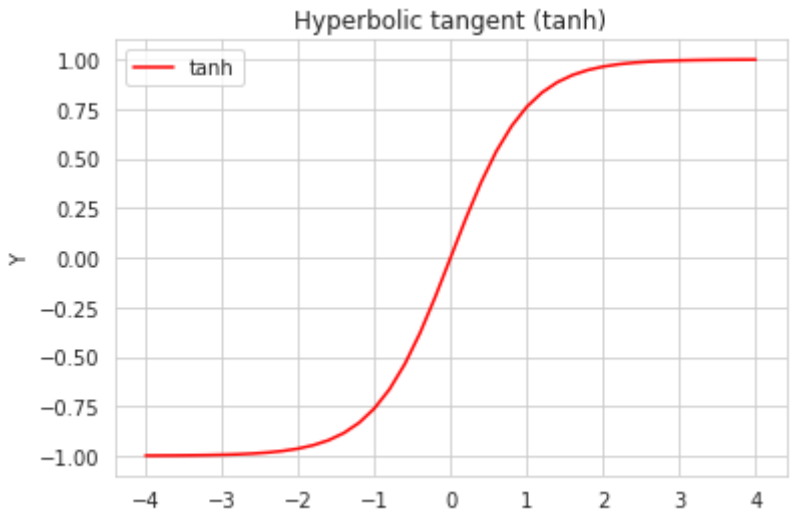
Подібно до сигмоїдної функції, це також диференційована та монотонна функція. Функція tanh реалізована таким чином:

```
def tanh(x):
    numerator = 1-np.exp(-2*x)
    denominator = 1+np.exp(-2*x)
    return numerator/denominator
```

```
tanh_list = []

#x_list = range(-20,20)
print('x_list=',list(x_list))

for x in x_list:
    y = tanh(x)
    tanh_list.append(y)
print('sigmoid_list=',tanh_list)
```

```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999
sigmoid_list= [-0.999329299739067, -0.9989995977858409, -0.99850794
```

```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(x_list, tanh_list, color='red', label='tanh')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Hyperbolic tangent (tanh)')
plt.legend()
plt.show()
```



Функція ReLU

Зрізаний лінійний вузол, або випрямлений лінійний вузол (Rectified Linear Unit - ReLU) є ще однією з найпоширеніших функцій активації. Функція ReLU виводить значення від о до innity. Це в основному кускова функція, і її можна виразити наступним чином::

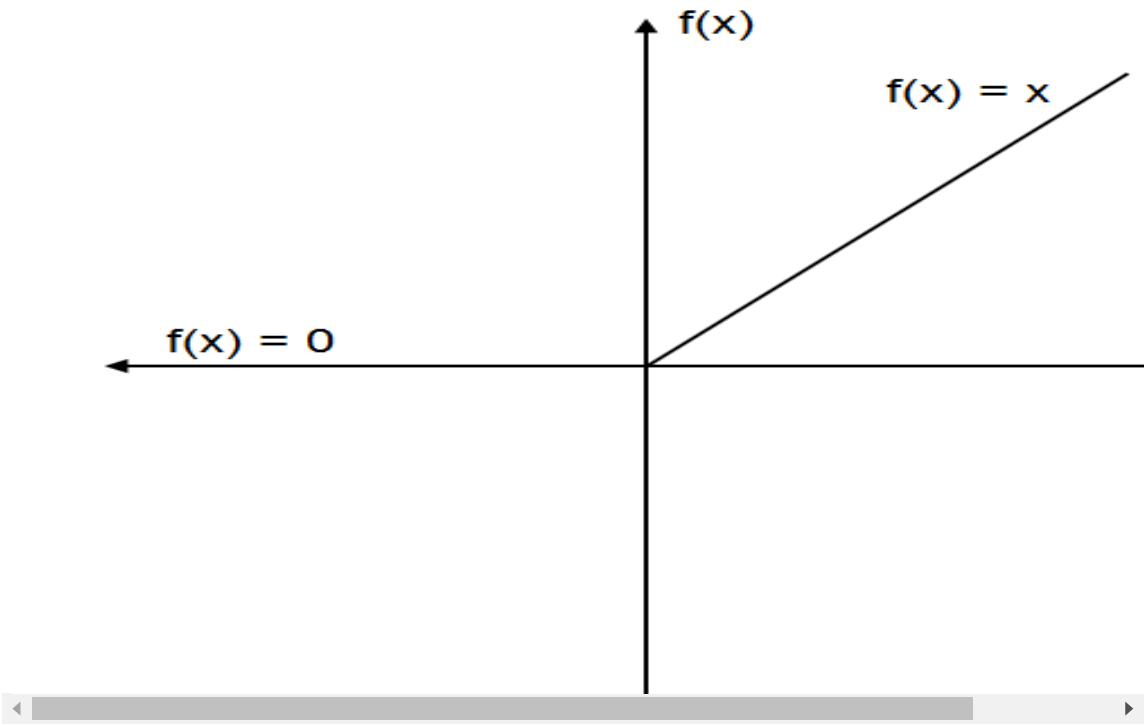
$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Тобто, $f(x)$ повертає нуль, якщо значення x менше нуля, і $f(x)$ повертає x, якщо значення x більше або дорівнює нулю. Це також можна виразити таким чином:

$$f(x) = \max(0, x)$$

Функція ReLU показана на наступному малюнку:

```
%%html
<img src="images/8.png" style="width:650px;height:400px;">
```



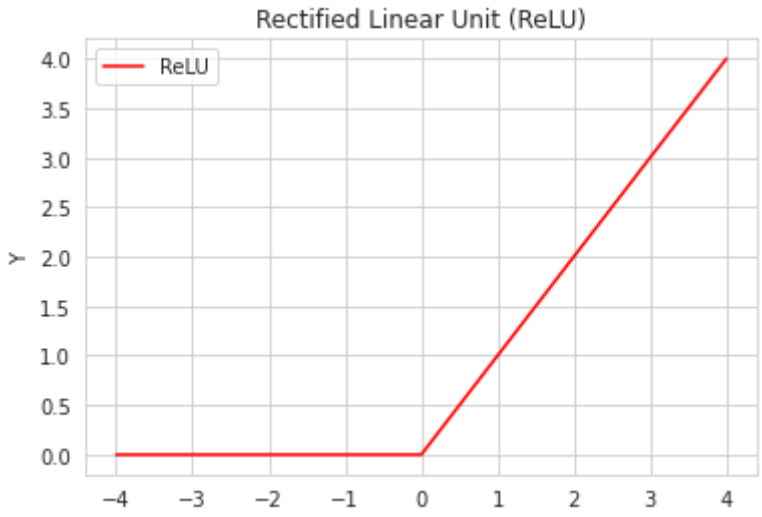
Як ми можемо бачити на попередній діаграмі, коли ми подаємо будь-який негативний вхідний сигнал у функцію ReLU, він перетворює його на нуль. Перешкода бути нульовим для всіх від'ємних значень — це проблема, яка називається "вмираючим" ReLU, і нейрон вважається "мертвим", якщо він завжди виводить нуль. Функцію ReLU можна реалізувати таким чином:

```
def ReLU(x):
    if x<0:
        return 0
```

```
    else:
        return x
ReLU_list = []

#x_list = range(-20,20)
print('x_list=',list(x_list))

for x in x_list:
    y = ReLU(x)
    ReLU_list.append(y)
print('sigmoid_list=',ReLU_list)
```

```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999
sigmoid_list= [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

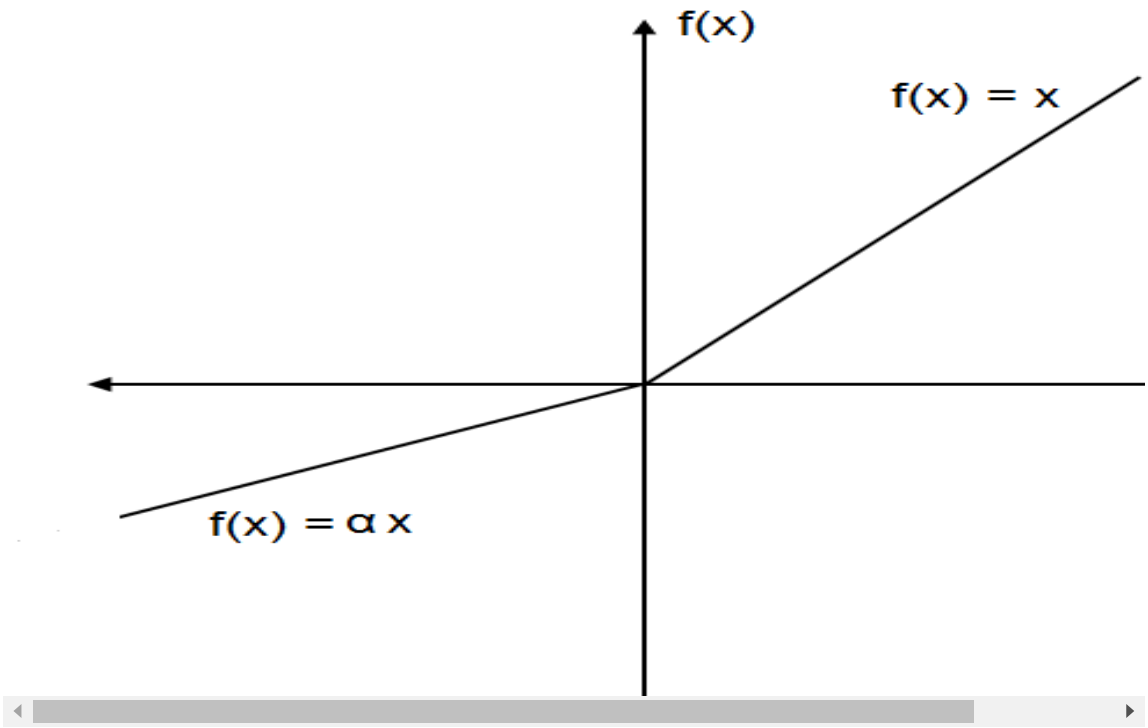
```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(x_list, ReLU_list, color='red', label='ReLU')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Rectified Linear Unit (ReLU)')
plt.legend()
plt.show()
```



Нещільна функція ReLU

Нещільна (leaky) функція ReLU - це варіант функції ReLU, яка вирішує проблему ReLU, яка вмирає. Замість того, щоб перетворювати кожен від'ємний вхід на нуль, він має невеликий нахил для від'ємного значення, як показано:

```
%%html
<img src="images/9.png" style="width:650px;height:400px;">
```



Leaky ReLU можна виразити таким чином:

$$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

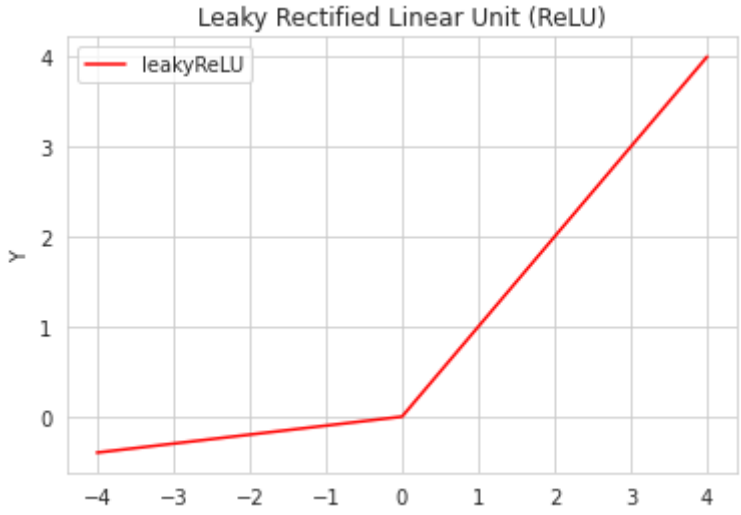
```
def leakyReLU(x,alpha=0.01):
    if x<0:
        return (alpha*x)
    else:
        return x
```

```
leakyReLU_list = []

#x_list = range(-20,20)
print('x_list=',list(x_list))
alpha = 0.1
for x in x_list:
    y = leakyReLU(x,alpha)
    leakyReLU_list.append(y)
print('leakyReLU_list=',leakyReLU_list)
```

```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999
leakyReLU_list= [-0.4, -0.38, -0.36000000000000004, -0.34, -0.32000
```

```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(x_list, leakyReLU_list, color='red', label='leakyReLU')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Leaky Rectified Linear Unit (ReLU)')
```

```
plt.legend()
plt.show()
```



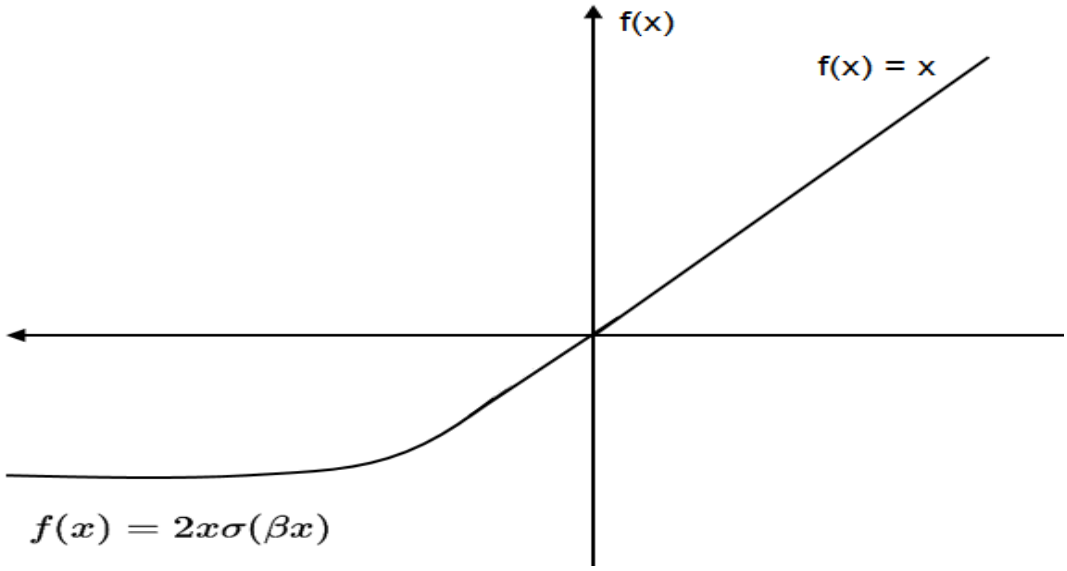
The value α of is typically set to 0.01. The leaky ReLU function is implemented as follows: Instead of setting some default values to α, we can send them as a parameter to a neural network and make the network learn the optimal value of α. Such an activation function can be termed as a Parametric ReLU function. We can also set the value of α to some random value and it is called as Randomized ReLU function.

Значення α зазвичай встановлюється як 0,01. Нещільна функція ReLU реалізована таким чином: замість встановлення деяких значень за замовчуванням для α, ми можемо надіслати їх як параметр до нейронної мережі та змусити мережу дізнатися оптимальне значення α. Таку функцію активації можна назвати параметричною функцією ReLU (Parametric ReLU). Ми також можемо встановити для α деяке випадкове значення, і це називається функцією ReLU із шумом (Randomized ReLU).

Функція ELU

Експоненціальний лінійний вузол (Exponential linear unit - ELU), як і Leaky ReLU, має невеликий нахил для від'ємних значень. Але замість прямої лінії він має логарифмічну криву, як показано на наступній діаграмі:

```
%%html
<img src="images/10.png" style="width:650px;height:400px;">
```



Це можна виразити так:

$$f(x) = \begin{cases} \alpha\left(e^{x} - 1\right) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

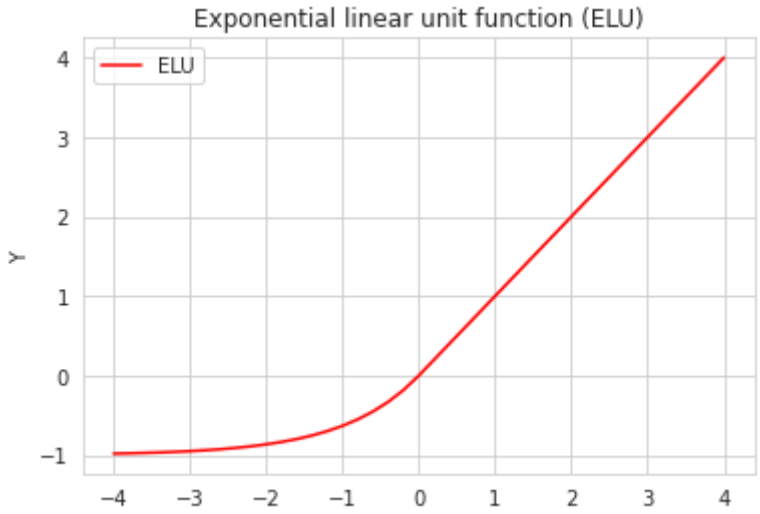
The ELU function is implemented in python as follows:

```
def ELU(x,alpha=0.01):
    if x<0:
        return (alpha*(np.exp(x)-1))

    else:
        return x
```

```
ELU_list = []

#x_list = range(-20,20)
print('x_list=',list(x_list))
alpha = 1.0
for x in x_list:
    y = ELU(x,alpha)
    ELU_list.append(y)
print('leakyReLU_list=',ELU_list)
```

```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999
leakyReLU_list= [-0.9816843611112658, -0.9776292281438344, -0.97267
```

```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(x_list, ELU_list, color='red', label='ELU')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Exponential linear unit function (ELU)')
plt.legend()
plt.show()
```



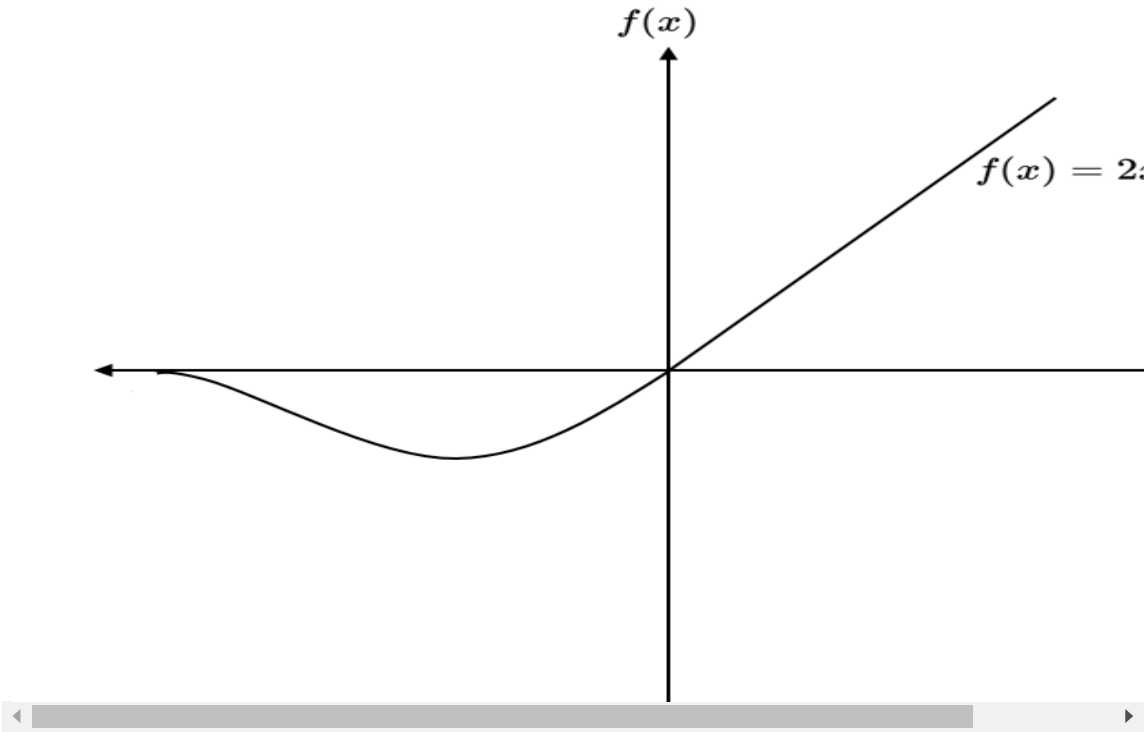
Фнукція Swish

Функція Swish нещодавно представлена компанією Google. На відміну від інших функцій активації, які є монотонними, Swish є немонотонною функцією, що означає, що вона не завжди постійно зростає, або спадає. Вона забезпечує кращу продуктивність, ніж ReLU. Функція Swish має наступний вираз:

$$f(x) = x\sigma(x)$$

Тут, $\sigma(x)$ - це сигмоїдна функція. Функцію Swish показано на наступній схемі:

```
%%html
<img src="images/11.png" style="width:650px;height:400px;">
```



Ми також можемо перепараметризувати функцію Swish і виразити її таким чином:

$$f(x) = 2x\sigma(\beta x)$$

Коли значення β = 0, то ми отримуємо функцію тотожності $f(x) = x$. Вона стає лінійною функцією і, коли значення β tends to infinity, тоді $f(x)$ стає $2max(0, x)$, яка в основному є функцією ReLU, помноженою на деяке постійне значення. Отже, значення β діє як хороша інтерполяція між лінійною та нелінійною функціями. Функцію swish можна реалізувати, як показано нижче:

```
def swish(x,beta):
    return 2*x*sigmoid(beta*x)
```

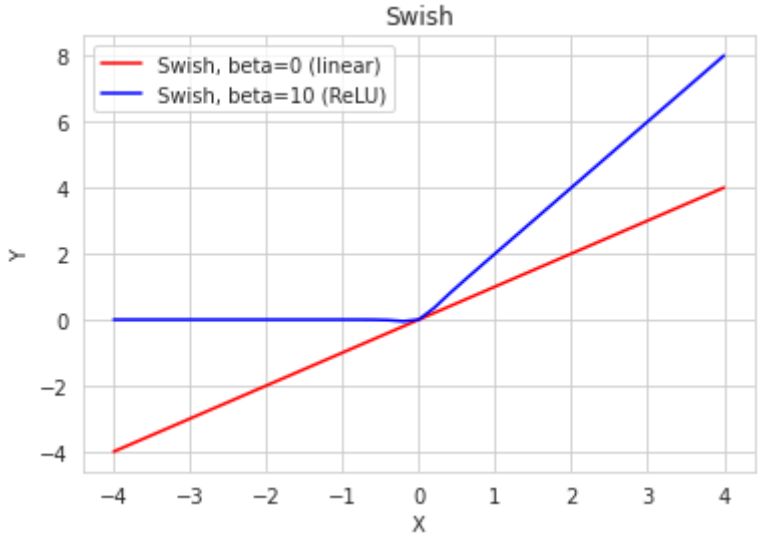
```
swish_list_0 = []

#x_list = range(-20,20)
print('x_list=',list(x_list))
beta = 0.0
for x in x_list:
    y = swish(x,beta)
    swish_list_0.append(y)
print('swish_list=',swish_list_0)
```

```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999996, -2.4,
swish_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999996, -
```

```
swish_list_1 = []

#x_list = range(-20,20)
print('x_list=',list(x_list))
beta = 10.0
for x in x_list:
    y = swish(x,beta)
    swish_list_1.append(y)
print('swish_list=',swish_list_1)
```

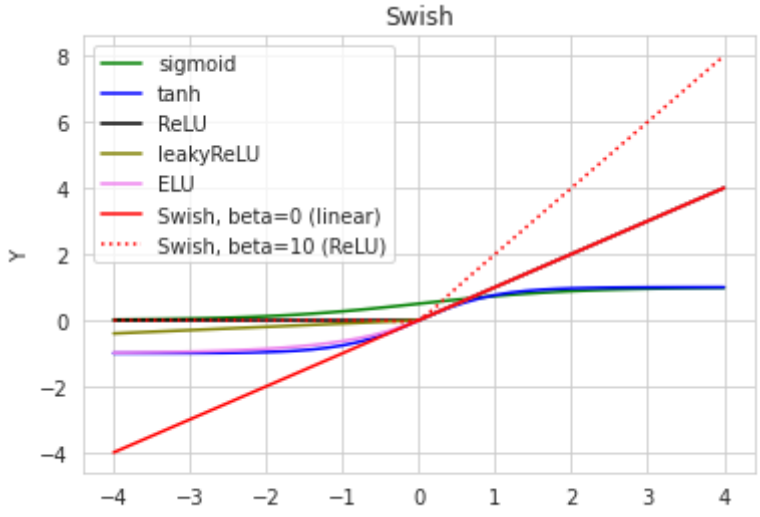
```
x_list= [-4.0, -3.8, -3.6, -3.4, -3.2, -3.0, -2.8, -2.5999999999999996, -2.4,
swish_list= [-3.398683404233271e-17, -2.385740921956502e-16, -1.6700564377753
```

```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(x_list, swish_list_0, color='red', label='Swish, beta=0 (linear)')
plt.plot(x_list, swish_list_1, color='blue', label='Swish, beta=10 (ReLU)')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Swish')
plt.legend()
plt.show()
```



Порівняльний аналіз

```
# Plot ALL:
sns.set_style("whitegrid")
plt.plot(x_list, sigmoid_list, color='green', label='sigmoid')
plt.plot(x_list, tanh_list, color='blue', label='tanh')
plt.plot(x_list, ReLU_list, color='black', label='ReLU')
plt.plot(x_list, leakyReLU_list, color='olive', label='leakyReLU')
plt.plot(x_list, ELU_list, color='violet', label='ELU')
plt.plot(x_list, swish_list_0, color='red', label='Swish, beta=0 (linear)')
plt.plot(x_list, swish_list_1, color='red', linestyle='dotted', label='Swish, beta
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Swish')
plt.legend()
plt.show()
```



Функція softmax

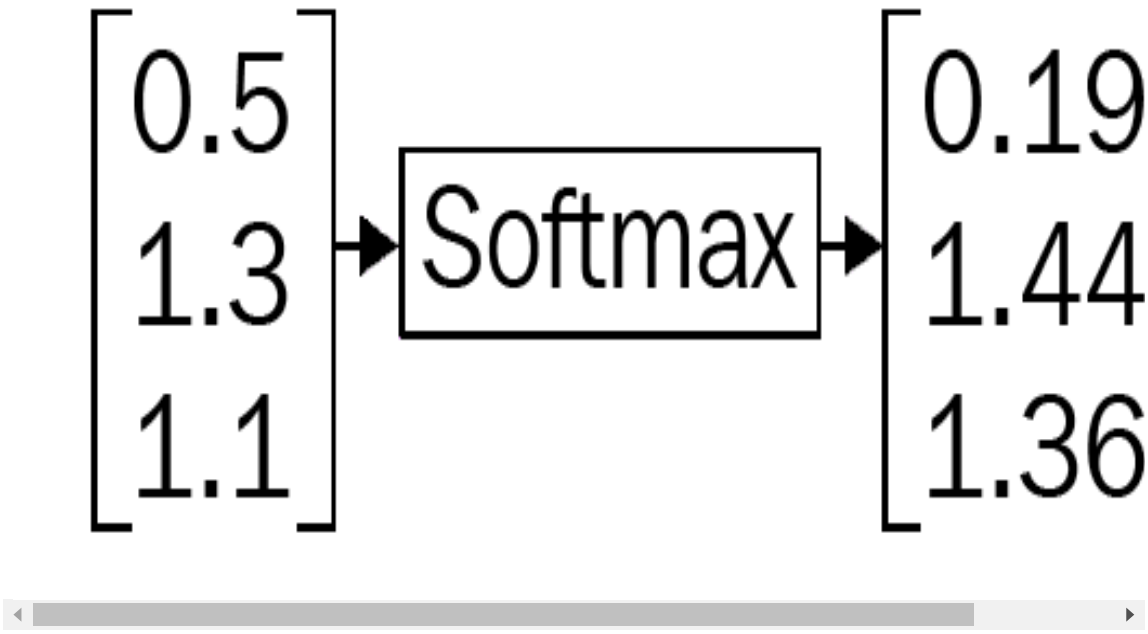
Функція softmax в основному є узагальненням сигмоїдної функції. Зазвичай він застосовується до останнього рівня мережі та під час виконання завдань класифікації кількох класів. Він дає ймовірність виведення кожного класу, і, таким чином, сума значень softmax завжди дорівнюватиме 1.

Її можна представити так:

$$f\left(x_i\right) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Як показано на наступній діаграмі, функція softmax перетворює їхні вхідні дані на ймовірності (будь ласка, зверніть увагу та поясніть ... :) ... помилку на цьому малюнку нижче):

```
%%html
<img src="images/12.png" style="width:650px;height:400px;">
```



Функцію softmax можна реалізувати в Python наступним чином:

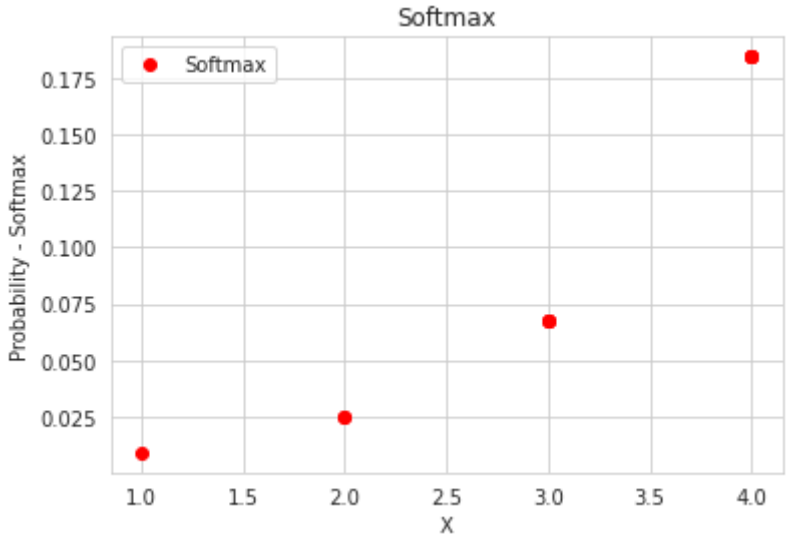
```
def softmax(x):
    return np.exp(x) / np.exp(x).sum(axis=0)
```

```
xs_list = [1,2,2,3,3,3,4,4,4,4]
#xs_list = [0.5,1.3,1.1]
softmax_list = softmax(xs_list)

print('x_list=',xs_list)
```

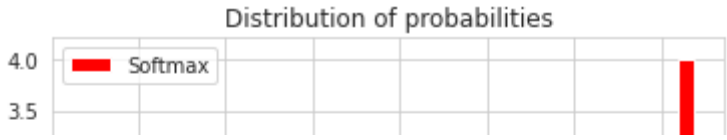
```
print('softmax_list=',softmax_list)
print('sum(softmax_list)=',sum(softmax_list))
print('rounded_sum(softmax_list)=',round(sum(softmax_list),3))
```

```
x_list= [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
softmax_list= [0.00917887 0.02495075 0.02495075 0.06782318 0.06782318 0.06782
 0.18436252 0.18436252 0.18436252 0.18436252]
sum(softmax_list)= 0.9999999999999999
rounded_sum(softmax_list)= 1.0
```

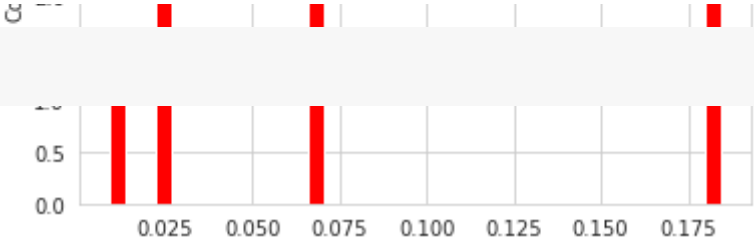
```
# Побудувати графік:
sns.set_style("whitegrid")
plt.plot(xs_list, softmax_list, 'o', color='red', label='Softmax')
plt.xlabel('X')
plt.ylabel('Probability - Softmax')
plt.title('Softmax')
plt.legend()
plt.show()
```




```
# Побудувати графік розподілу ймовірностей
plt.hist(softmax_list, bins=40, color='red', label='Softmax')
plt.ylabel('Probability - Softmax')
plt.ylabel('Counts')
plt.title('Distribution of probabilities')
plt.legend()
plt.show()
```



У наступному блокноті Jupyter ми розповімо, як нейронна мережа використовує пряме розповсюдження для прогнозування результату.



Попередні умови: монтувати Google Drive + скопіювати необхідні файли (бібліотеки + набори даних)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
! pwd
```

```
/content
```

```
! ls /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DEMO
```

```
'DEMO_01_Activation Functions_EXAMPLE.ipynb'
'DEMO_01_Activation Functions.ipynb'
'DEMO_01_Activation Functions_UA.ipynb'
'DEMO_02_Neural Network from scratch_EXAMPLE.ipynb'
'DEMO_02_Neural Network from scratch.ipynb'
'DEMO_02_Neural Network from scratch_UA.ipynb'
 images
```

```
! cp -r /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DE
```

```
! ls
```

```
'DEMO_01_Activation Functions_EXAMPLE.ipynb'
'DEMO_01_Activation Functions.ipynb'
'DEMO_01_Activation Functions_UA.ipynb'
'DEMO_02_Neural Network from scratch_EXAMPLE.ipynb'
'DEMO_02_Neural Network from scratch.ipynb'
'DEMO_02_Neural Network from scratch_UA.ipynb'
 drive
 images
 sample_data
```

```
# Хитрість для використання статичних зображень у Google Colaboratory
path = '/usr/local/share/jupyter/nbextensions/google.colab'
!cp -r {path}/* .
!rm -r {path}
!ln -s /content {path}
# змінити базовий тег
from google.colab.output import eval_js
def change_base_url():
  eval_js("""
  var base = document.createElement('base')
  base.href = 'https://localhost:8080/nbextensions/google.colab/'
  document.head.prepend(base)
  """)
```

```
# зробити так, щоб він запускався автоматично в кожній клітинці
get_ipython().events.register('pre_run_cell', change_base_url)
```

```
DATA_HOME = "/content/drive/MyDrive/2022_COLAB_ML"
from IPython.core.display import HTML
def css_styling():
    styles = open(DATA_HOME + "/styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Нейронна мережа з нуля

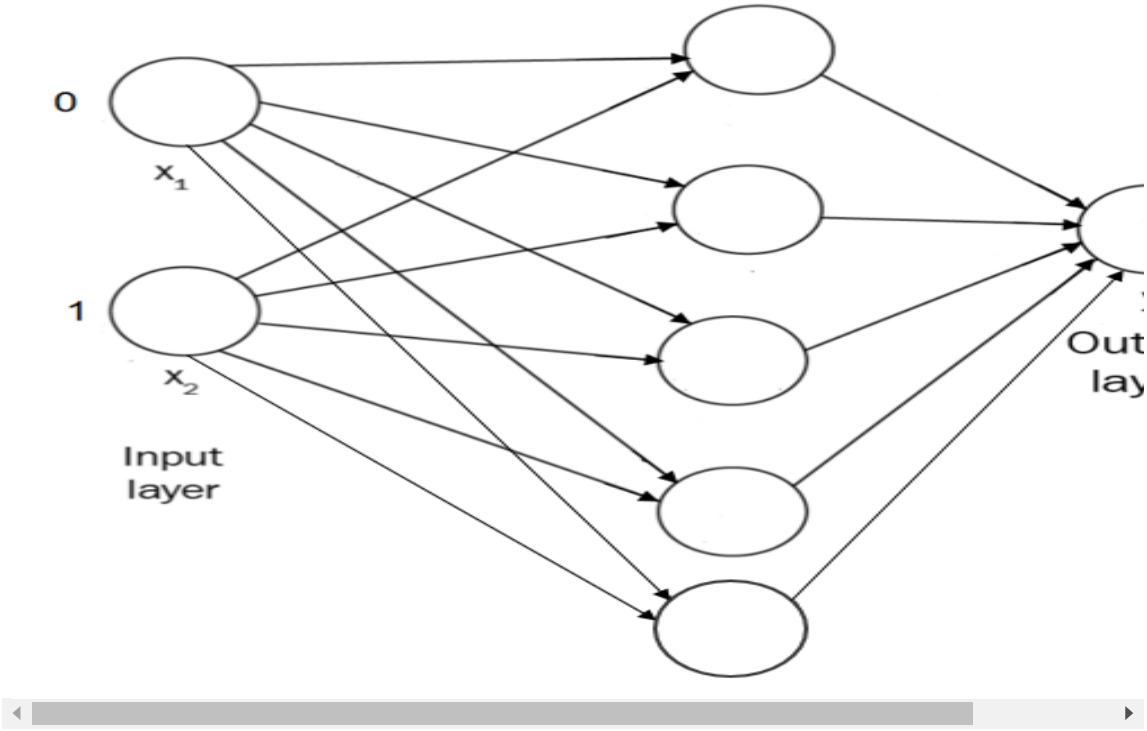
Зібравши всі концепції, які ми вивчили досі, ми побачимо, як побудувати нейронну мережу з нуля. Ми дізнаємося, як нейронна мережа вчиться виконувати операцію XOR. Шлюз XOR повертає 1 лише тоді, коли лише один із його входів дорівнює 1, інакше він повертає 0, як показано на наступному малюнку:

```
%%html
<img src="images/1.png" style="width:650px;height:400px;">
```

Input(x)		Output(y)
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Щоб виконати операцію вентиля XOR, ми будуємо просту двошарову нейронну мережу, як показано на наступному малюнку. Як ви бачите, ми маємо вхідний шар із двома вузлами, прихований шар із п’ятьма вузлами та вихідні шари, які складаються з 1 вузла:

```
%%html
<img src="images/2.png" style="width:650px;height:400px;">
```



Для початку, імпортуйте бібліотеки:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Підготуйте дані, як показано в наведеній вище таблиці XOR:

```
x = np.array([ [0, 1], [1, 0], [1, 1],[0, 0] ])
y = np.array([ [1], [1], [0], [0]])
```

Визначте кількість вузлів у кожному шарі:

```
num_input = 2
num_hidden = 5
num_output = 1
```

Ініціалізація ваги та зміщення випадковим чином. Спочатку ми ініціалізуємо, вводимо ваги прихованого шару:

```
Wxh = np.random.randn(num_input,num_hidden)
bh = np.zeros((1,num_hidden))
```

Тепер ініціалізуйте, приховані ваги вихідного шару:

```
Why = np.random.randn (num_hidden,num_output)
by = np.zeros((1,num_output))
```

Визначте функцію активації сигмоїд:

```
def sigmoid(z):
    return 1 / (1+np.exp(-z))
```

Визначимо похідну функції сигмоїд:

```
def sigmoid_derivative(z):
    return np.exp(-z)/((1+np.exp(-z))**2)
```

Визначте пряме поширення:

```
def forward_prop(x,Wxh,Why):
    z1 = np.dot(x,Wxh) + bh
    a1 = sigmoid(z1)
    z2 = np.dot(a1,Why) + by
    y_hat = sigmoid(z2)

    return z1,a1,z2,y_hat
```

Визначити зворотне поширення:

```
def backword_prop(y_hat, z1, a1, z2):
    delta2 = np.multiply(-(y-y_hat),sigmoid_derivative(z2))
    dJ_dWhy = np.dot(a1.T, delta2)
    delta1 = np.dot(delta2,Why.T)*sigmoid_derivative(z1)
    dJ_dWxh = np.dot(x.T, delta1)

    return dJ_dWxh, dJ_dWhy
```

Визначте функцію витрат:

```
def cost_function(y, y_hat):
    J = 0.5*sum((y-y_hat)**2)

    return J
```

Встановіть швидкість навчання та кількість ітерацій навчання:

```
alpha = 0.01
num_iterations = 5000
```

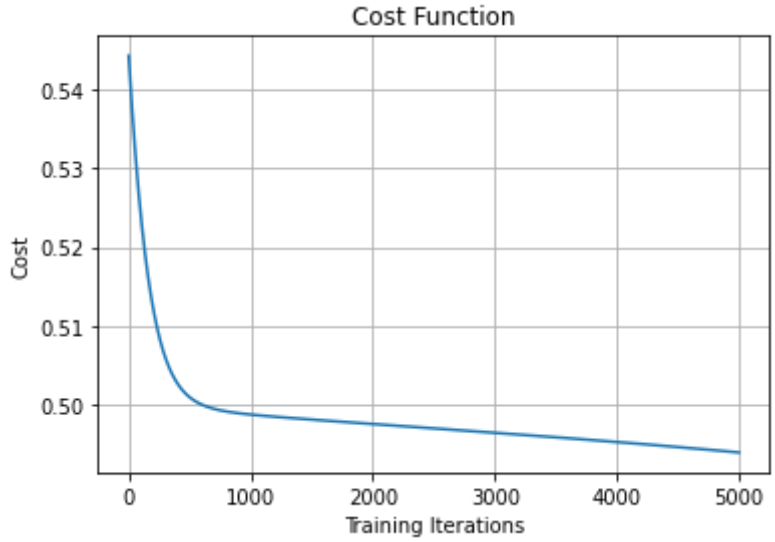
Тепер приступимо до навчання мережі:

```
cost = []
for i in range(num_iterations):

    #perform forward propagation and predict output
    z1,a1,z2,y_hat = forward_prop(x,Wxh,Why)

    #perform backward propagation and calculate gradients
    dJ_dWxh, dJ_dWhy = backword_prop(y_hat, z1, a1, z2)

    #update the weights
    Wxh = Wxh -alpha * dJ_dWxh
    Why = Why -alpha * dJ_dWhy

    #compute cost
    c = cost_function(y, y_hat)

    #store the cost
    cost.append(c)
```

Побудуйте графік функції витрат:

```
plt.grid()
plt.plot(range(num_iterations),cost)

plt.title('Cost Function')
plt.xlabel('Training Iterations')
plt.ylabel('Cost')
```

```
Text(0, 0.5, 'Cost')
```



Як ви можете помітити, втрати зменшуються протягом ітерацій навчання. Отже, ми навчилися створювати нейронну мережу з нуля, а в наступному розділі ми розглянемо одну з популярних бібліотек глибокого навчання під назвою TensorFlow.

Логістична регресія за допомогою нейронної мережі

(C) на основі курсів Coursera (з деякими рішеннями!):

Neural Networks and Deep Learning (Week 2 assignment)

Заборонено копіювати та вставляти ці рішення на ваш трек Coursera! :) :)

Ви створите класифікатор логістичної регресії для розпізнавання котів. Це допоможе вам зрозуміти, як це робити за допомогою мислення нейронної мережі, а також відточити вашу інтуїцію щодо глибокого навчання.

Ви навчитеся:

- Створювати загальну архітектуру алгоритму навчання, включаючи::
  - Параметри ініціалізації
  - Розрахунок функції витрат та її градієнта за допомогою
  - Алгоритм оптимізації (градієнтний спуск)
- Зберіть усі три вищенаведені функції в основну функцію моделі в правильному порядку.

```

```

Попередні умови: монтувати Google Drive + скопіювати необхідні файли (бібліотеки + набори даних)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
! pwd
```

```
/content
```

```
! ls /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DEMO_
```

```
01_LR_BuildNN_EXAMPLE.ipynb  01_LR_BuildNN_UA.ipynb  images
01_LR_BuildNN.ipynb          datasets                lr_utils.py
```

```
! cp -r /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DE
```

```

```

```
! ls
```

```
    01_LR_BuildNN_EXAMPLE.ipynb  01_LR_BuildNN_UA.ipynb  drive   lr_utils.py
    01_LR_BuildNN.ipynb          datasets                images  sample_data
```

```
# Хитрість для використання статичних зображень у Google Colaboratory
path = '/usr/local/share/jupyter/nbextensions/google.colab'
!cp -r {path}/* .
!rm -r {path}
!ln -s /content {path}
# змінити базовий тег
from google.colab.output import eval_js
def change_base_url():
  eval_js("""
  var base = document.createElement('base')
  base.href = 'https://localhost:8080/nbextensions/google.colab/'
  document.head.prepend(base)
  """)
# зробити так, щоб він запускався автоматично в кожній клітинці
get_ipython().events.register('pre_run_cell', change_base_url)
```

```
DATA_HOME = "/content/drive/MyDrive/2022_COLAB_ML"
from IPython.core.display import HTML
def css_styling():
    styles = open(DATA_HOME + "/styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

1 - Пакети

Спочатку давайте запустимо клітинку нижче, щоб імпортувати всі пакети, які вам знадобляться під час цього призначення.

- numpy - це основний пакет для наукових обчислень на Python.
- h5py - це пакет для взаємодії з набором даних, який зберігається у файлі H5.
- matplotlib - це відома бібліотека для побудови графіків на Python.
- PIL і scipy використовуються тут, щоб перевірити вашу модель із вашим власним зображенням у кінці.

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from lr_utils import load_dataset

%matplotlib inline
```

2 - Огляд набору задач

Постановка проблеми: Вам надається набір даних ("data.h5"), який містить: - навчальний набір зображень m_train, позначених як cat (y=1) або не-cat (y=0) - тестовий набір зображень m_test, позначених як cat або noncat - кожне зображення має форму (num_px, num_px, 3), де 3 означає 3 канали (RGB). Таким чином, кожне зображення є квадратним (висота = num_px) і (ширина = num_px).

Ви створите простий алгоритм розпізнавання зображень, який зможе правильно класифікувати зображення як кішки чи не кішки.

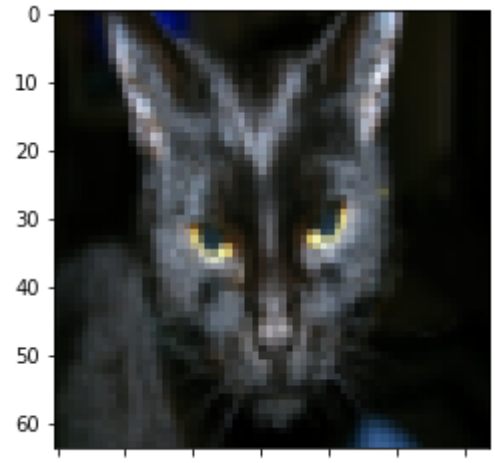
Давайте ближче познайомимося з набором даних. Завантажте дані, запустивши наступний код.

```
# Завантаження даних (кішка/не кішка)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset
```

Ми додали «_orig» у кінці наборів даних зображень (навчання та тестування), тому що збираємося їх попередньо обробити. Після попередньої обробки ми отримаємо train_set_x і test_set_x (мітки train_set_y і test_set_y не потребують попередньої обробки). Кожен рядок ваших train_set_x_orig і test_set_x_orig є масивом, що представляє зображення. Ви можете візуалізувати приклад, запустивши наступний код. Не соромтеся також змінити індекс значення і повторно запустіть, щоб побачити інші зображення.

```
# Приклад зображення
index = 25
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" + classes[np.squeeze(tra
```

```
y = [1], it's a 'cat' picture.
```



Багато програмних помилок у глибокому навчанні походять від наявності матриць/ векторних розмірів, яких немає. Якщо ви можете підтримувати правильні розміри

матриці/вектора, ви значно зробите шлях до усунення багатьох помилок.

Вправа: Знайдіть значення для:

- m_train (кількість навчальних прикладів)
- m_test (кількість тестових прикладів)
- num_px (= height = width навчального зображення)

Пам'ятайте, що `train_set_x_orig` - це numpy-масив форми (m_train, num_px, num_px, 3). Наприклад, ви можете отримати доступ до `m_train`, якщо написати train_set_x_orig.shape[0].

```
### ПОЧАТИ КОД ТУТ ### (≈ 3 рядки коду)
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig[0].shape[0]
### ЗАВЕРШИТИ КОД ТУТ ###

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

Очікуваний результат для m_train, m_test і num_px:

m_train	209
m_test	50
num_px	64

Для зручності тепер вам слід змінити форму зображень форми (num_px, num_px, 3) у масив numpy форми (num_px ∗ num_px ∗ 3, 1). Після цього наш навчальний (і тестовий) набір даних є numpyarray, де кожен стовпець представляє зображення. Повинно бути m_train стовпців (і відповідно m_test стовпців).

Вправа: Змініть форму навчальних і тестових наборів даних так, щоб зображення розміром (num_px, num_px, 3) зводилися до векторів форми (num_px ∗ num_px ∗ 3, 1).

Хитрість в тому, що коли ви хочете переформатувати матрицю X форми (a,b,c,d) на матрицю X_flatten форми (b∗c∗d, a), то треба використати наступне:

```
X_flatten = X.reshape(X.shape[0], -1).T      # X.T - це транспонована версія X
```

```
# Переформатуйте навчальні та тестові приклади

### ПОЧАТИ КОД ТУТ ### (≈ 2 рядки коду)
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T
### ЗАВЕРШИТИ КОД ТУТ ###

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

```
train_set_x_flatten shape: (12288, 209)
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
sanity check after reshaping: [17 31 56 22 33]
```

Очікуваний результат:

train_set_x_flatten shape	(12288, 209)
train_set_y shape	(1, 209)
test_set_x_flatten shape	(12288, 50)
test_set_y shape	(1, 50)
перевірка після зміни форми	[17 31 56 22 33]

Для представлення кольорових зображень червоний, зелений і синій канали (RGB) мають бути визначені для кожного пікселя, тому значення пікселя насправді є вектором із трьох чисел у діапазоні від 0 до 255.

Одним із поширених етапів попередньої обробки в машинному навчанні є центрування та стандартизація вашого набору даних, тобто ви віднімаєте середнє значення всього масиву numpy з кожного прикладу, а потім ділите кожен приклад на стандартне відхилення всього масиву numpy. Але для наборів даних зображень простіше і зручніше, а також працює майже так само добре, просто розділити кожен рядок набору даних на 255 (максимальне значення каналу пікселів).

Давайте стандартизуємо наш набір даних.

```
train_set_x = train_set_x_flatten/255.
```

```
test_set_x = test_set_x_flatten/255.
```

Що потрібно пам'ятати:

Загальні кроки для попередньої обробки нового набору даних:

- Визначте розміри та форми проблеми (m_train, m_test, num_px, ...)
- Змініть набори даних так, щоб кожен приклад тепер був вектором розміру (num_px * num_px * 3, 1)
- «Стандартизуйте» дані

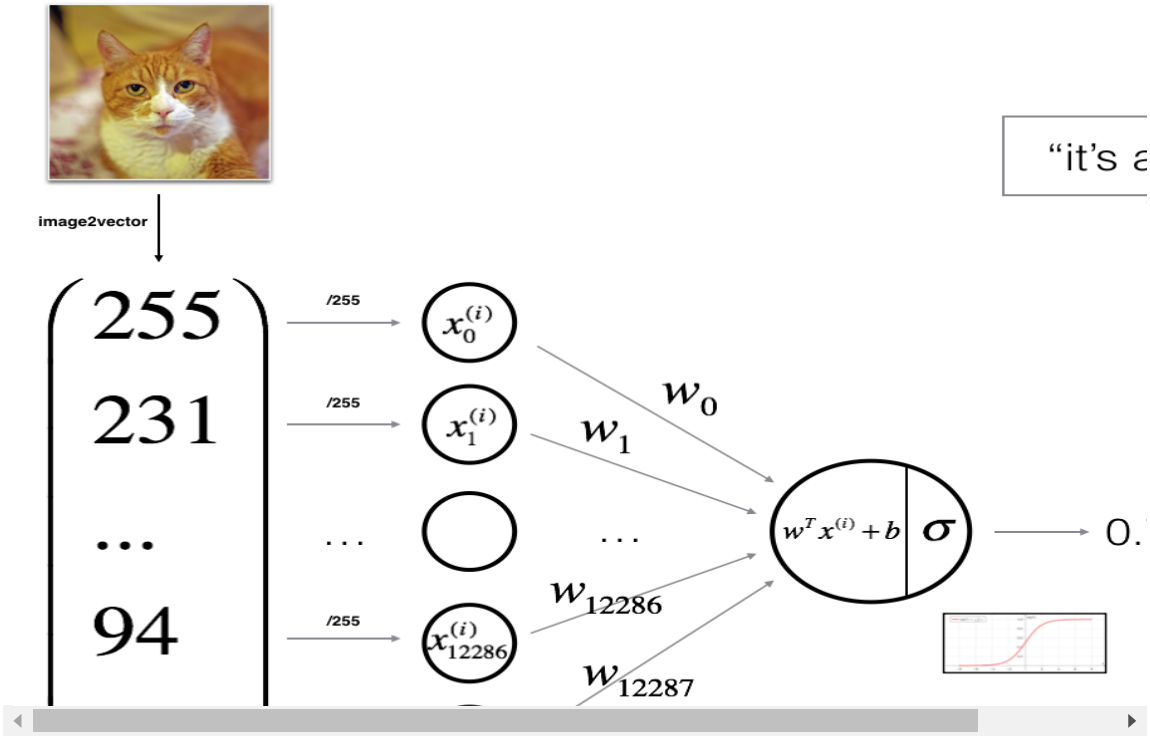
3 - Загальна архітектура алгоритму навчання

Настав час розробити простий алгоритм, щоб відрізняти зображення котів від зображень, які не є котами.

Ви побудуєте логістичну регресію, використовуючи мислення нейронної мережі. Наступний малюнок пояснює чому

Логістична регресія насправді є дуже простою нейронною мережею!

```
%%html
<img src="images/LogReg_kiank.png" style="width:650px;height:400px;">
```



Математичне вираження алгоритму:

Для одного прикладу $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b \tag{1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$
$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \tag{3}$$

Втрата обчислюється шляхом підсумовування всіх навчальних прикладів:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{6}$$

Ключові кроки: У цій вправі ви виконаєте наступні кроки:

- Ініціалізуйте параметри моделі
- Вивчіть параметри для моделі шляхом мінімізації витрат
- Використовуйте вивчені параметри для прогнозування (на тестовому наборі)
- Аналізуйте результати та робіть висновки

4 - Побудова частин нашого алгоритму

Основні етапи створення нейронної мережі:

1. Визначте структуру моделі (наприклад, кількість вхідних функцій)
2. Ініціалізація параметрів моделі
3. Цикл:
    - Розрахувати втрати струму (пряме поширення)
    - Розрахувати градієнт струму (зворотне поширення)
    - Оновити параметри (градієнтний спад)

Ви часто створюєте 1-3 окремо та об'єднуєте їх в одну функцію, яку ми називаємо `model()`.

4.1 - Допоміжні функції

Вправа: Використовуючи код з попередніх розділів, реалізуйте `sigmoid()`. Як видно із малюнка зверху, потрібно обчислити $sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$ для обчислення прогнозів. Використайте np.exp().

```
# ЗАВДАННЯ: створити функцію sigmoid

def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """
```

```
    ### ПОЧАТИ КОД ТУТ ### (≈ 1 рядок коду)
    s = 1 / (1 + np.exp(-z))
    ### ЗАВЕРШИТИ КОД ТУТ ###

    return s
```

```
print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
```

```
 sigmoid([0, 2]) = [0.5        0.88079708]
```

Очікуваний результат:

sigmoid([0, 2]) [0.5 0.88079708]

4.2 - Параметри ініціалізації

Вправа: Реалізуйте ініціалізацію параметра в комірці нижче. Ви повинні ініціалізувати w як вектор нулів. Якщо ви не знаєте, яку функцію numpy використовувати, знайдіть np.zeros() у документації бібліотеки Numpy.

```
# ЗАВДАННЯ: створити функцію initialize_with_zeros

def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w and initializes

    Argument:
    dim -- size of the w vector we want (or number of parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """

    ### ПОЧАТИ КОД ТУТ ### (≈ 1 рядок коду)
    w = np.zeros([dim, 1])
    b = 0
    ### ЗАВЕРШИТИ КОД ТУТ ###

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

```
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

```
w = [[0.]
 [0.]]
```

Очікуваний результат:

```
** w **   [[ 0.] [ 0.]]
** b **   0
```

Для вхідних зображень w матиме форму (num_px $\times$ num_px $\times$ 3, 1).

4.3 - Пряме і зворотне розповсюдження

Тепер, коли ваші параметри ініціалізовано, ви можете виконати кроки "вперед" і "назад" для вивчення параметрів.

Вправа: Реалізуйте функцію `propagate()` який обчислює функцію втрат та її градієнт.

Підказка:

Пряме поширення:

- Отримайте X
- Обчисліть $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, \ldots, a^{(m-1)}, a^{(m)})$
- Обчисліть функцію втрат: $J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Ось дві формули, які ви будете використовувати:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T \tag{7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{8}$$

```
# ЗАВДАННЯ: створити функцію propagate

def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained abo

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, numbe

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b

    Tips:
    - Write your code step by step for the propagation. np.log(), np.dot()
    """
```

```
    m = X.shape[1]

    # ПРЯМЕ ПОШИРЕННЯ (ВІД X ДО ВТРАТ)
    ### ПОЧАТИ КОД ТУТ ### (≈ 2 рядки коду)
    A = sigmoid(np.dot(w.T, X) + b)                                    # обчислити фу
    cost = -np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A)) / m     # обчислити вт
    ### ЗАВЕРШИТИ КОД ТУТ ###

    # ЗВОРОТНЕ ПОШИРЕННЯ (ЗНАЙТИ ГРАДІЄНТ)
    ### ПОЧАТИ КОД ТУТ ### (≈ 2 рядки коду)
    dw = np.dot(X, (A - Y).T) / m
    db = np.sum(A - Y) / m
    ### ЗАВЕРШИТИ КОД ТУТ ###

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

```
w, b, X, Y = np.array([[1],[2]]), 2, np.array([[1,2],[3,4]]), np.array([[1,0]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

```
dw = [[0.99993216]
 [1.99980262]]
db = 0.49993523062470574
cost = 6.000064773192205
```

Очікуваний результат:

** dw **	[[0.99993216] [1.99980262]]
** db **	0.499935230625
** cost **	6.000064773192205

d) Оптимізація

- ініціалізувати параметри
- обчислити функцію втрат та її градієнт
- оновити параметри за допомогою градієнтного спуску.

Вправа: Створити функцію оптимізації. Мета навчити w і b шляхом мінімізації функції втрат J. Для параметру θ, правило оновлення $\theta = \theta - \alpha\ d\theta$, де α - це темп навчання.

```
# ЗАВДАННЯ: створити функцію optimize

def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, numb
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with respe
    costs -- list of all the costs computed during the optimization, this will be u

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters. Use prop
        2) Update the parameters using gradient descent rule for w and b.
    """

    costs = []

    for i in range(num_iterations):


        # Cost and gradient calculation (≈ 1-4 lines of code)
        ### START CODE HERE ###
        grads, cost = propagate(w, b, X, Y)
        ### END CODE HERE ###

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule (≈ 2 lines of code)
        ### START CODE HERE ###
        w = w - learning_rate * dw
        b = b - learning_rate * db
        ### END CODE HERE ###

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

        # Print the cost every 100 training examples
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
```

```
                "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

```
params, grads, costs = optimize(w, b, X, Y, num_iterations= 100, learning_rate = 0

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

```
w = [[0.1124579 ]
 [0.23106775]]
b = 1.5593049248448891
dw = [[0.90158428]
 [1.76250842]]
db = 0.4304620716786828
```

Очікуваний результат:

w	[[0.1124579] [0.23106775]]
b	1.55930492484
dw	[[0.90158428] [1.76250842]]
db	0.430462071679

Вправа: Попередня функція виведе вивчені w і b. Ми можемо використовувати w і b для прогнозування міток для набору даних X. Реалізуйте функцію `predict()`. Є два кроки обчислення передбачення:

1. Обчисліть $\hat{Y} = A = \sigma(w^T X + b)$
2. Перетворює записи a на 0 (якщо активація <= 0,5) або 1 (якщо активація > 0,5), зберіжіть передбачення у векторі `Y_prediction`. Якщо ви бажаєте, ви можете використовувати `if`/`else` в циклі `for` (хоча є також спосіб векторизувати це).

```
# ЗАВДАННЯ: створити функцію predict

def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression paramete

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for th
```

```
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    # Compute vector "A" predicting the probabilities of a cat being present in the
    ### START CODE HERE ### (≈ 1 line of code)
    A = sigmoid(np.dot(w.T, X) + b)
    ### END CODE HERE ###

    for i in range(A.shape[1]):

        # Convert probabilities A[0,i] to actual predictions p[0,i]
        ### START CODE HERE ### (≈ 4 lines of code)
        Y_prediction[0][i] = 1 if A[0][i] > 0.5 else 0
        ### END CODE HERE ###

    assert(Y_prediction.shape == (1, m))

    return Y_prediction
```

```
print ("predictions = " + str(predict(w, b, X)))
```

```
     predictions = [[1. 1.]]
```

Очікуваний результат:

predictions [[1. 1.]]

Що слід пам’ятати: Ви реалізували кілька функцій: - Ініціалізація (w,b) - Ітеративна оптимізація втрат для вивчення параметрів (w,b): - обчислення вартості та її градієнта - оновлення параметрів за допомогою градієнта спуск - Використання навчених (w,b), щоб передбачити мітки для заданого набору прикладів.

5 - Об'єднайте всі функції в модель

Тепер ви побачите, як загальна модель структурована шляхом об’єднання всіх будівельних блоків (функцій, реалізованих у попередніх частинах) разом у правильному порядку.

Вправа: Реалізувати функцію моделі. Використовуйте таку нотацію:

- Y_prediction для ваших прогнозів на тестовому наборі
- Y_prediction_train для ваших прогнозів на тренувальному наборі
- w, втрати, градієнти для результатів optimize()

```
# ЗАВДАННЯ: створити функцію model

def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate =
```

```
    """
    Builds the logistic regression model by calling the function you've implemente

    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px * num_px
    Y_train -- training labels represented by a numpy array (vector) of shape (1, m
    X_test -- test set represented by a numpy array of shape (num_px * num_px * 3,
    Y_test -- test labels represented by a numpy array (vector) of shape (1, m_test
    num_iterations -- hyperparameter representing the number of iterations to optim
    learning_rate -- hyperparameter representing the learning rate used in the upda
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """

    ### START CODE HERE ###

    # initialize parameters with zeros (≈ 1 line of code)
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent (≈ 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, lea

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples (≈ 2 lines of code)
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    ### END CODE HERE ###

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_t


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

Запустіть наступну клітинку, щоб навчити свою модель.

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000,
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

Очікуваний результат:

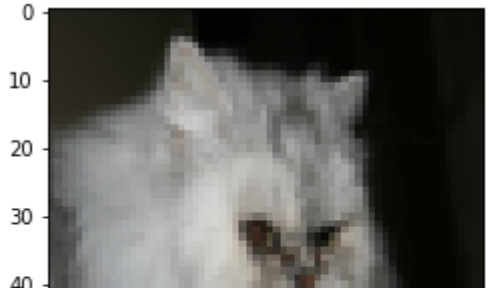
Train Accuracy	99.04306220095694 %
Test Accuracy	70.0 %

Коментар: Тренувальна точність наближається до 100%. Це хороша перевірка працездатності: ваша модель працює та має достатньо високу здатність для тренувальних даних. Тестувальна точність становить 70%. Насправді це непогано для цієї простої моделі, враховуючи невеликий набір даних, який ми використовували, і те, що логістична регресія є лінійним класифікатором. Але не хвилюйтеся, наступного тижня ви створите ще кращий класифікатор!

Крім того, ви бачите, що модель явно перенавчається на треунальному наборі дані. Пізніше ви дізнаєтеся, як зменшити перенавчання, наприклад, за допомогою регулярізації. Використовуючи наведений нижче код (і змінюючи змінну `index`) ви можете переглянути прогнози на малюнках тестового набору.

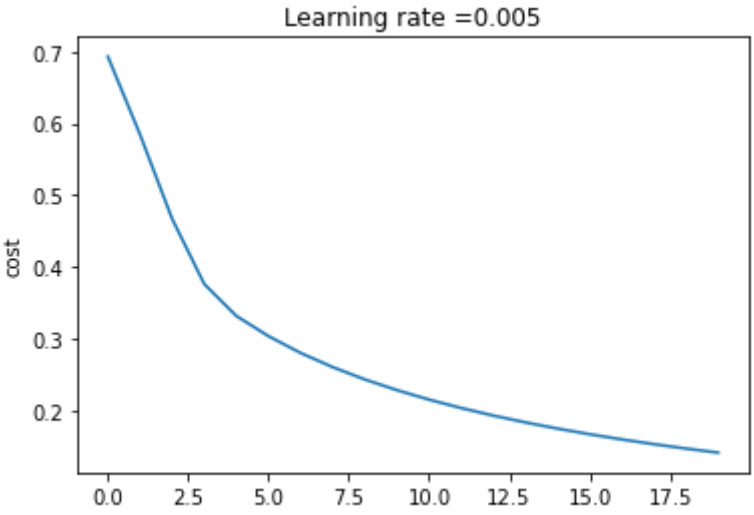
```
# Приклад зображення, яке було неправильно класифіковано.
index = 10
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is a '" + class
```

```
y = 1, you predicted that it is a 'non-cat' picture.
```



Давайте також побудуємо графік функції витрат.

```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```



Інтерпретація: Ви бачите, як втрати зменшуються. Це показує, що параметри навчаються. Однак ви бачите, що ви могли б ще більше тренувати модель на тренувальному наборі. Спробуйте збільшити кількість ітерацій у клітинці вище та повторно запустити код у клітинці. Ви можете побачити, що точність навчального набору підвищується, але точність тестового набору знижується. Це називається перенавчанням (overfitting).

6 - Подальший аналіз

Вітаємо зі створенням вашої першої класичної моделі зображення. Давайте проаналізуємо це далі та розглянемо можливі варіанти швидкості навчання α.

Вибір темпу навчання

Нагадування: Щоб градієнтний спуск працював, потрібно розумно вибрати швидкість навчання α. Швидкість навчання α визначає, як швидко ми оновлюємо параметри. Якщо швидкість навчання надто велика, ми можемо «проскочити» оптимальне значення. Подібним чином, якщо α занадто малий, нам знадобиться занадто багато ітерацій, щоб досягти найкращих значень. Ось чому вкрай важливо використовувати добре налаштований темп навчання.

Давайте порівняємо криву навчання нашої моделі з кількома варіантами швидкості навчання. Запустіть клітинку нижче. Це має зайняти приблизно 1 хвилину. Не соромтеся також спробувати інші значення, ніж три, які ми ініціалізували.

```
learning_rates = [0.01, 0.001, 0.0001]
models = {}
for i in learning_rates:
    print ("learning rate is: " + str(i))
    models[str(i)] = model(train_set_x, train_set_y, test_set_x, test_set_y, num_i
    print ('\n' + "-------------------------------------------------------" + '\n'

for i in learning_rates:
    plt.plot(np.squeeze(models[str(i)]["costs"]), label= str(models[str(i)]["learn

plt.ylabel('cost')
plt.xlabel('iterations')

legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()
```

```
learning rate is: 0.01
train accuracy: 99.52153110047847 %
test accuracy: 68.0 %

-------------------------------------------------------

learning rate is: 0.001
train accuracy: 88.99521531100478 %
test accuracy: 64.0 %

-------------------------------------------------------
```

Інтерпретація:

- Різні темпи навчання дають різні втрати і, отже, різні результати прогнозів. - Якщо швидкість навчання занадто висока (0,01), вартість може коливатися вгору та вниз. Він може навіть відрізнятися (хоча в цьому прикладі використання 0,01 все одно в кінцевому підсумку дає хороше значення втрат).
- Нижчі втрати не означають кращу модель. Ви повинні перевірити наявність перенавчання. Це трапляється, коли точність на тренувальному наборі значно перевищує точність на тестувальному наборі.
- У глибокому навчанні ми зазвичай рекомендуємо вам:
    - Виберіть швидкість навчання, яка краще мінімізує функцію витрат.
    - Якщо ваша модель перенавчається, використовуйте інші техніки, щоб зменшити перенавчання. (Ми поговоримо про це далі.)

7 - Тест на власних зображеннях

Ви можете використати власне зображення та переглянути результат своєї моделі.

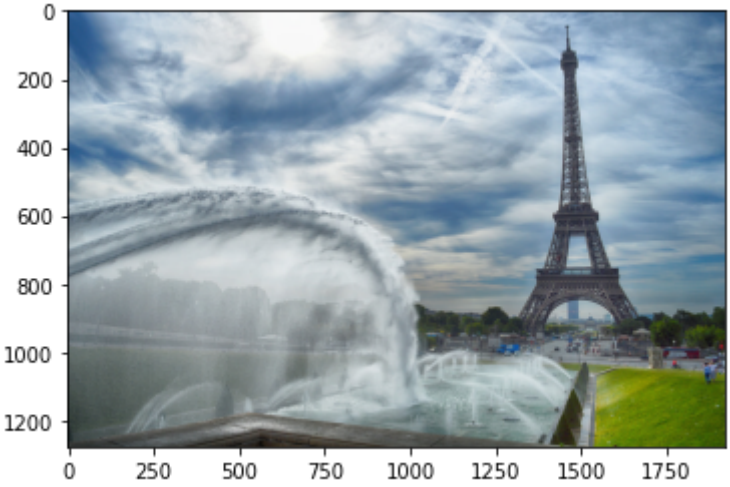
Щоб зробити це:

1. Додайте своє зображення до каталогу цього блокнота Jupyter у папку «images».
2. Змініть назву свого зображення в наступному коді
3. Запустіть код і перевірте, чи правильний алгоритм (1 = кіт, 0 = не кіт)!

```
## START CODE HERE ## (PUT YOUR IMAGE NAME)
my_image = "my_image.jpg"   # change this to the name of your image file
## END CODE HERE ##

import imageio
from skimage.transform import resize

# We preprocess the image to fit your algorithm.
fname = "images/" + my_image
image = np.array(imageio.imread(fname))
my_image = resize(image, (num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)
```

```
plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a "
```

```
y = 0.0, your algorithm predicts a "non-cat" picture.
```



Що слід запам'ятати з цього завдання:

1. Попередня обробка набору даних є важливою.
2. Ви реалізували кожну функцію окремо: initialize(), propagate(), optimize(). Потім ви створили model().
3. Налаштування швидкості навчання (що є прикладом «гіперпараметра») може значно змінити алгоритм. Ви побачите більше прикладів цього пізніше в цьому курсі!

Нарешті, якщо ви хочете, можете спробувати різні речі в цьому блокноті:

- Пограйтеся зі швидкістю навчання та кількістю ітерацій
- Спробуйте різні методи ініціалізації та порівняйте результати
- Перевірте інші види попередньої обробки (відцентруйте дані або розділіть кожен рядок на стандартне відхилення).

```

```

Colab paid products - Cancel contracts here

✓ 0s completed at 9:01 PM

Planar data classification with one hidden layer

based on (C) Coursera Course (with some solutions!):

Neural Networks and Deep Learning (Week 3 assignment)

It is forbidden to copy-paste these solutions to your Coursera track! :)

It's time to build your first neural network, which will have a hidden layer. You will see a big difference between this model and the one you implemented using logistic regression.

You will learn how to:

- Implement a 2-class classification neural network with a single hidden layer
- Use units with a non-linear activation function, such as tanh
- Compute the cross entropy loss
- Implement forward and backward propagation

0 - Pre-requisites: Mount Google Drive + copy necessary files (libraries + datasets)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
! pwd
```

```
/content
```

```
! ls /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DEMO_
```

```
02_NN1_FULL.ipynb                    images
02_NN_with_1-N_layers_EXAMPLE.ipynb  planar_utils.py
02_NN_with_1-N_layers.ipynb          testCases.py
```

```
! cp -r /content/drive/MyDrive/2022_COLAB_NN/Lecture_02_from_Scratch/Lecture_02_DE
```

```
! ls
```

```
01_LR                                   drive                tabbar.css
```

```
# Trick for usage of static images at Google Colaboratory
path = '/usr/local/share/jupyter/nbextensions/google.colab'
!cp -r {path}/* .
!rm -r {path}
!ln -s /content {path}
# change base tag
from google.colab.output import eval_js
def change_base_url():
  eval_js("""
  var base = document.createElement('base')
  base.href = 'https://localhost:8080/nbextensions/google.colab/'
  document.head.prepend(base)
  """)
# make it run automatically in every cell
get_ipython().events.register('pre_run_cell', change_base_url)
```

```
cp: '/usr/local/share/jupyter/nbextensions/google.colab/01_LR' and './01_LR'
cp: '/usr/local/share/jupyter/nbextensions/google.colab/02_NN1' and './02_NN1
cp: '/usr/local/share/jupyter/nbextensions/google.colab/02_NN1_FULL.ipynb' ar
cp: '/usr/local/share/jupyter/nbextensions/google.colab/02_NN_with_1-N_layers
cp: '/usr/local/share/jupyter/nbextensions/google.colab/02_NN_with_1-N_layers
cp: '/usr/local/share/jupyter/nbextensions/google.colab/drive' and './drive'
cp: '/usr/local/share/jupyter/nbextensions/google.colab/files.js' and './file
cp: '/usr/local/share/jupyter/nbextensions/google.colab/images' and './images
cp: '/usr/local/share/jupyter/nbextensions/google.colab/planar_utils.py' and
cp: '/usr/local/share/jupyter/nbextensions/google.colab/sample_data' and './s
cp: '/usr/local/share/jupyter/nbextensions/google.colab/tabbar.css' and './ta
cp: '/usr/local/share/jupyter/nbextensions/google.colab/tabbar_main.min.js' a
cp: '/usr/local/share/jupyter/nbextensions/google.colab/testCases.py' and './
```

```
DATA_HOME = "/content/drive/MyDrive/2022_COLAB_ML"
from IPython.core.display import HTML
def css_styling():
    styles = open(DATA_HOME + "/styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

1 - Packages

Let's first import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- sklearn provides simple and efficient tools for data mining and data analysis.
- matplotlib is a library for plotting graphs in Python.
- testCases provides some test examples to assess the correctness of your functions
- planar_utils provide various useful functions used in this assignment

```
# Package imports
```

```
import numpy as np
import matplotlib.pyplot as plt
from testCases import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset, loa

%matplotlib inline

np.random.seed(1) # set a seed so that the results are consistent
```

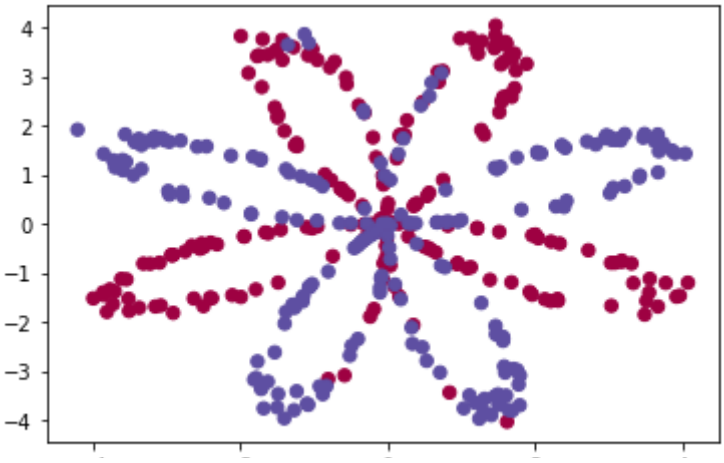
2 - Dataset

First, let's get the dataset you will work on. The following code will load a "flower" 2-class dataset into variables `X` and `Y`.

```
X, Y = load_planar_dataset()
```

Visualize the dataset using matplotlib. The data looks like a "flower" with some red (label y=0) and some blue (y=1) points. Your goal is to build a model to fit this data.

```
# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



You have: - a numpy-array (matrix) X that contains your features (x1, x2) - a numpy-array (vector) Y that contains your labels (red:0, blue:1).

Lets first get a better sense of what our data is like.

Exercise: How many training examples do you have? In addition, what is the `shape` of the variables `X` and `Y`?

Hint: How do you get the shape of a numpy array? (help)

```
### START CODE HERE ### (≈ 3 lines of code)
shape_X = X.shape
shape_Y = Y.shape
m = shape_X[1]  # training set size
### END CODE HERE ###

print ('The shape of X is: ' + str(shape_X))
print ('The shape of Y is: ' + str(shape_Y))
print ('I have m = %d training examples!' % (m))
```

```
The shape of X is: (2, 400)
The shape of Y is: (1, 400)
I have m = 400 training examples!
```

Expected Output:

shape of X	(2, 400)
shape of Y	(1, 400)
m	400

3 - Simple Logistic Regression

Before building a full neural network, lets first see how logistic regression performs on this problem. You can use sklearn's built-in functions to do that. Run the code below to train a logistic regression classifier on the dataset.

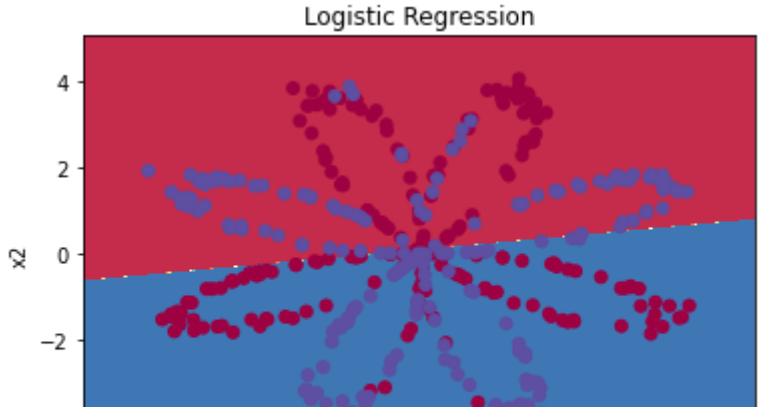
```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:
  y = column_or_1d(y, warn=True)
```

You can now plot the decision boundary of these models. Run the code below.

```
# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")

# Print accuracy
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 47 % (percentage of correctly labe
```



Expected Output:

Accuracy	47%

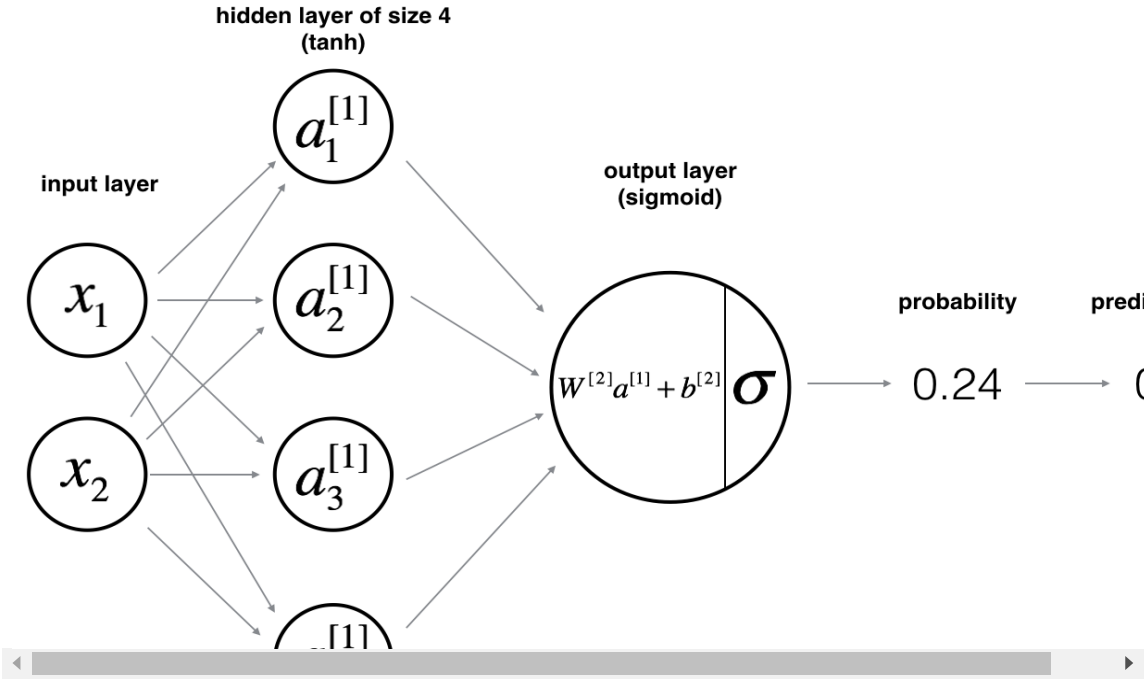
Interpretation: The dataset is not linearly separable, so logistic regression doesn't perform well. Hopefully a neural network will do better. Let's try this now!

4 - Neural Network model

Logistic regression did not work well on the "flower dataset". You are going to train a Neural Network with a single hidden layer.

Here is our model:

```
%%html
<img src="images/classification_kiank.png" style="width:600px;height:370px;">
```



Mathematically:

For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1](i)} \tag{1}$$
$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{2}$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2](i)} \tag{3}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{4}$$
$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right) \tag{6}$$

Reminder: The general methodology to build a Neural Network is to: 1. Define the neural network structure (# of input units, # of hidden units, etc). 2. Initialize the model's parameters 3. Loop: - Implement forward propagation - Compute loss - Implement backward propagation to get the gradients - Update parameters (gradient descent)

You often build helper functions to compute steps 1-3 and then merge them into one function we call `nn_model()`. Once you've built `nn_model()` and learnt the right parameters, you can make predictions on new data.

4.1 - Defining the neural network structure

Exercise: Define three variables: - n_x: the size of the input layer - n_h: the size of the hidden layer (set this to 4) - n_y: the size of the output layer

Hint: Use shapes of X and Y to find n_x and n_y. Also, hard code the hidden layer size to be 4.

```
# GRADED FUNCTION: layer_sizes

def layer_sizes(X, Y):
    """
    Arguments:
    X -- input dataset of shape (input size, number of examples)
    Y -- labels of shape (output size, number of examples)

    Returns:
    n_x -- the size of the input layer
    n_h -- the size of the hidden layer
    n_y -- the size of the output layer
    """
    ### START CODE HERE ### (≈ 3 lines of code)
    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0] # size of output layer
    ### END CODE HERE ###
    return (n_x, n_h, n_y)
```

```
X_assess, Y_assess = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(X_assess, Y_assess)
print("The size of the input layer is: n_x = " + str(n_x))
print("The size of the hidden layer is: n_h = " + str(n_h))
print("The size of the output layer is: n_y = " + str(n_y))
```

```
The size of the input layer is: n_x = 5
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = 2
```

Expected Output (these are not the sizes you will use for your network, they are just used to assess the function you've just coded).

```
The size of the input layer is: n_x = 5
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = 2
```

4.2 - Initialize the model's parameters

Exercise: Implement the function `initialize_parameters()`.

Instructions:

- Make sure your parameters' sizes are right. Refer to the neural network figure above if needed.
- You will initialize the weights matrices with random values.
    - Use: `np.random.randn(a,b) * 0.01` to randomly initialize a matrix of shape (a,b).
- You will initialize the bias vectors as zeros.
    - Use: `np.zeros((a,b))` to initialize a matrix of shape (a,b) with zeros.

```
# GRADED FUNCTION: initialize_parameters

def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    params -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(2) # we set up a seed so that your output matches ours although
```

```
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))
    ### END CODE HERE ###

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
n_x, n_h, n_y = initialize_parameters_test_case()

parameters = initialize_parameters(n_x, n_h, n_y)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00416758 -0.00056267]
 [-0.02136196  0.01640271]
 [-0.01793436 -0.00841747]
 [ 0.00502881 -0.01245288]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.01057952 -0.00909008  0.00551454  0.02292208]]
b2 = [[0.]]
```

Expected Output:

W1 =	[[-0.00416758 -0.00056267] [-0.02136196 0.01640271] [-0.01793436 -0.00841747] [0.00502881 -0.01245288]]
b1 =	[[0.] [0.] [0.] [0.]]
W2 =	[[-0.01057952 -0.00909008 0.00551454 0.02292208]]
b2 =	[[0.]]

4.3 - The Loop

Question: Implement `forward_propagation()`.

Instructions:

- Look above at the mathematical representation of your classifier.
- You can use the function `sigmoid()`. It is built-in (imported) in the notebook.
- You can use the function `np.tanh()`. It is part of the numpy library.
- The steps you have to implement are:
    1. Retrieve each parameter from the dictionary "parameters" (which is the output of `initialize_parameters()`) by using `parameters[".."]`.
    2. Implement Forward Propagation. Compute $Z^{[1]}$, $A^{[1]}$, $Z^{[2]}$ and $A^{[2]}$ (the vector of all your predictions on all the examples in the training set).
- Values needed in the backpropagation are stored in "`cache`". The `cache` will be given as an input to the backpropagation function.

```
# GRADED FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing your parameters (output of initiali

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    ### END CODE HERE ###

    # Implement Forward Propagation to calculate A2 (probabilities)
    ### START CODE HERE ### (≈ 4 lines of code)
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    ### END CODE HERE ###

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
```

```
X_assess, parameters = forward_propagation_test_case()
```

```
A2, cache = forward_propagation(X_assess, parameters)

# Note: we use the mean here just to make sure that your output matches ours.
print(np.mean(cache['Z1']) ,np.mean(cache['A1']),np.mean(cache['Z2']),np.mean(cach
```

```
-0.0004997557777419913 -0.00049696335323177 9 0.0004381874509591465
```

Expected Output:

-0.000499755777742 -0.000496963353232 0.000438187450959 0.500109546852

Now that you have computed $A^{[2]}$ (in the Python variable "`A2`"), which contains $a^{[2](i)}$ for every example, you can compute the cost function as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right) \tag{13}$$

Exercise: Implement `compute_cost()` to compute the value of the cost J.

Instructions:

- There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented $-\sum_{i=0}^{m} y^{(i)}\log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)                # no need to use a for loop!
```

(you can use either `np.multiply()` and then `np.sum()` or directly `np.dot()`).

```
# GRADED FUNCTION: compute_cost

def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 -- The sigmoid output of the second activation, of shape (1, number of exam
    Y -- "true" labels vector of shape (1, number of examples)
    parameters -- python dictionary containing your parameters W1, b1, W2 and b2

    Returns:
    cost -- cross-entropy cost given equation (13)
    """

    m = Y.shape[1] # number of example

    # Compute the cross-entropy cost
    ### START CODE HERE ### (≈ 2 lines of code)
    logprobs = np.dot(Y, np.log(A2).T) + np.dot(1 - Y, np.log(1 - A2).T)
```

```
    cost = np.float64(-logprobs / m)
    ### END CODE HERE ###

    cost = np.squeeze(cost)     # makes sure cost is the dimension we expect.
                                # E.g., turns [[17]] into 17
    assert(isinstance(cost, float))

    return cost
```

```
A2, Y_assess, parameters = compute_cost_test_case()

print("cost = " + str(compute_cost(A2, Y_assess, parameters)))
```

```
     cost = 0.6929198937761264
```

Expected Output:

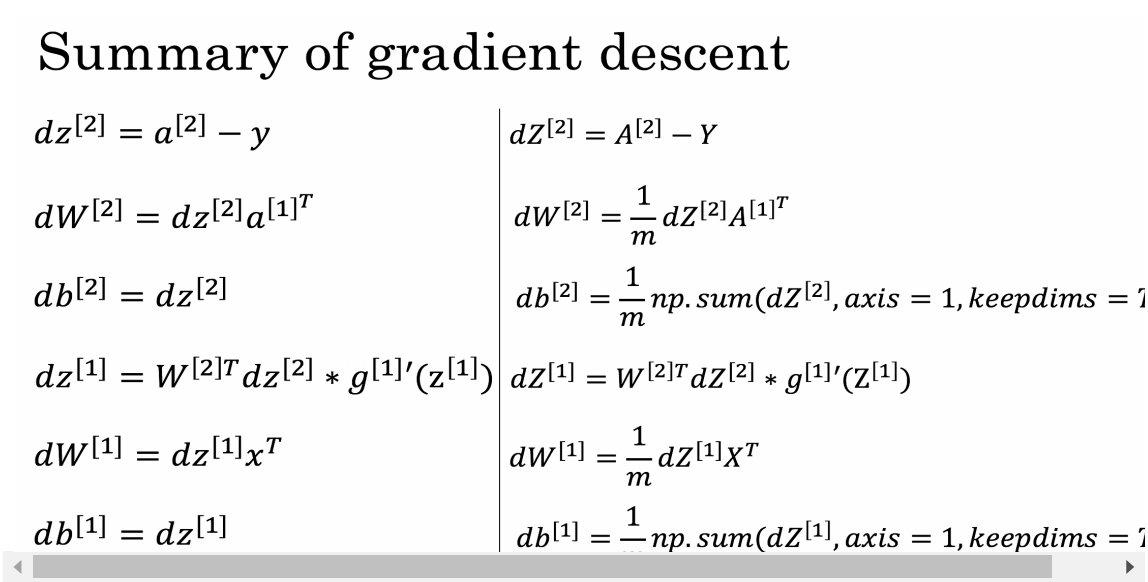
```
cost =  0.692919893776
```

Using the cache computed during forward propagation, you can now implement backward propagation.

Question: Implement the function `backward_propagation()`.

Instructions: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.

```
%%html
<img src="images/grad_summary.png" style="width:600px;height:300px;">
```



- Tips:

- To compute dZ1 you'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}(.)$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1 - a^2$. So you can compute $g^{[1]'}(Z^{[1]})$ using `(1 - np.power(A1, 2))`.

```
# GRADED FUNCTION: backward_propagation

def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

    Returns:
    grads -- python dictionary containing your gradients with respect to different
    """
    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    ### START CODE HERE ### (≈ 2 lines of code)
    W1 = parameters['W1']
    W2 = parameters['W2']
    ### END CODE HERE ###

    # Retrieve also A1 and A2 from dictionary "cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    A1 = cache['A1']
    A2 = cache['A2']
    ### END CODE HERE ###

    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on sl
    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m
    ### END CODE HERE ###

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

```
parameters, cache, X_assess, Y_assess = backward_propagation_test_case()
```

```
grads = backward_propagation(parameters, cache, X_assess, Y_assess)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dW2 = "+ str(grads["dW2"]))
print ("db2 = "+ str(grads["db2"]))
```

```
dW1 = [[ 0.01018708 -0.00708701]
 [ 0.00873447 -0.0060768 ]
 [-0.00530847  0.00369379]
 [-0.02206365  0.01535126]]
db1 = [[-0.00069728]
 [-0.00060606]
 [ 0.000364  ]
 [ 0.00151207]]
dW2 = [[ 0.00363613  0.03153604  0.01162914 -0.01318316]]
db2 = [[0.06589489]]
```

Expected output:

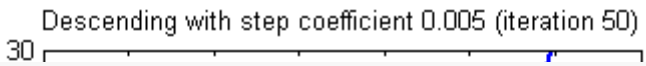
dW1 =	[[0.01018708 -0.00708701] [0.00873447 -0.0060768] [-0.00530847 0.00369379] [-0.02206365 0.01535126]]
db1 =	[[-0.00069728] [-0.00060606] [0.000364] [0.00151207]]
dW2 =	[[0.00363613 0.03153604 0.01162914 -0.01318316]]
db2 =	[[0.06589489]]

Question: Implement the update rule. Use gradient descent. You have to use (dW1, db1, dW2, db2) in order to update (W1, b1, W2, b2).

General gradient descent rule: $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ where α is the learning rate and θ represents a parameter.

Illustration: The gradient descent algorithm with a good learning rate (converging) and a bad learning rate (diverging). Images courtesy of Adam Harley.

```
%%html
<img src="images/sgd.gif" style="width:400;height:400;"> <img src="images/sgd_bad.
```




```
# GRADED FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Updates parameters using the gradient descent update rule given above

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    ### END CODE HERE ###

    # Retrieve each gradient from the dictionary "grads"
    ### START CODE HERE ### (≈ 4 lines of code)
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']
    ## END CODE HERE ###

    # Update rule for each parameter
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    ### END CODE HERE ###

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
parameters, grads = update_parameters_test_case()
parameters = update_parameters(parameters, grads)

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00643025  0.01936718]
 [-0.02410458  0.03978052]
 [-0.01653973 -0.02096177]
 [ 0.01046864 -0.05990141]]
b1 = [[-1.02420756e-06]
 [ 1.27373948e-05]
 [ 8.32996807e-07]
 [-3.20136836e-06]]
W2 = [[-0.01041081 -0.04463285  0.01758031  0.04747113]]
b2 = [[0.00010457]]
```

Expected Output:

W1 =	[[-0.00643025 0.01936718] [-0.02410458 0.03978052] [-0.01653973 -0.02096177] [0.01046864 -0.05990141]]
b1 =	[[-1.02420756e-06] [1.27373948e-05] [8.32996807e-07] [-3.20136836e-06]]
W2 =	[[-0.01041081 -0.04463285 0.01758031 0.04747113]]
b2 =	[[0.00010457]]

4.4 - Integrate parts 4.1, 4.2 and 4.3 in nn_model()

Question: Build your neural network model in `nn_model()`.

Instructions: The neural network model has to use the previous functions in the right order.

```
# GRADED FUNCTION: nn_model

def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
    Arguments:
    X -- dataset of shape (2, number of examples)
    Y -- labels of shape (1, number of examples)
    n_h -- size of the hidden layer
    num_iterations -- Number of iterations in gradient descent loop
    print_cost -- if True, print the cost every 1000 iterations

    Returns:
    parameters -- parameters learnt by the model. They can then be used to predict
    """


    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]

    # Initialize parameters, then retrieve W1, b1, W2, b2. Inputs: "n_x, n_h, n_y"
    ### START CODE HERE ### (≈ 5 lines of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    ### END CODE HERE ###
```

```
    # Loop (gradient descent)

    for i in range(0, num_iterations):

        ### START CODE HERE ### (≈ 4 lines of code)
        # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
        A2, cache = forward_propagation(X, parameters)

        # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
        cost = compute_cost(A2, Y, parameters)

        # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
        grads = backward_propagation(parameters, cache, X, Y)

        # Gradient descent parameter update. Inputs: "parameters, grads". Outputs:
        parameters = update_parameters(parameters, grads)

        ### END CODE HERE ###

        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

```
X_assess, Y_assess = nn_model_test_case()

parameters = nn_model(X_assess, Y_assess, 4, num_iterations=10000, print_cost=Fals
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:20: Ru
/content/planar_utils.py:34: RuntimeWarning: overflow encountered i
  s = 1/(1+np.exp(-x))
W1 = [[-4.18493493  5.33221044]
 [-7.52989391  1.24306177]
 [-4.19294751  5.32632293]
 [ 7.52983635 -1.24309475]]
b1 = [[ 2.32926716]
 [ 3.79458985]
 [ 2.33002427]
 [-3.79469013]]
W2 = [[-6033.83672449 -6008.12980599 -6033.1009563   6008.06638407]
```

Expected Output:

W1 =	[[-4.18494056 5.33220609] [-7.52989382 1.24306181] [-4.1929459 5.32632331] [7.52983719 -1.24309422]]
b1 =	[[2.32926819] [3.79458998] [2.33002577] [-3.79468846]]
W2 =	[[-6033.83672146 -6008.12980822 -6033.10095287 6008.06637269]]
b2 =	[[-52.66607724]]

4.5 Predictions

Question: Use your model to predict by building predict(). Use forward propagation to predict results.

Reminder: predictions =

$$y_{prediction} = 1\{\text{activation} > 0.5\} = \begin{cases} 1 & \text{if } activation > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

As an example, if you would like to set the entries of a matrix X to 0 and 1 based on a threshold you would do: `X_new = (X > threshold)`

```
# GRADED FUNCTION: predict

def predict(parameters, X):
    """
    Using the learned parameters, predicts a class for each example in X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (n_x, m)

    Returns
    predictions -- vector of predictions of our model (red: 0 / blue: 1)
    """

    # Computes probabilities using forward propagation, and classifies to 0/1 usin
    ### START CODE HERE ### (≈ 2 lines of code)
    A2, cache = forward_propagation(X, parameters)
    predictions = (A2 > .5)
    ### END CODE HERE ###

    return predictions
```

```
parameters, X_assess = predict_test_case()

predictions = predict(parameters, X_assess)
print("predictions mean = " + str(np.mean(predictions)))
```

```
predictions mean = 0.6666666666666666
```

Expected Output:

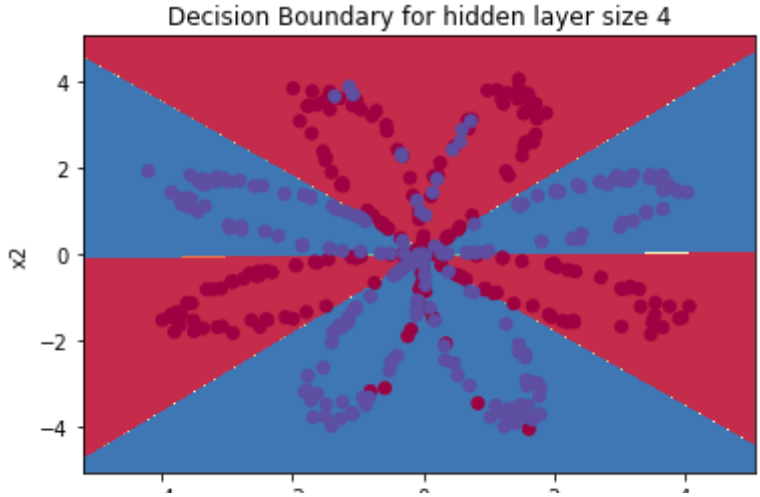
predictions mean = 0.666666666667

It is time to run the model and see how it performs on a planar dataset. Run the following code to test your model with a single hidden layer of n_h hidden units.

```
# Build a model with a n_h-dimensional hidden layer
```

```
parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000, print_cost=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.693048
Cost after iteration 1000: 0.288083
Cost after iteration 2000: 0.254385
Cost after iteration 3000: 0.233864
Cost after iteration 4000: 0.226792
Cost after iteration 5000: 0.222644
Cost after iteration 6000: 0.219731
Cost after iteration 7000: 0.217504
Cost after iteration 8000: 0.219556
Cost after iteration 9000: 0.218585
Text(0.5, 1.0, 'Decision Boundary for hidden layer size 4')
```



Expected Output:

Cost after iteration 9000:	0.218607

```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.
```

```
Accuracy: 90%
```

Expected Output:

Accuracy:	90%

Accuracy is really high compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

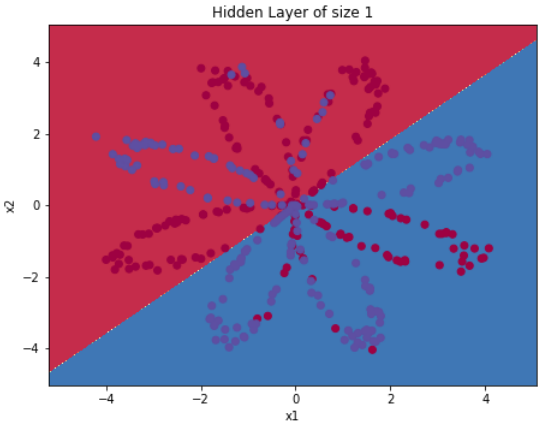
Now, let's try out several hidden layer sizes.

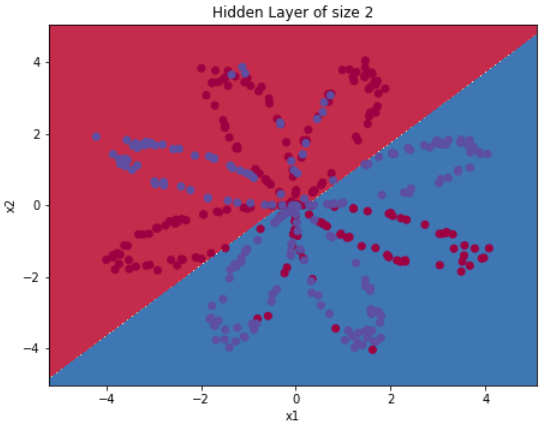
4.6 - Tuning hidden layer size

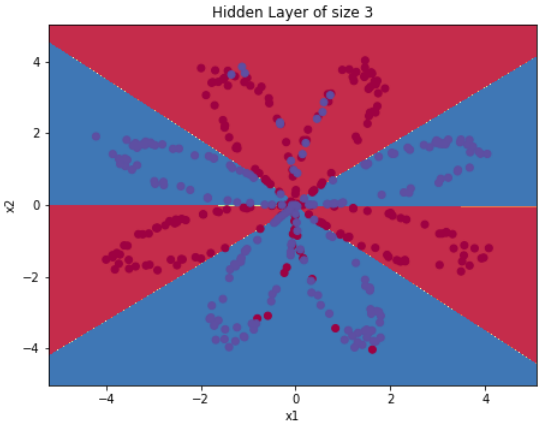
Run the following code. It may take 1-2 minutes. You will observe different behaviors of the model for various hidden layer sizes.

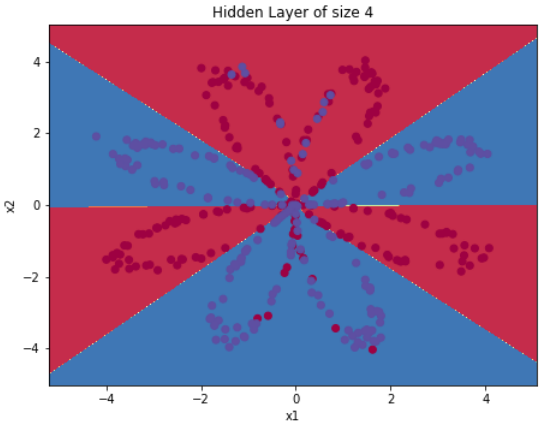
```
%time
# This may take about 2 minutes to run
from tqdm import tqdm

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
for i, n_h in tqdm(enumerate(hidden_layer_sizes)):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

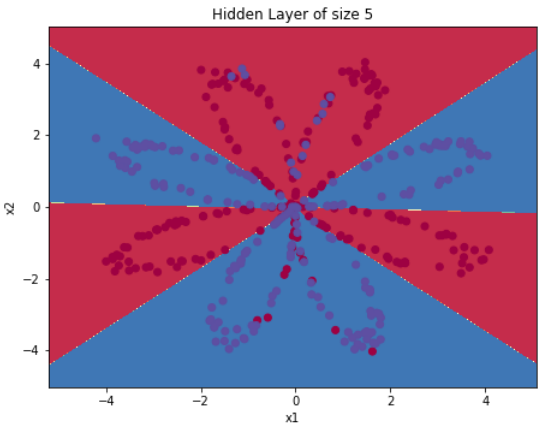
```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 6.44 µs
1it [00:00,  1.09it/s]Accuracy for 1 hidden units: 67.5 %
2it [00:01,  1.01s/it]Accuracy for 2 hidden units: 67.25 %
3it [00:03,  1.09s/it]Accuracy for 3 hidden units: 90.75 %
4it [00:04,  1.18s/it]Accuracy for 4 hidden units: 90.5 %
5it [00:05,  1.29s/it]Accuracy for 5 hidden units: 91.25 %
6it [00:09,  2.00s/it]Accuracy for 20 hidden units: 90.0 %
7it [00:19,  2.77s/it]Accuracy for 50 hidden units: 90.25 %
```

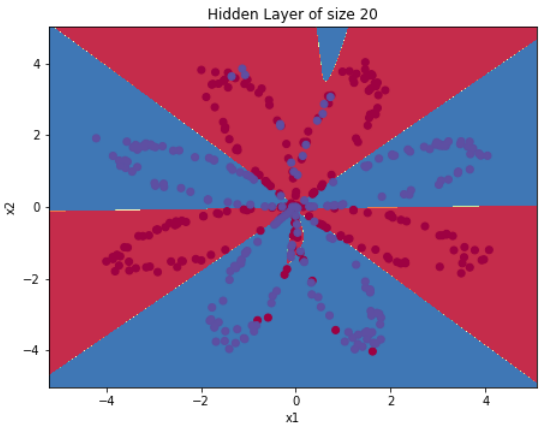


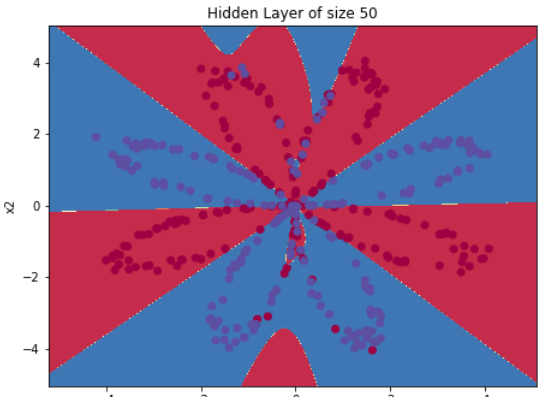












Interpretation:

- The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data.
- The best hidden layer size seems to be around n_h = 5. Indeed, a value around here seems to fits the data well without also incurring noticable overfitting.
- You will also learn later about regularization, which lets you use very large models (such as n_h = 50) without much overfitting.

Optional questions:

Note: Remember to submit the assignment but clicking the blue "Submit Assignment" button at the upper-right.

Some optional/ungraded questions that you can explore if you wish:

- What happens when you change the tanh activation for a sigmoid activation or a ReLU activation?
- Play with the learning_rate. What happens?
- What if we change the dataset? (See part 5 below!)

You've learnt to:

- Build a complete neural network with a hidden layer
- Make a good use of a non-linear unit
- Implemented forward propagation and backpropagation, and trained a neural network
- See the impact of varying the hidden layer size, including overfitting.

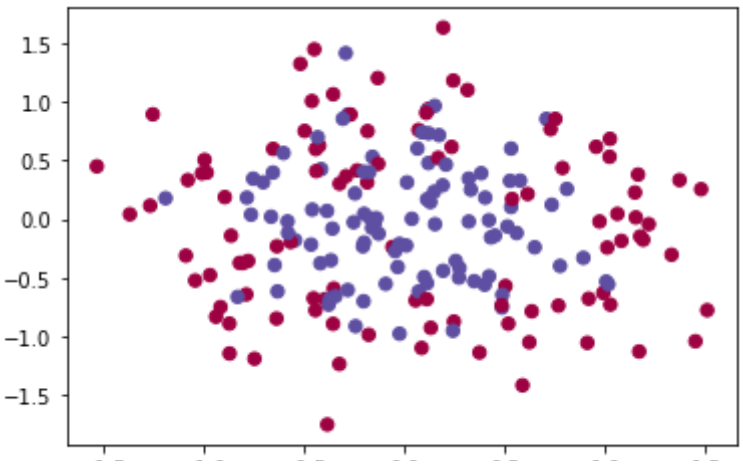
Nice work!

5) Performance on other datasets

If you want, you can rerun the whole notebook (minus the dataset part) for each of the following datasets.

```
# Datasets
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure = load_extra_d

datasets = {"noisy_circles": noisy_circles,
            "noisy_moons": noisy_moons,
            "blobs": blobs,
            "gaussian_quantiles": gaussian_quantiles,
            "no_structure": no_structure}

### START CODE HERE ### (choose your dataset)
```

```
dataset = "noisy_circles"
### END CODE HERE ###

X, Y = datasets[dataset]
X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary
if dataset == "blobs":
    Y = Y%2

# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



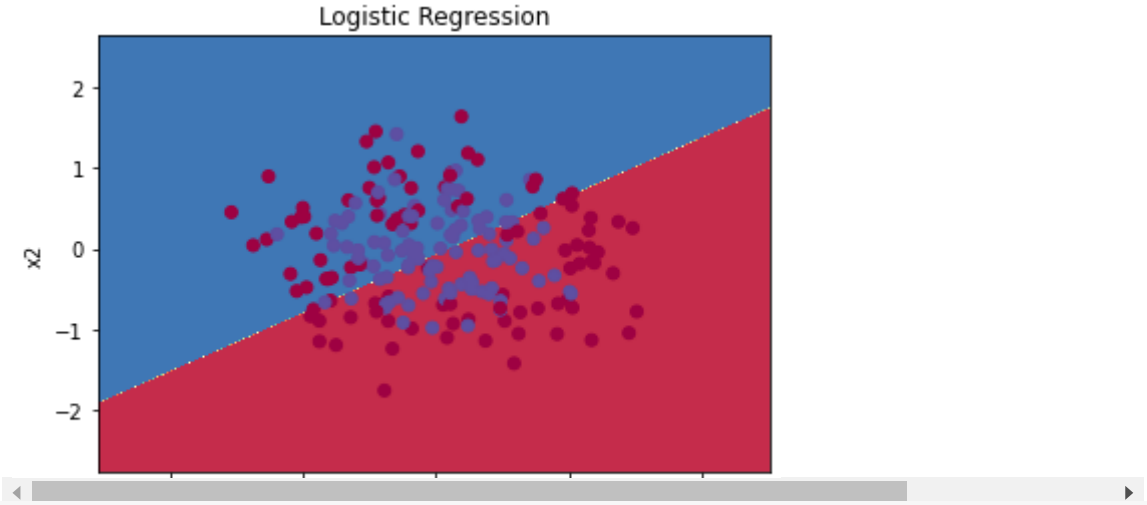

```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:
  y = column_or_1d(y, warn=True)
```

You can now plot the decision boundary of these models. Run the code below.

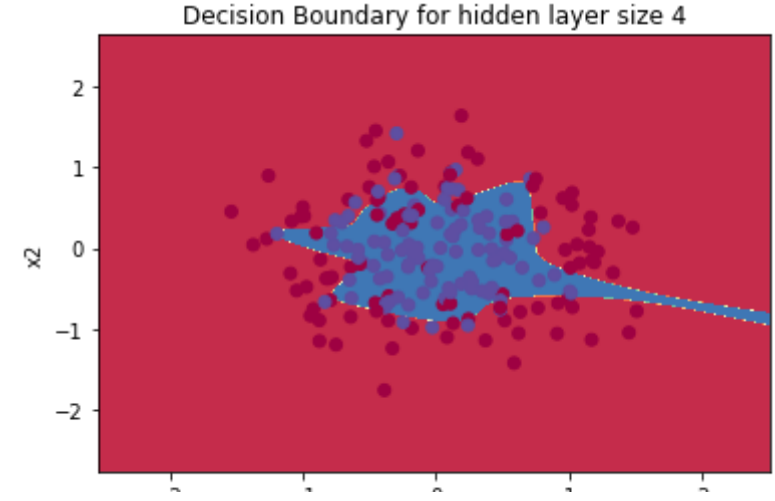
```
# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")

# Print accuracy
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 54 % (percentage of correctly labe
```




```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 10, num_iterations = 10000, print_cost=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.693146
Cost after iteration 1000: 0.446865
Cost after iteration 2000: 0.410573
Cost after iteration 3000: 0.399398
Cost after iteration 4000: 0.395236
Cost after iteration 5000: 0.390120
Cost after iteration 6000: 0.385051
Cost after iteration 7000: 0.381602
Cost after iteration 8000: 0.379797
Cost after iteration 9000: 0.378530
Text(0.5, 1.0, 'Decision Boundary for hidden layer size 4')
```



Expected Output:

```
Cost after iteration 9000:  0.218607
```

```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.
```

```
Accuracy: 79%
```

Expected Output:

```
Accuracy:  90%
```

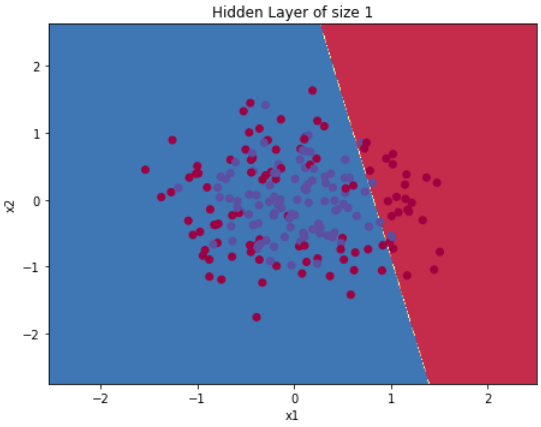
Accuracy is really high compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

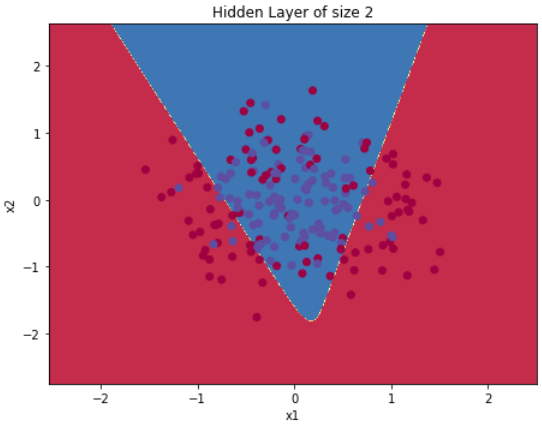
Now, let's try out several hidden layer sizes.

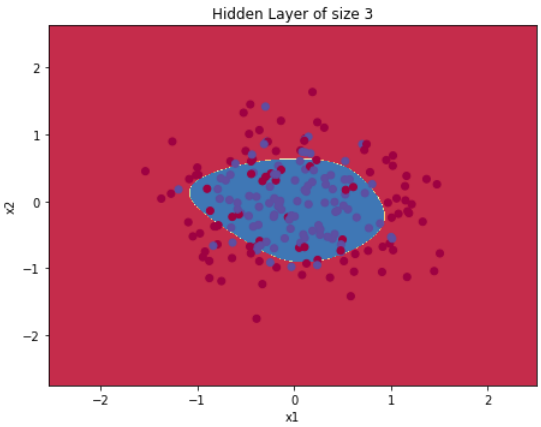
```

```

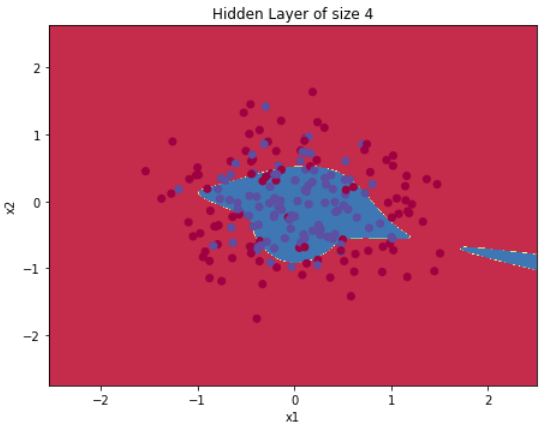
```
# This may take about 2 minutes to run
from tqdm import tqdm

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 6, 8, 16, 32, 64]
for i, n_h in tqdm(enumerate(hidden_layer_sizes)):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 10000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

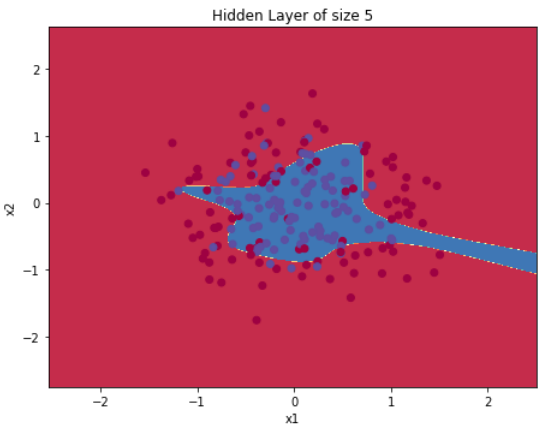
```
1it [00:01,  1.11s/it]Accuracy for 1 hidden units: 59.5 %
2it [00:02,  1.21s/it]Accuracy for 2 hidden units: 72.5 %
3it [00:03,  1.31s/it]Accuracy for 3 hidden units: 79.0 %
4it [00:05,  1.38s/it]Accuracy for 4 hidden units: 76.0 %
5it [00:06,  1.49s/it]Accuracy for 5 hidden units: 79.5 %
6it [00:08,  1.57s/it]Accuracy for 6 hidden units: 79.0 %
7it [00:10,  1.68s/it]Accuracy for 8 hidden units: 78.5 %
8it [00:13,  1.99s/it]Accuracy for 16 hidden units: 80.5 %
9it [00:17,  2.80s/it]Accuracy for 32 hidden units: 81.5 %
10it [00:28,  2.87s/it]Accuracy for 64 hidden units: 81.0 %
```

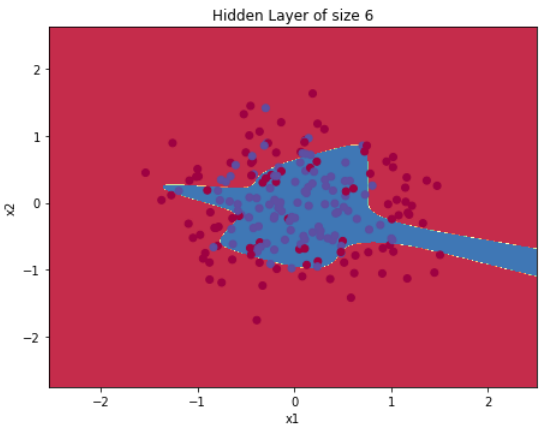


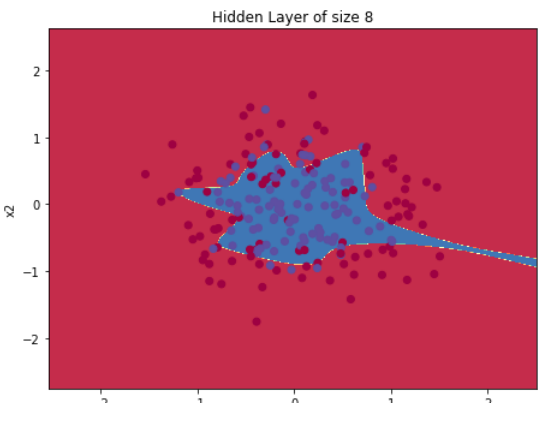


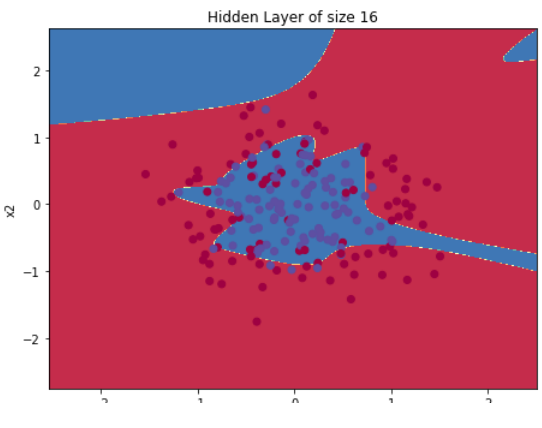


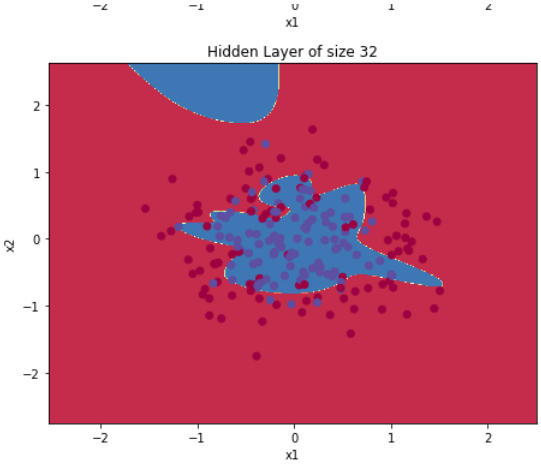


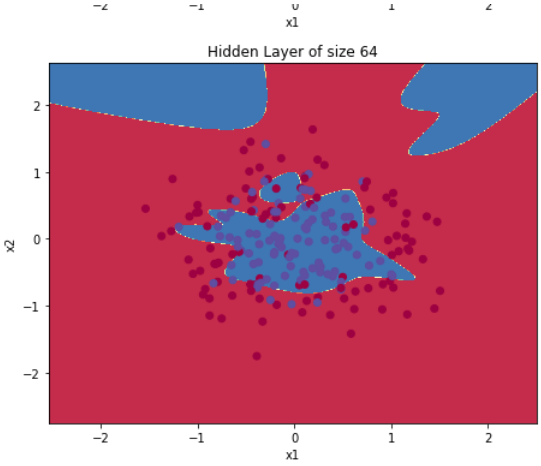




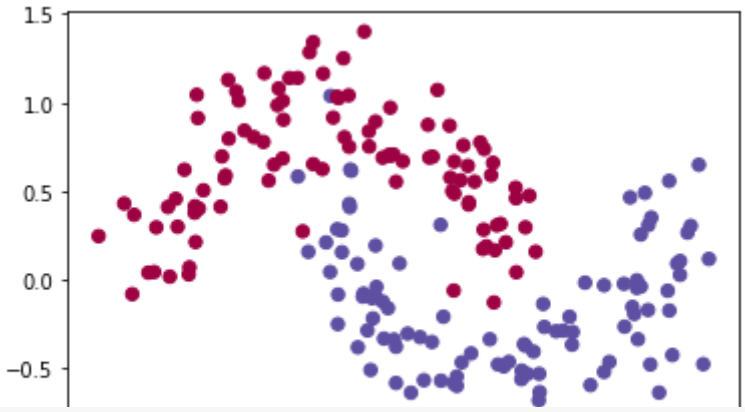






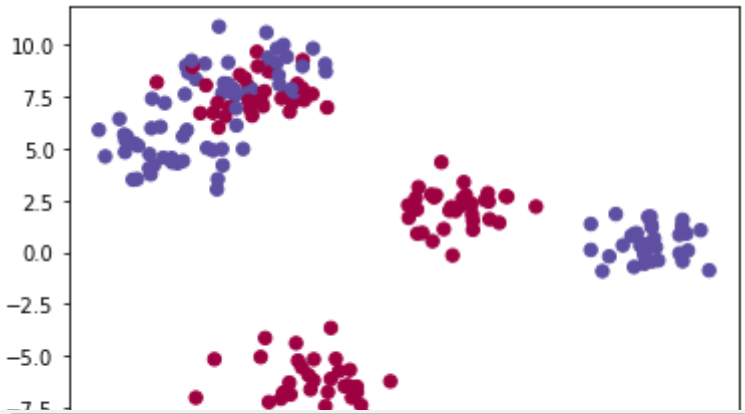



```
### START CODE HERE ### (choose your dataset)
dataset = "noisy_moons"
### END CODE HERE ###

X, Y = datasets[dataset]
X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary
if dataset == "blobs":
    Y = Y%2

# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```

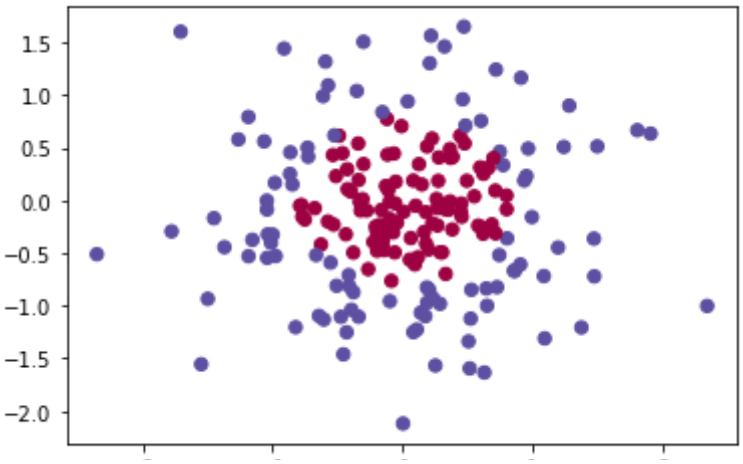



```
### START CODE HERE ### (choose your dataset)
dataset = "blobs"
### END CODE HERE ###

X, Y = datasets[dataset]
print(Y.shape[0])
print(Y)
X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary
if dataset == "blobs":
    Y = Y%2
print(Y)
# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```

```
200
[2 3 4 3 1 2 5 2 0 2 0 1 0 0 4 2 0 4 4 2 4 1 2 5 1 0 5 0 2 3 4 2 3
 0 2 5 0 4 1 1 0 0 3 5 0 0 2 0 3 0 5 2 5 2 1 0 2 4 4 2 5 4 3 5 1 4
 1 4 1 3 1 3 2 3 4 5 1 4 4 0 0 2 4 2 5 4 0 4 5 3 1 3 3 1 0 1 2 1 1
 0 5 2 1 5 3 3 1 4 2 5 1 3 3 5 2 1 3 4 0 5 1 1 1 5 3 1 0 0 5 3 5 5
 1 0 5 4 2 5 5 2 0 0 1 3 3 3 3 0 2 4 0 4 4 1 3 0 4 5 2 4 4 5 1 1 3
 1 2 5 0 4 3 1 5 4 3 4 2 5 3 0]
[[0 1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 0 0 1
  0 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 1 1 1
  0 1 1 0 1 1 1 1 0 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0
  1 0 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 1 1 0 0 1
  0 1 1 0 1 0 1 0 0 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0
  1 1 1 0 0 1 0 1 0 0 1 1 1 0 1 0 0 1 1 0]]
```



```
### START CODE HERE ### (choose your dataset)
dataset = "gaussian_quantiles"
### END CODE HERE ###

X, Y = datasets[dataset]
X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary
if dataset == "blobs":
    Y = Y%2

# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```

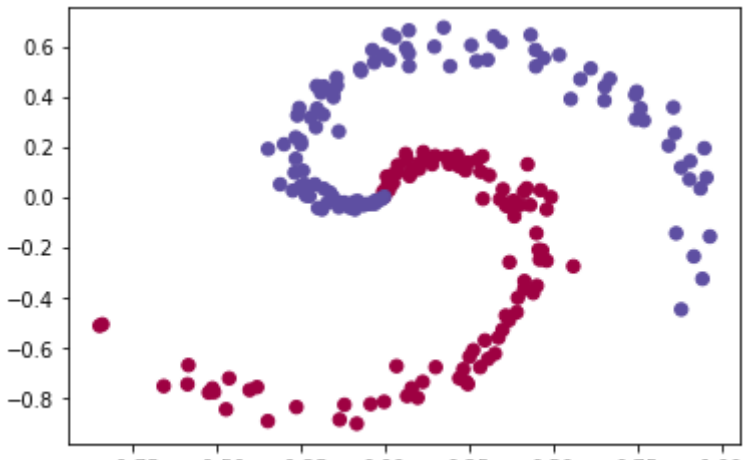



```
### START CODE HERE ### (choose your dataset)
dataset = "no_structure"
### END CODE HERE ###

X, Y = datasets[dataset]
#X, Y = X.T, Y.reshape(1, Y.shape[0])
# Adapt data formats
X, Y = X.T, Y[:,0].reshape(1, Y.shape[0])

# make "no_structure" binary
if dataset == "no_structure":
    Y = np.round(Y)
    #print(Y)

# Visualize the data
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```

Variant 5

Dataset - visualize the data

```
N = 100 # number of points per class
D = 2 # dimensionality
K = 2 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels
for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j

# Conversion to provide compatibility with other examples
Y = y
X, Y = X.T, Y.T
Y = Y.reshape(1, Y.shape[0])
# lets visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral)
plt.show()
```



Train the logistic regression classifier

```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
```

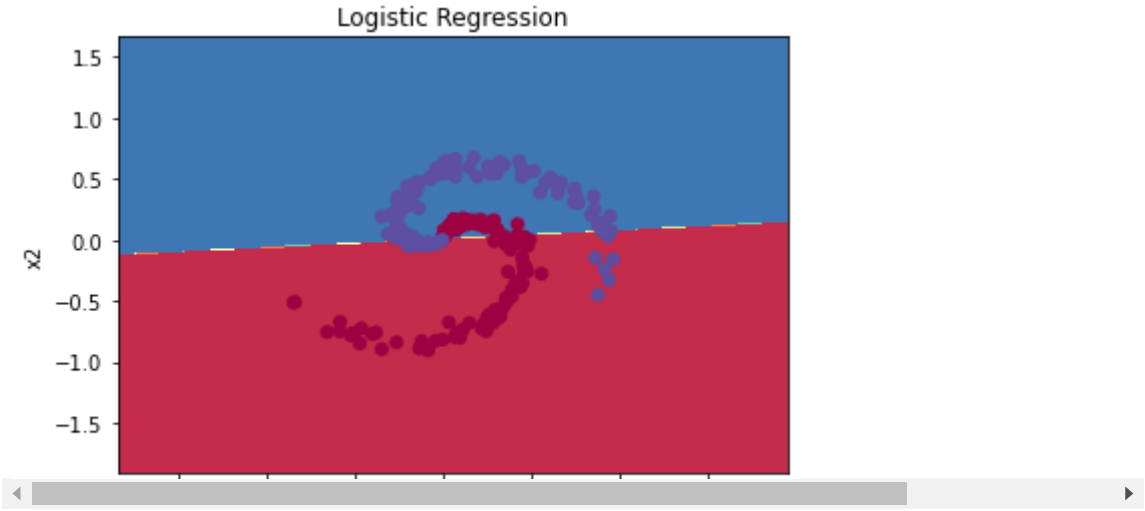
```
clf.fit(X.T, Y.T);
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:
  y = column_or_1d(y, warn=True)
```

Plot the decision boundary for logistic regression

```
# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")

# Print accuracy
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

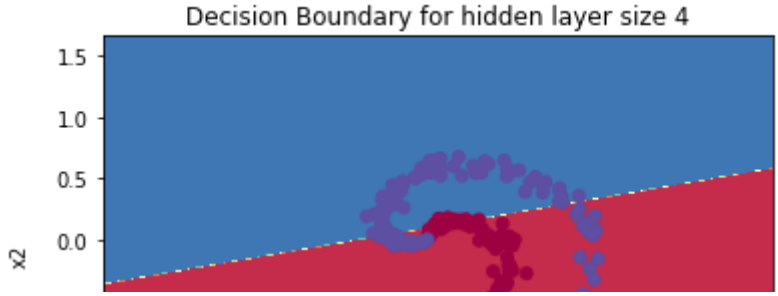
```
Accuracy of logistic regression: 70 % (percentage of correctly labe
```



Build a model with a n_h-dimensional hidden layer

```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 1, num_iterations = 10000, print_cost=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.693145
Cost after iteration 1000: 0.432859
Cost after iteration 2000: 0.433114
Cost after iteration 3000: 0.433357
Cost after iteration 4000: 0.433544
Cost after iteration 5000: 0.433691
Cost after iteration 6000: 0.433812
Cost after iteration 7000: 0.433912
Cost after iteration 8000: 0.433998
Cost after iteration 9000: 0.434073
Text(0.5, 1.0, 'Decision Boundary for hidden layer size 4')
```



Determine accuracy

```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.
```

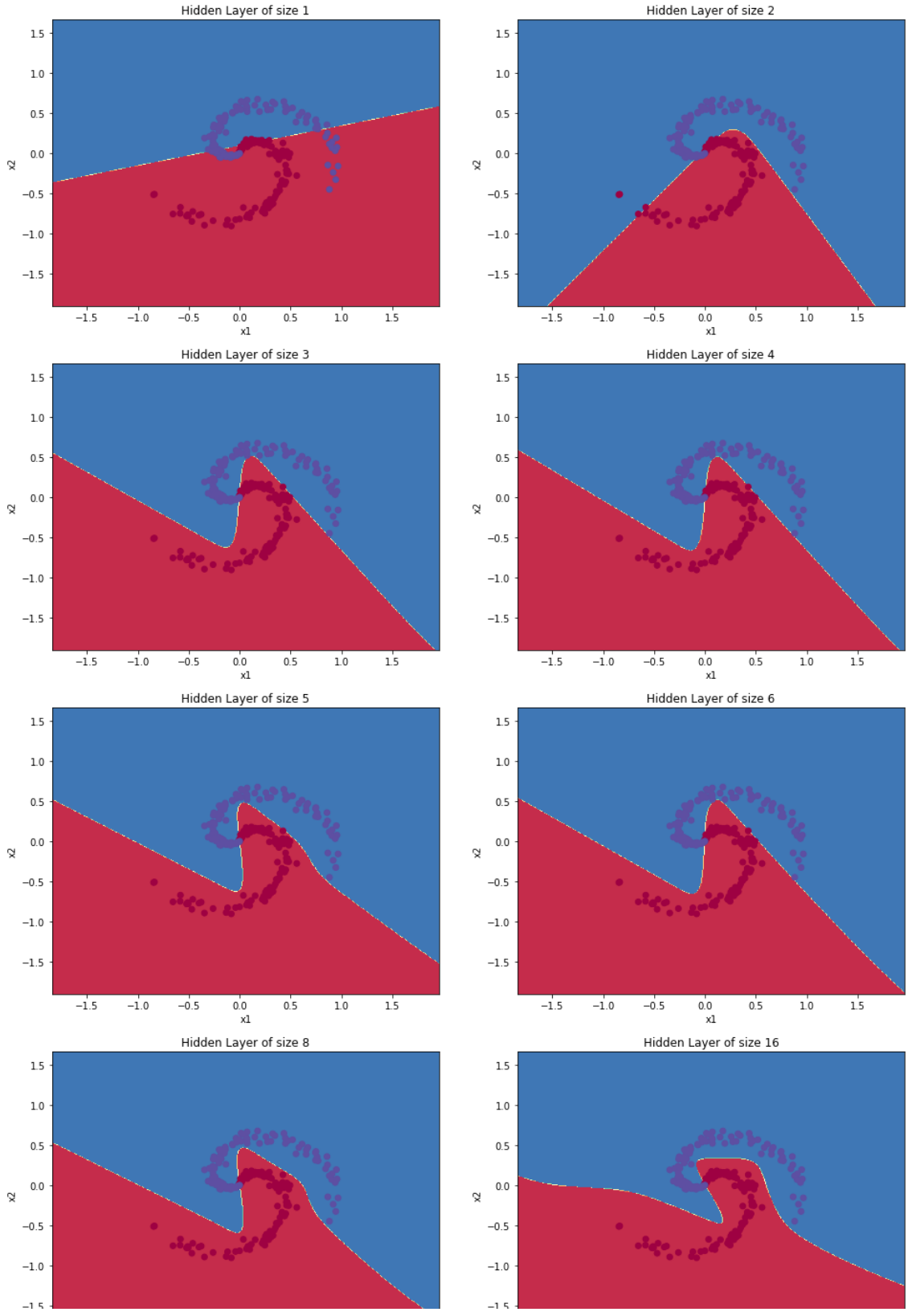
```
Accuracy: 76%
```

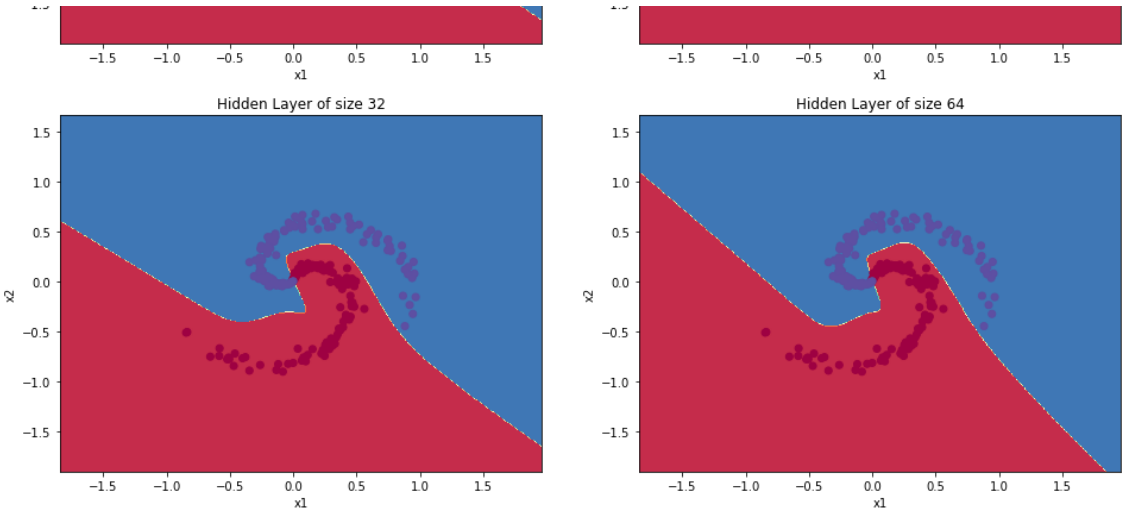
Try out several **numbers of hidden layer sizes**

Find the **optimal** number: when **accuracy start to be saturated**!

```
%time
# This may take about 2 minutes to run
from tqdm import tqdm

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 6, 8, 16, 32, 64]
for i, n_h in tqdm(enumerate(hidden_layer_sizes)):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 10000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

```
CPU times: user 2 µs, sys: 1 µs, total: 3 µs
Wall time: 7.15 µs
1it [00:01,  1.13s/it]Accuracy for 1 hidden units: 76.0 %
2it [00:02,  1.24s/it]Accuracy for 2 hidden units: 92.0 %
3it [00:03,  1.28s/it]Accuracy for 3 hidden units: 99.0 %
4it [00:05,  1.32s/it]Accuracy for 4 hidden units: 99.0 %
5it [00:06,  1.40s/it]Accuracy for 5 hidden units: 99.5 %
6it [00:08,  1.47s/it]Accuracy for 6 hidden units: 99.0 %
7it [00:10,  1.58s/it]Accuracy for 8 hidden units: 99.5 %
8it [00:12,  1.87s/it]Accuracy for 16 hidden units: 99.5 %
9it [00:16,  2.46s/it]Accuracy for 32 hidden units: 99.5 %
10it [00:26,  2.66s/it]Accuracy for 64 hidden units: 99.5 %
```

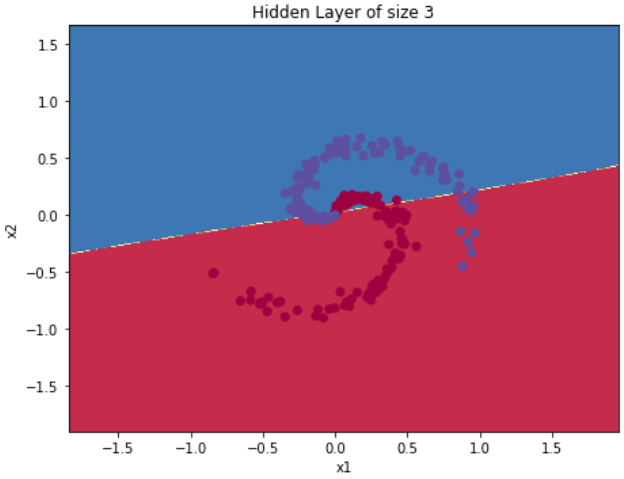


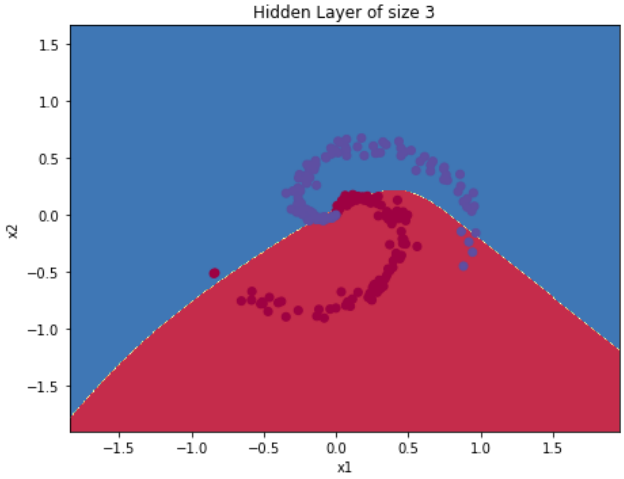


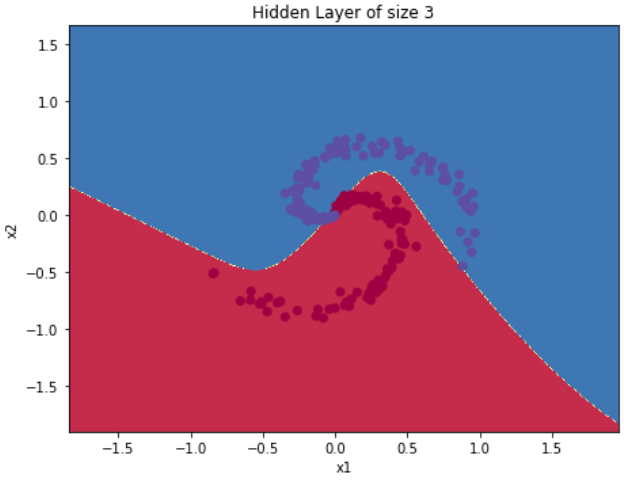
- Try out several **numbers of iterations** for the optimal hidden layer size

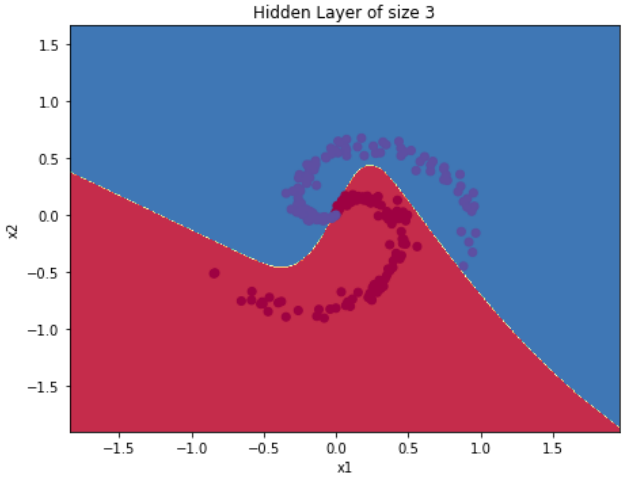
  Find the **optimal** number: when **accuracy start to be saturated**!

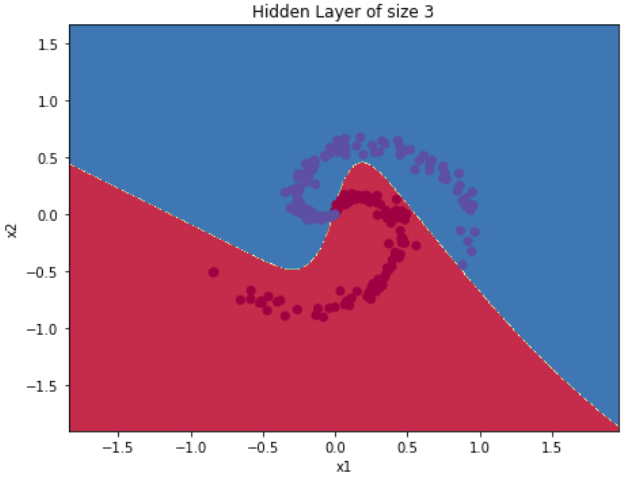
```
%time
# This may take about 2 minutes to run

plt.figure(figsize=(16, 32))
num_iterations = [100, 200, 300, 400, 500, 600, 800, 1600, 3200, 6400]
n_h = 3
for i, n_i in enumerate(num_iterations):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, n_i)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float
    print ("Accuracy for {} iterations: {} %".format(n_i, accuracy))
```

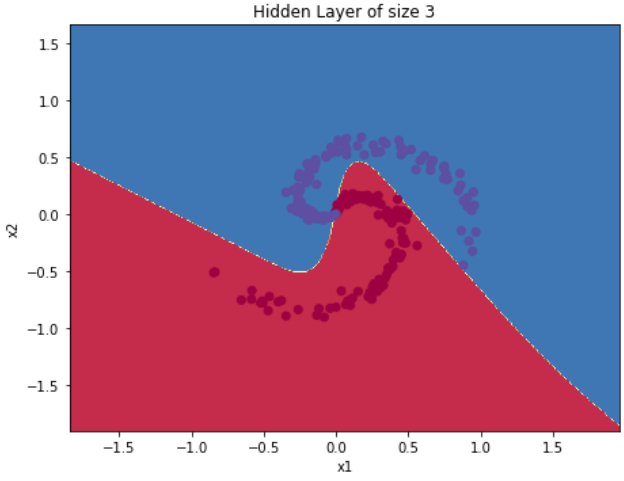


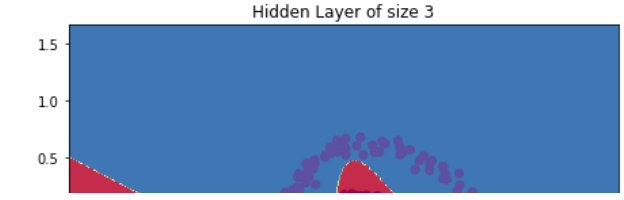


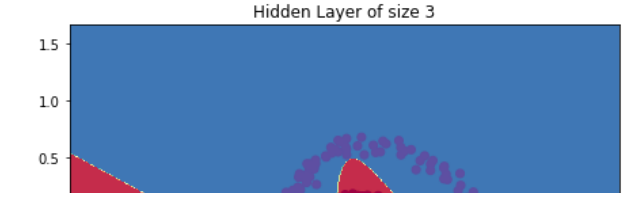




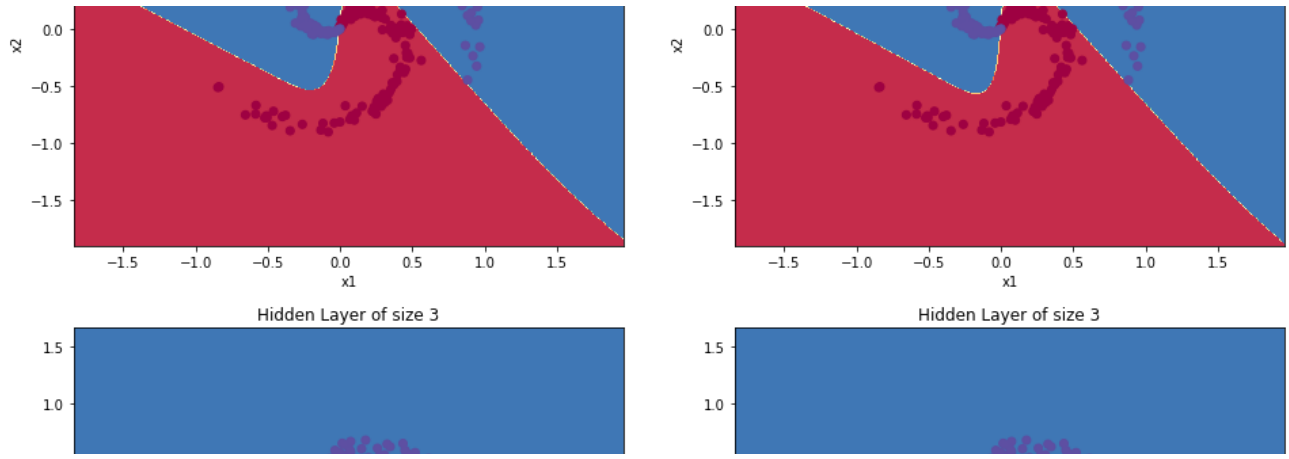






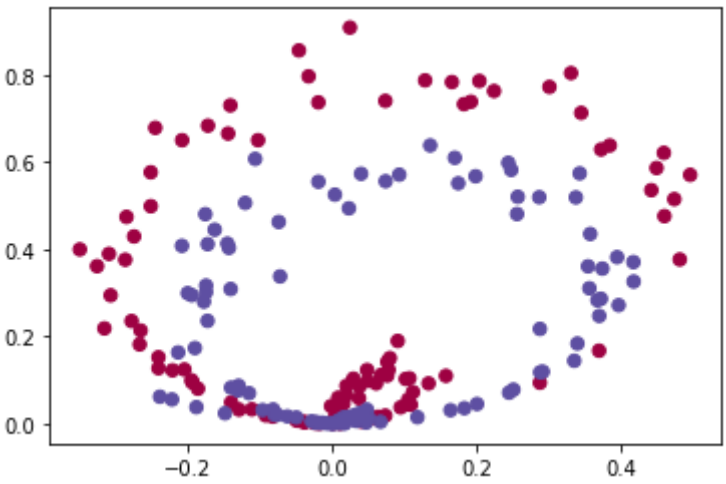


```
CPU times: user 3 µs, sys: 1e+03 ns, total: 4 µs
Wall time: 6.68 µs
Accuracy for 100 iterations: 70.5 %
Accuracy for 200 iterations: 84.5 %
Accuracy for 300 iterations: 95.0 %
Accuracy for 400 iterations: 97.5 %
Accuracy for 500 iterations: 98.5 %
Accuracy for 600 iterations: 98.5 %
Accuracy for 800 iterations: 99.0 %
Accuracy for 1600 iterations: 99.0 %
Accuracy for 3200 iterations: 99.0 %
Accuracy for 6400 iterations: 99.0 %
```



Variant 6

```
N = 100 # number of points per class
D = 2 # dimensionality
K = 2 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels
for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t)*np.cos(t), r*np.cos(t)*np.cos(t)]
  y[ix] = j

# Conversion to provide compatibility with other examples
Y = y
X, Y = X.T, Y.T
Y = Y.reshape(1, Y.shape[0])
# lets visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral)
plt.show()
```



Train the logistic regression classifier

```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: DataC
  y = column_or_1d(y, warn=True)
```

Plot the decision boundary for logistic regression

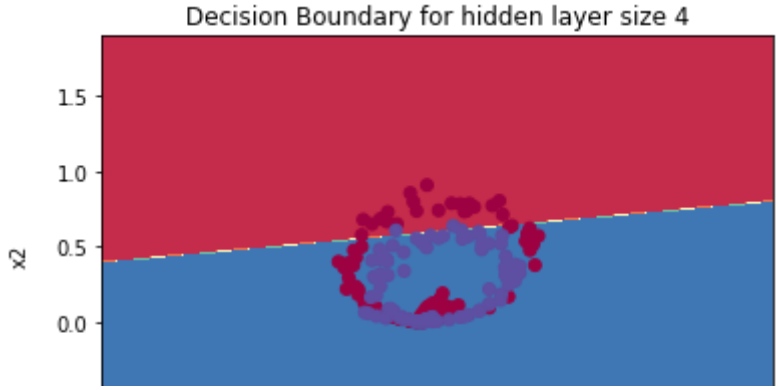
```
# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")

# Print accuracy
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 56 % (percentage of correctly labelled datap
```

Build a model with a n_h-dimensional hidden layer

```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(X, Y, n_h = 1, num_iterations = 10000, print_cost=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.693147
Cost after iteration 1000: 0.661838
Cost after iteration 2000: 0.636435
Cost after iteration 3000: 0.630370
Cost after iteration 4000: 0.627855
Cost after iteration 5000: 0.626443
Cost after iteration 6000: 0.625518
Cost after iteration 7000: 0.624854
Cost after iteration 8000: 0.624348
Cost after iteration 9000: 0.623945
Text(0.5, 1.0, 'Decision Boundary for hidden layer size 4')
```



Determine accuracy

```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.
```

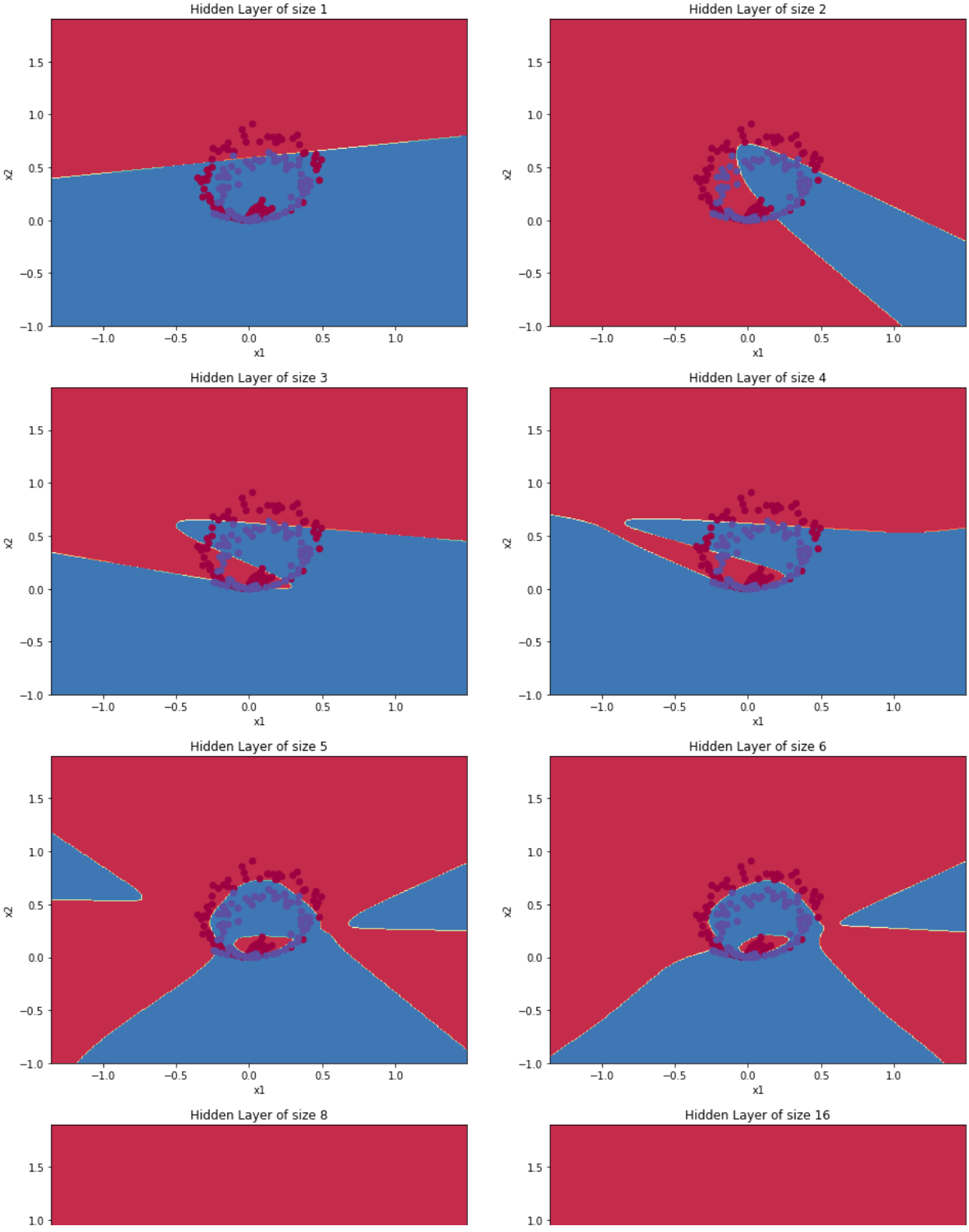
```
Accuracy: 59%
```

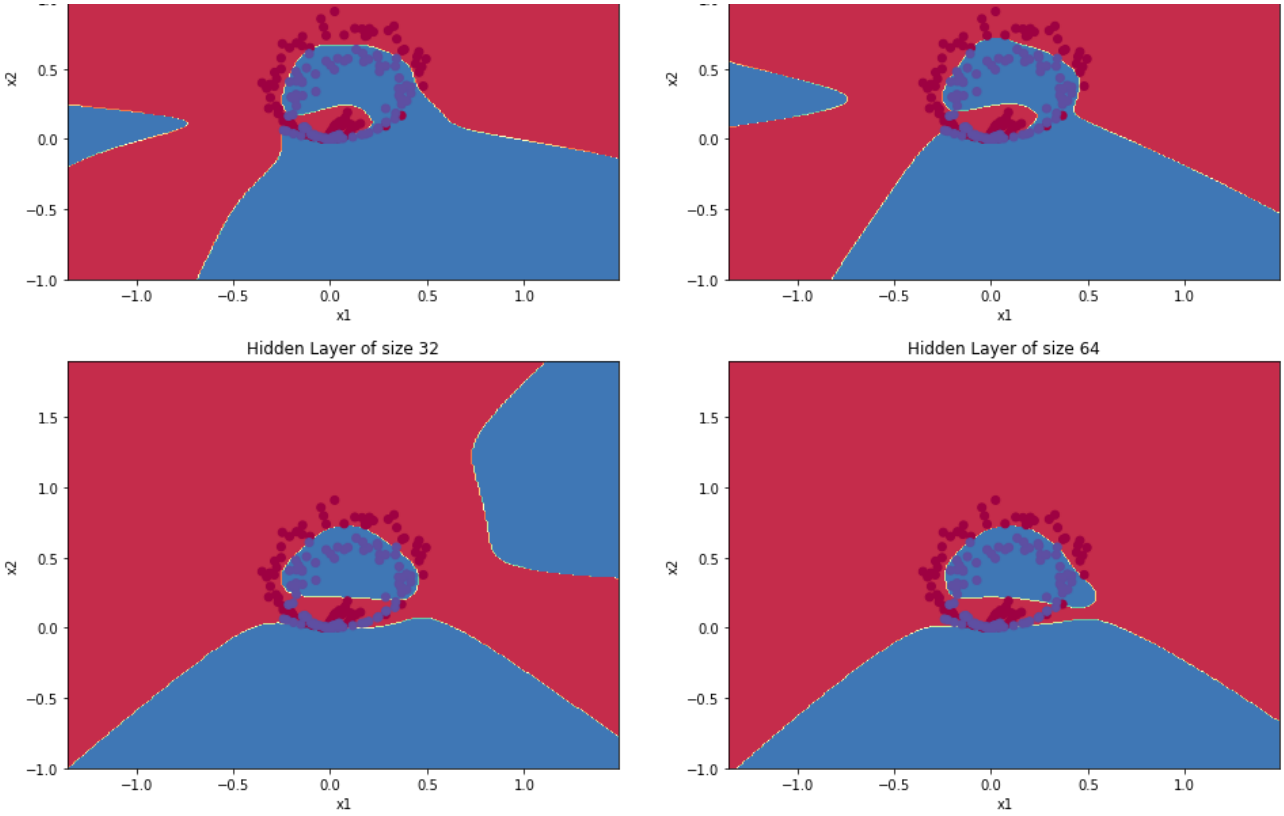
Try out several **numbers of hidden layer sizes**

Find the **optimal** number: when **accuracy start to be saturated**!

```
%time
# This may take about 2 minutes to run
from tqdm import tqdm

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 6, 8, 16, 32, 64]
for i, n_h in tqdm(enumerate(hidden_layer_sizes)):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 10000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

```
CPU times: user 3 µs, sys: 1e+03 ns, total: 4 µs
Wall time: 8.82 µs
1it [00:01,  1.04s/it]Accuracy for 1 hidden units: 59.5 %
2it [00:02,  1.15s/it]Accuracy for 2 hidden units: 65.0 %
3it [00:03,  1.24s/it]Accuracy for 3 hidden units: 72.0 %
4it [00:05,  1.30s/it]Accuracy for 4 hidden units: 72.0 %
5it [00:06,  1.37s/it]Accuracy for 5 hidden units: 84.0 %
6it [00:08,  1.44s/it]Accuracy for 6 hidden units: 85.0 %
7it [00:09,  1.53s/it]Accuracy for 8 hidden units: 82.5 %
8it [00:12,  1.82s/it]Accuracy for 16 hidden units: 80.0 %
9it [00:15,  2.32s/it]Accuracy for 32 hidden units: 71.5 %
10it [00:25,  2.51s/it]Accuracy for 64 hidden units: 76.0 %
```



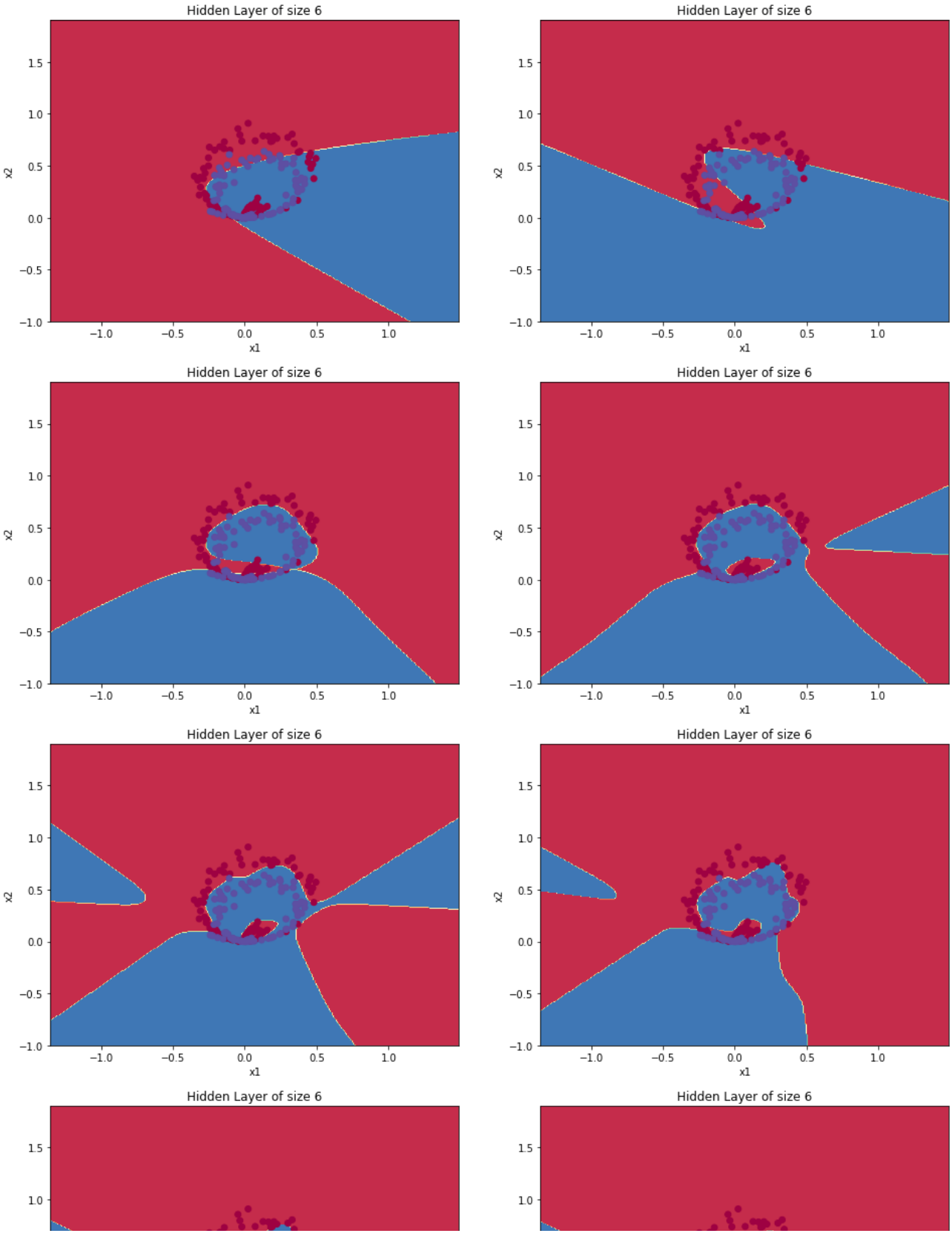


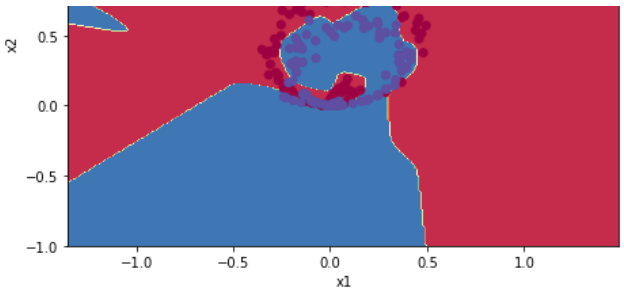
- Try out several **numbers of iterations** for the optimal hidden layer size

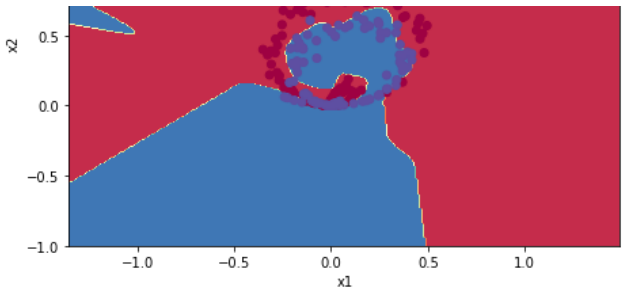
  Find the **optimal** number: when **accuracy start to be saturated**!

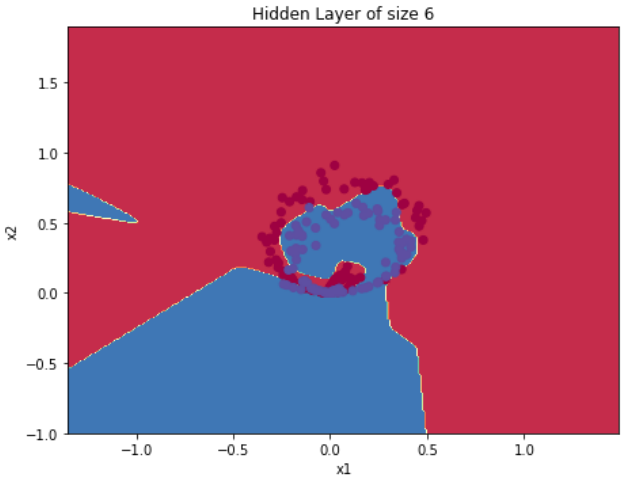
```
# This may take about 2 minutes to run
from tqdm import tqdm

plt.figure(figsize=(16, 32))
num_iterations = [1000, 2000, 5000, 10000, 20000, 40000, 80000, 160000, 320000, 64
n_h = 6
for i, n_i in tqdm(enumerate(num_iterations)):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, n_i)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float
    print ("Accuracy for {} iterations: {} %".format(n_i, accuracy))
```

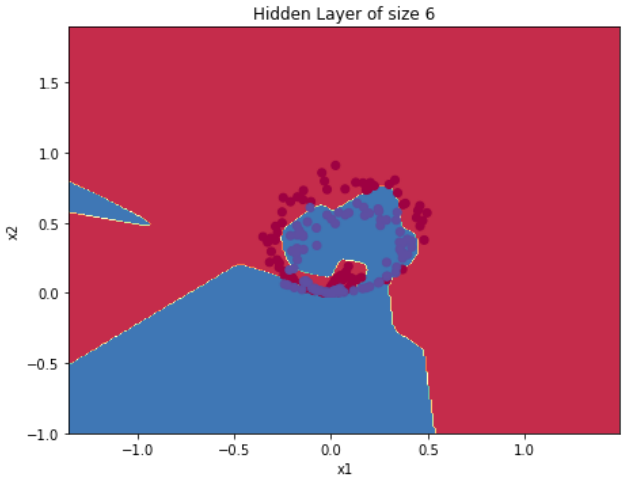
```
1it [00:00,  4.83it/s]Accuracy for 1000 iterations: 61.5 %
2it [00:00,  3.03it/s]Accuracy for 2000 iterations: 66.5 %
3it [00:01,  1.72it/s]Accuracy for 5000 iterations: 79.5 %
4it [00:03,  1.00s/it]Accuracy for 10000 iterations: 85.0 %
5it [00:06,  1.77s/it]Accuracy for 20000 iterations: 84.5 %
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:20: RuntimeWarni
6it [00:12,  3.35s/it]Accuracy for 40000 iterations: 85.0 %
7it [00:24,  6.20s/it]Accuracy for 80000 iterations: 86.0 %
8it [00:48, 11.87s/it]Accuracy for 160000 iterations: 81.0 %
9it [01:36, 23.03s/it]Accuracy for 320000 iterations: 86.0 %
10it [03:11, 19.18s/it]Accuracy for 640000 iterations: 86.0 %
```











Нейронні мережі

Лекція_03

* *

Слайди лекцій + інтерактивні ноутбуки Jupyter для хмари CPU/GPU/TPU Google Collaboratory:

https://cloud .comsys.kpi.ua/s/SMkBSsxRTazoTD6

* *

Лекція 03 - КАТЕГОРІЇ, ТИПИ, ПОХОДЖЕННЯ, РОЗВИТОК

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) у рамках створених в університеті:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.

https://kpi.ua/nvidia-info

ДЕМО 1
Версія CPU - класифікація цифр MNIST у TensorFlow 2.0
https://drive.google.com/file/d/18erMciUu3EN-0q8ZvkAGEkkeSrfbcviJ/view?usp=sharing

ДЕМО 2
Версія GPU – класифікація розрядів MNIST у TensorFlow 2.0
https://drive.google.com/file/d/1Px7CqFg4oxAUyp20tOBS7dF_fE0-t-oV/view?usp=sharing

ДЕМО 3
Версія TPU – класифікація цифр MNIST у TensorFlow 2.0
https://drive.google.com/file/d/1uokEko-10dgNW6JPImvDSwRrkxwFfl9b/view?usp=sharing

ДЕМО 4
Версія TPU – класифікація цифр MNIST у TensorFlow 2.0 https://drive.google.com/file/d/1EoGsHn_5qdLOyhQ6vQgivYSIjvBGyG5E/view?usp=sharing

НЕЙРОМЕРЕЖІ

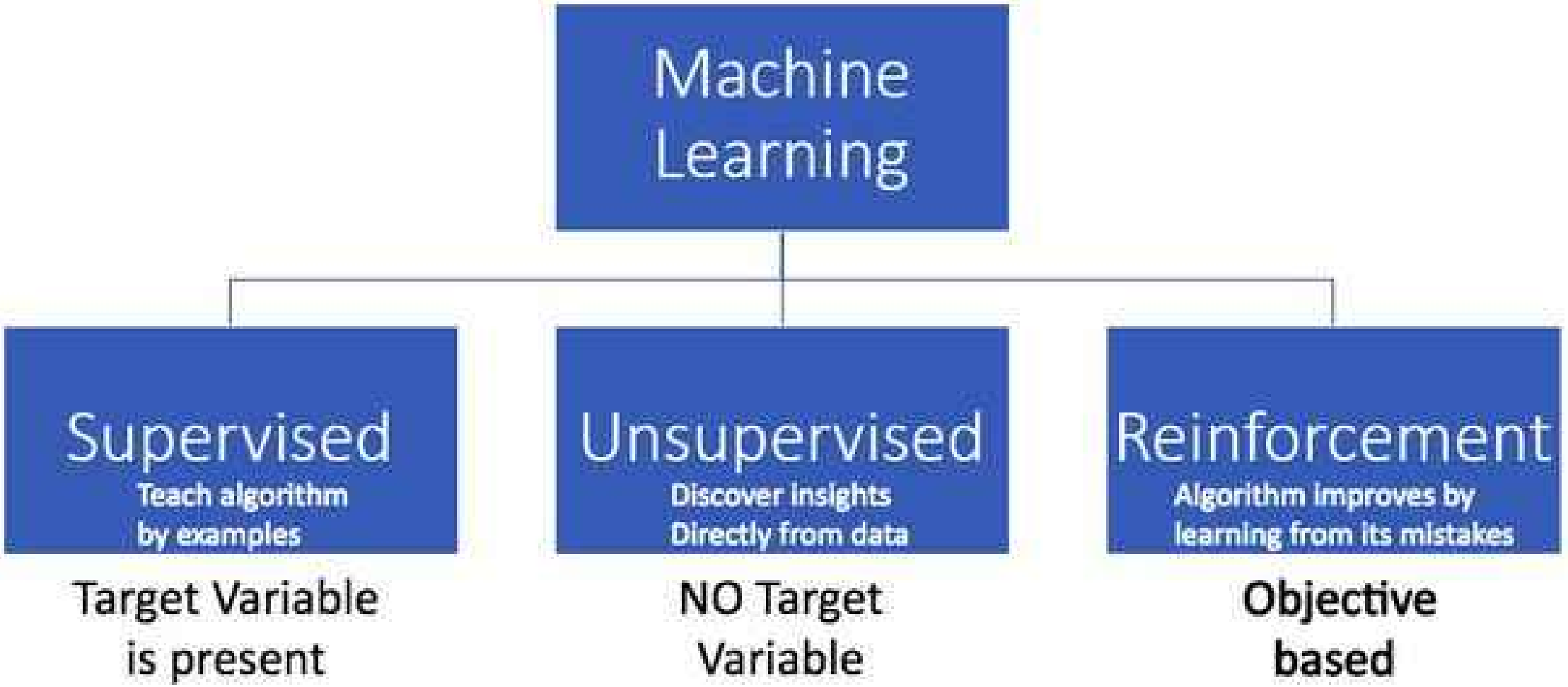
ЛЕКЦІЯ 3: КАТЕГОРІЇ, ТИПИ, ПОХОДЖЕННЯ, РОЗВИТОК

Юрій Гордієнко, DLI Certified Instructor

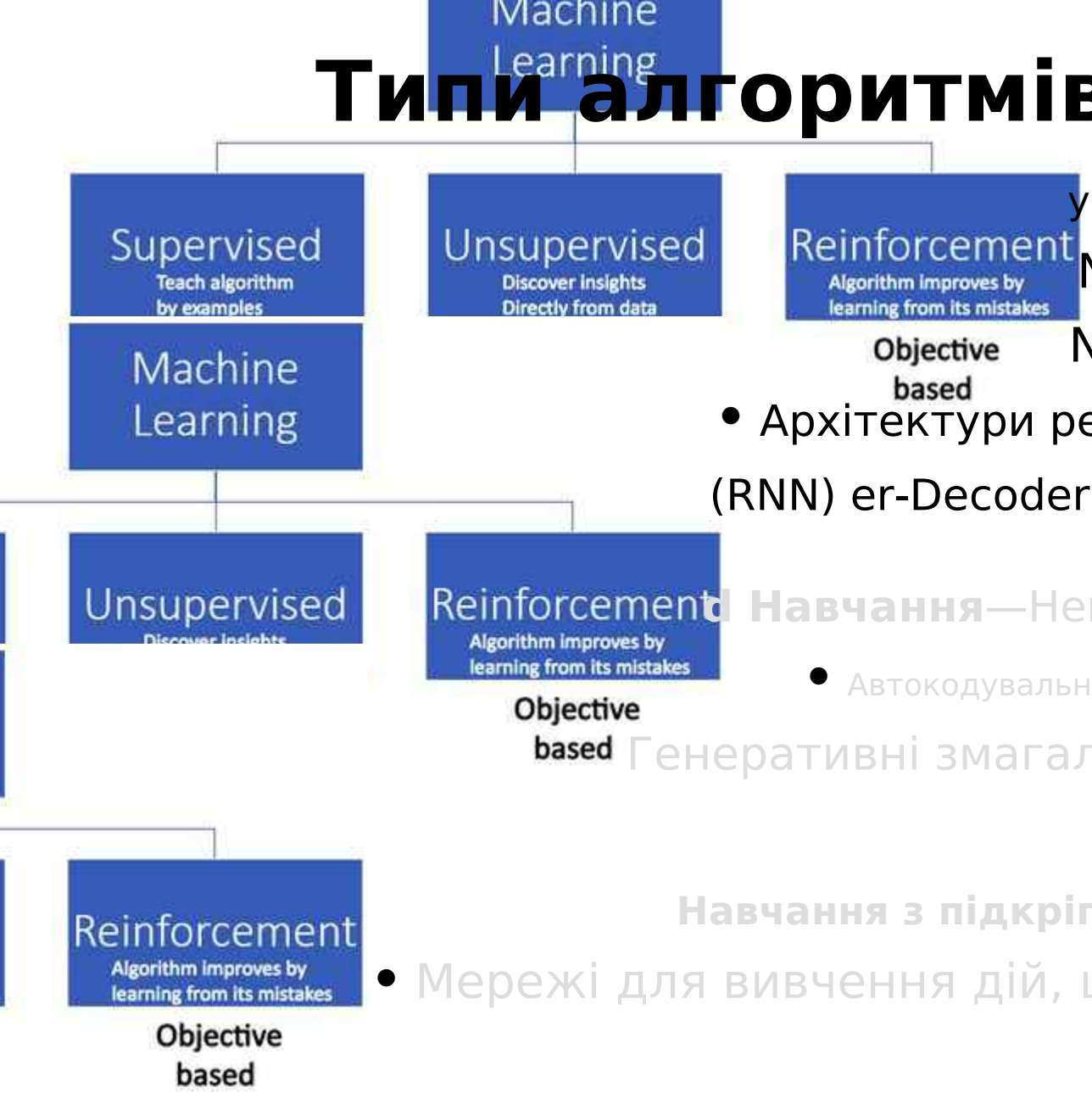
NVIDIA | DEEP LEARNING INSTITUTE

ВИДИ АЛГОРИТМІВ НАВЧАННЯ

Типи алгоритмів навчання



Типи алгоритмів навчання



уральські мережі (НН):

N (FNN)

N (CNN)

- Архітектури рекурентного NN

(RNN) er-Decoder (EDA)

d Навчання—Нейронні мережі (NN):

- Автокодувальник

Генеративні змагальні мережі

Навчання з підкріпленням

- Мережі для вивчення дій, цінностей і політики

(C) Лекс Фрідман

Типи алгоритмів навчання

уральські мережі (НН):

N (FNN)

N (CNN)

- Архітектури рекурентного NN

(RNN) er-Decoder (EDA)

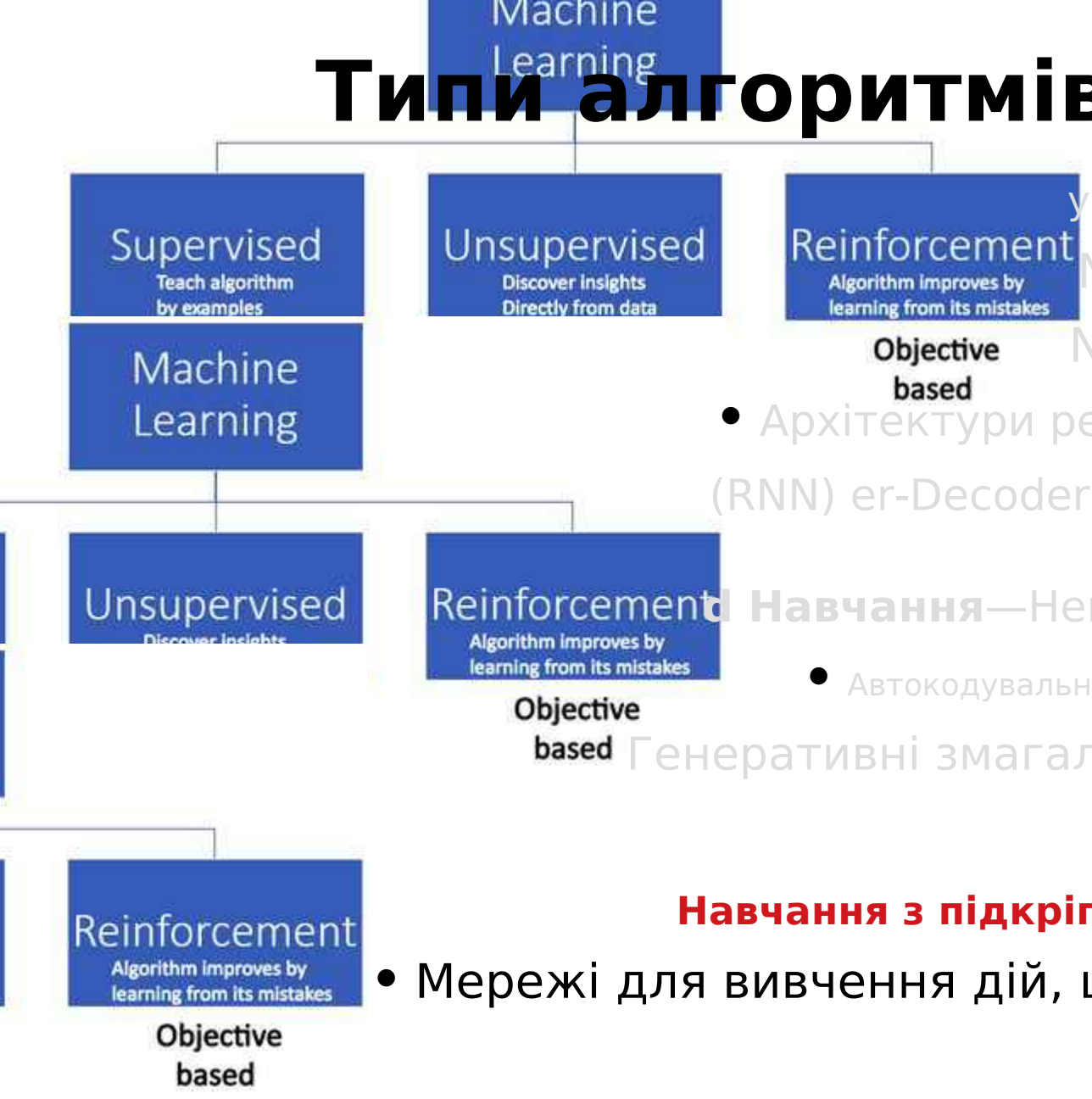
d Навчання—Нейронні мережі (NN):

- Автокодувальник
- Генеративні змагальні мережі

Навчання з підкріпленням

- Мережі для вивчення дій, цінностей і політики

Типи алгоритмів навчання



уральські мережі (НН):

N (FNN)

N (CNN)

- Архітектури рекурентного NN

(RNN) er-Decoder (EDA)

d Навчання—Нейронні мережі (NN):

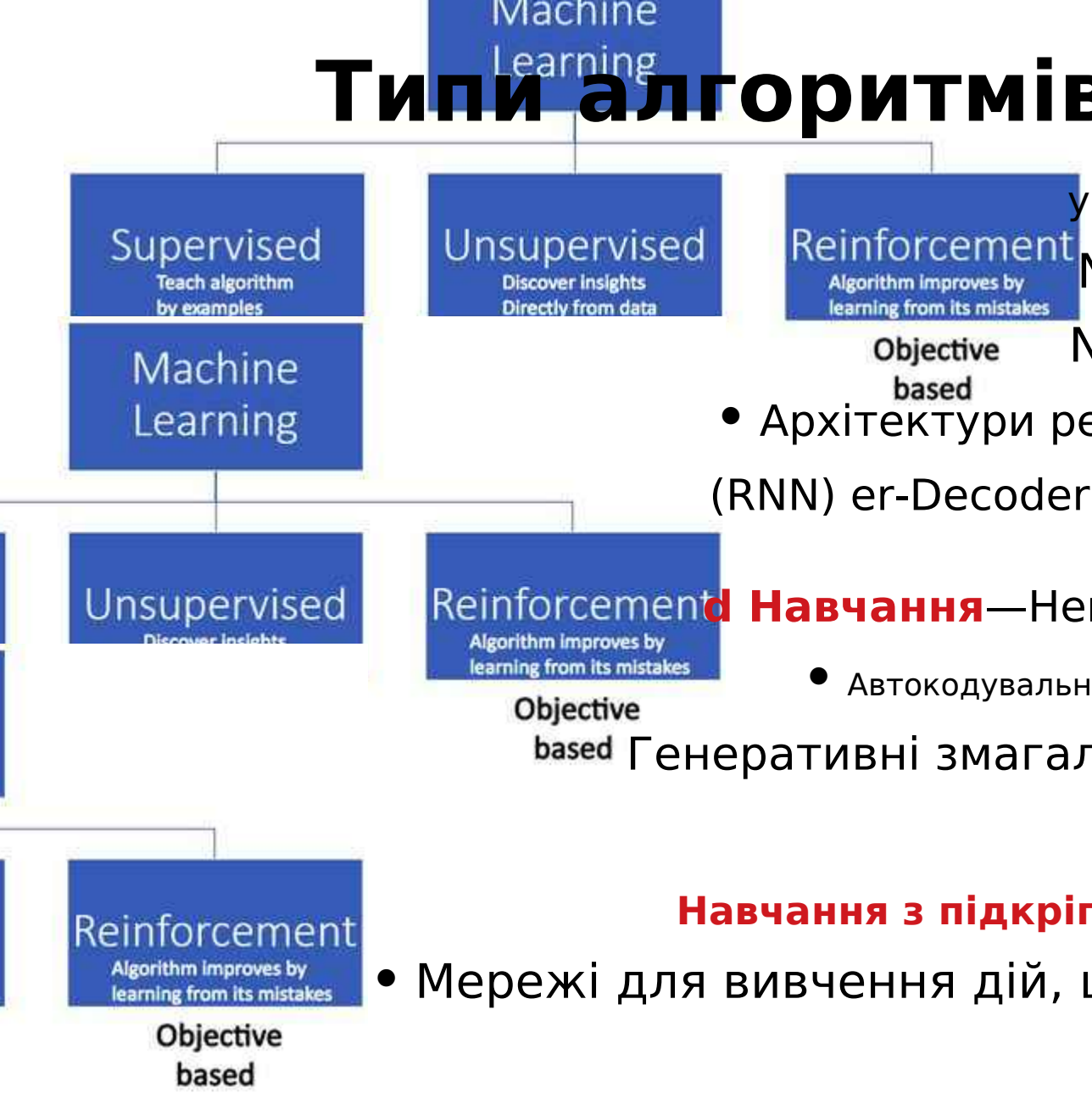
- Автокодувальник

Генеративні змагальні мережі

Навчання з підкріпленням

- Мережі для вивчення дій, цінностей і політики

Типи алгоритмів навчання



уральські мережі (НН):

N (FNN)

N (CNN)

- Архітектури рекурентного NN

(RNN) er-Decoder (EDA)

d Навчання—Нейронні мережі (NN):

- Автокодувальник

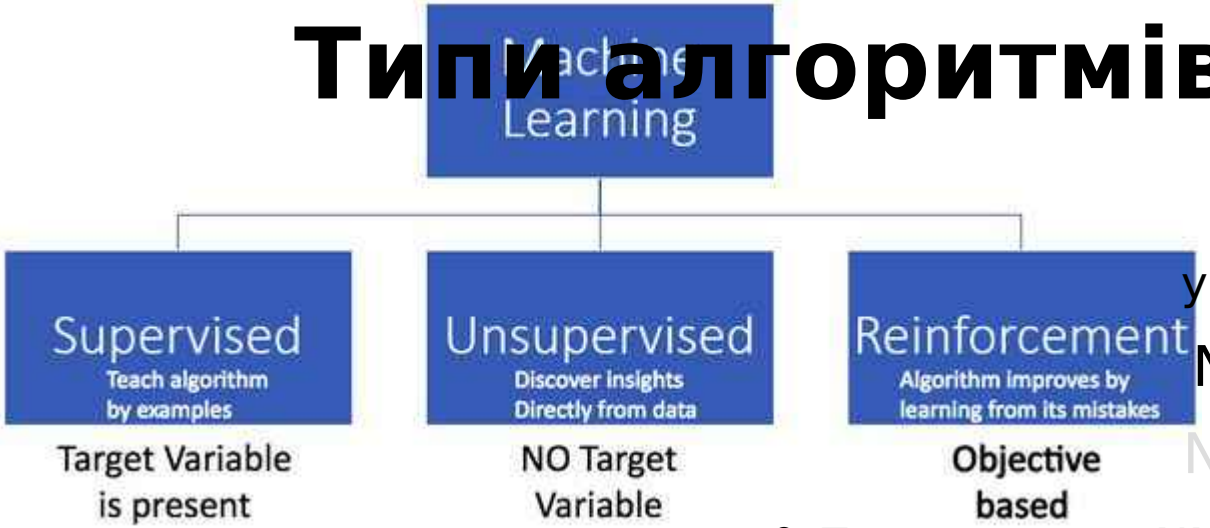
Генеративні змагальні мережі

Навчання з підкріпленням

- Мережі для вивчення дій, цінностей і політики

ВИДИ АЛГОРИТМІВ НАВЧАННЯ
-
АРХІТЕКТУРИ НЕЙРОМЕРЕЖ
-
ПІД НАГЛЯДОМ

Типи алгоритмів навчання



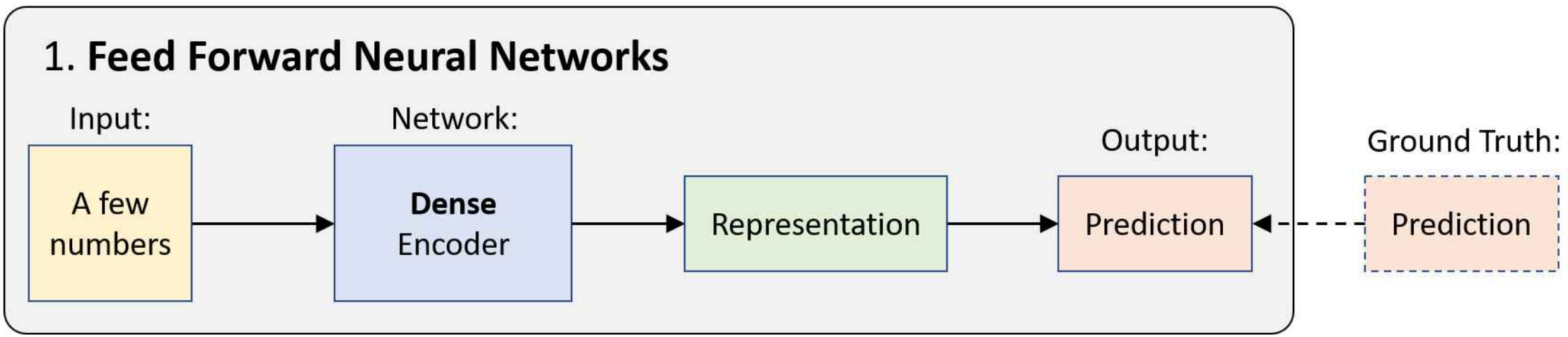
уральські мережі (НН):

N (FNN)

N (CNN)

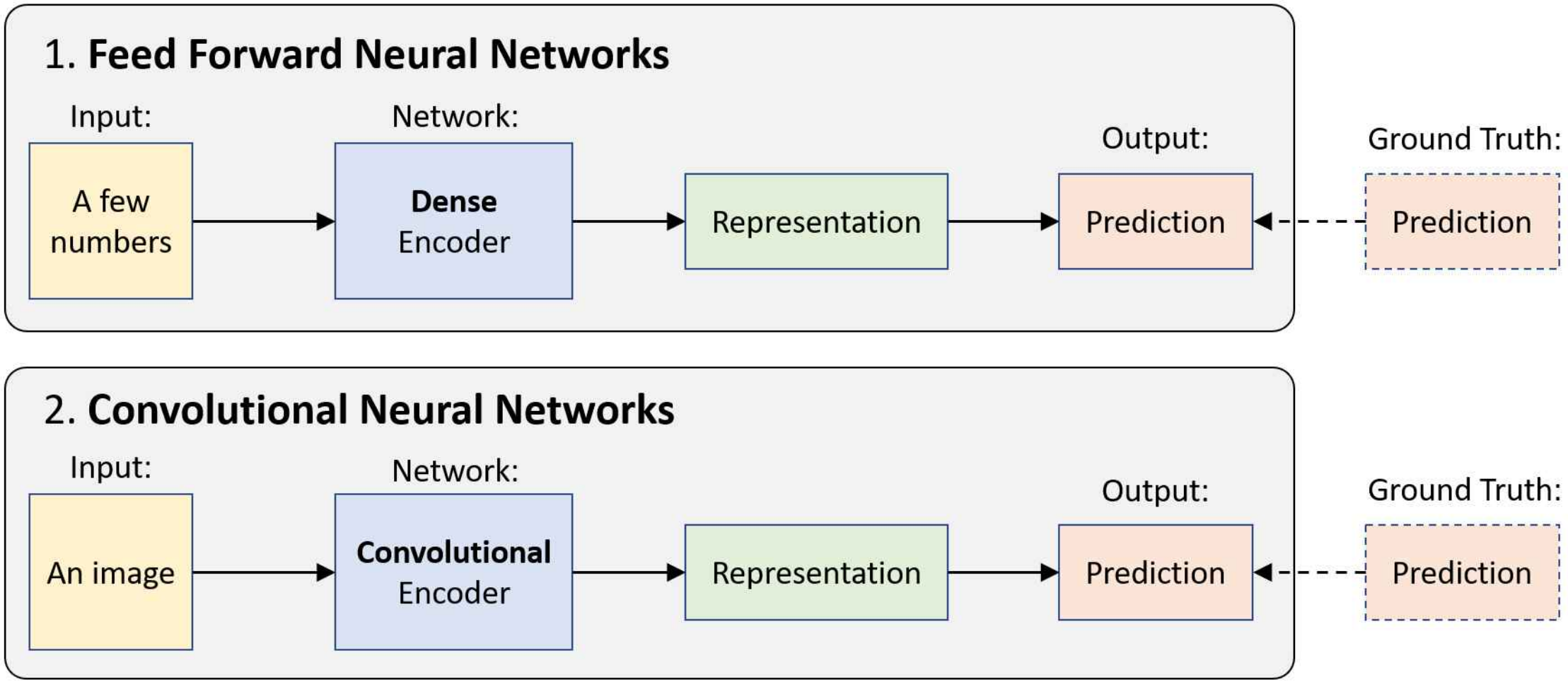
- Повторювана NN (RNN)

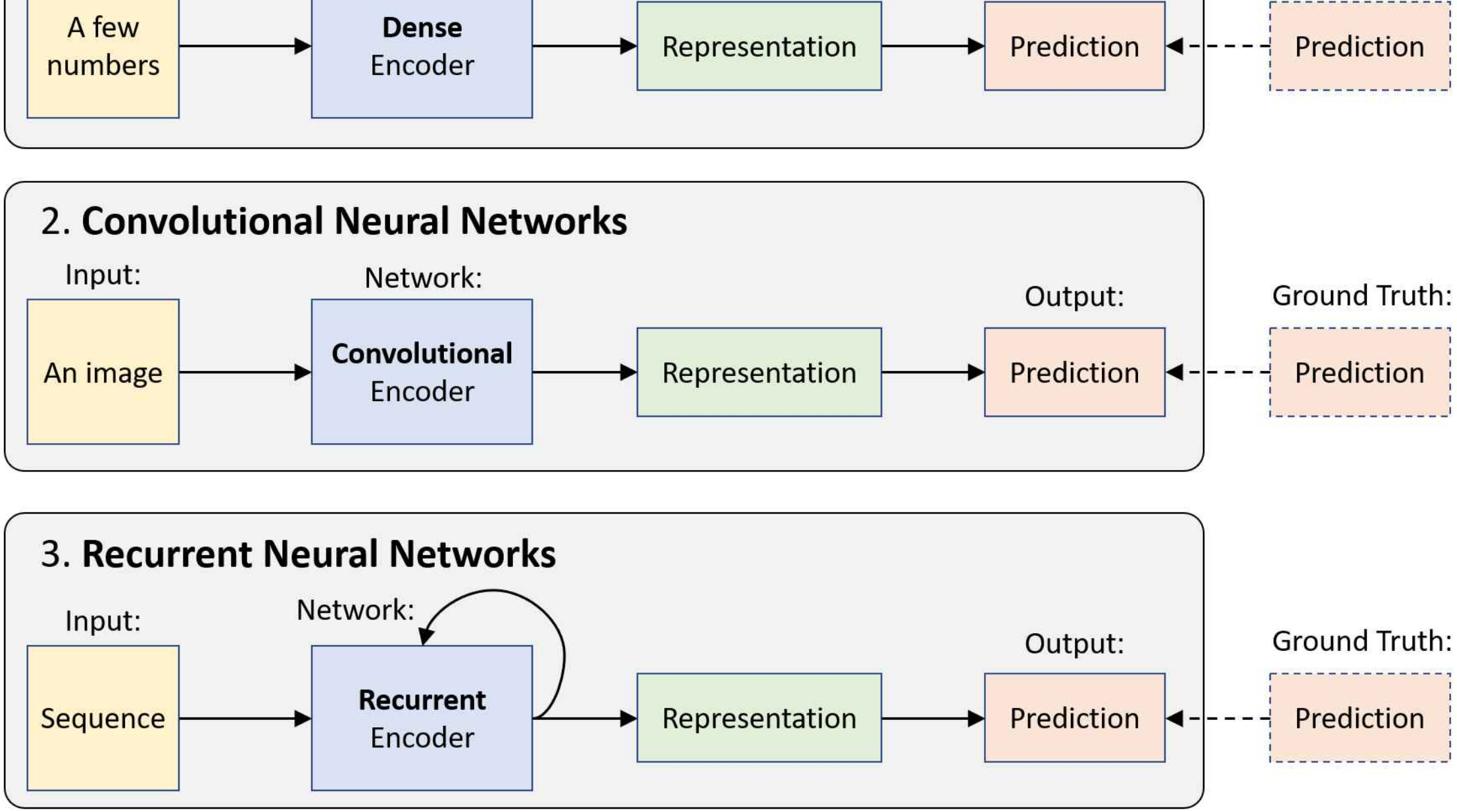
Supervised Learning

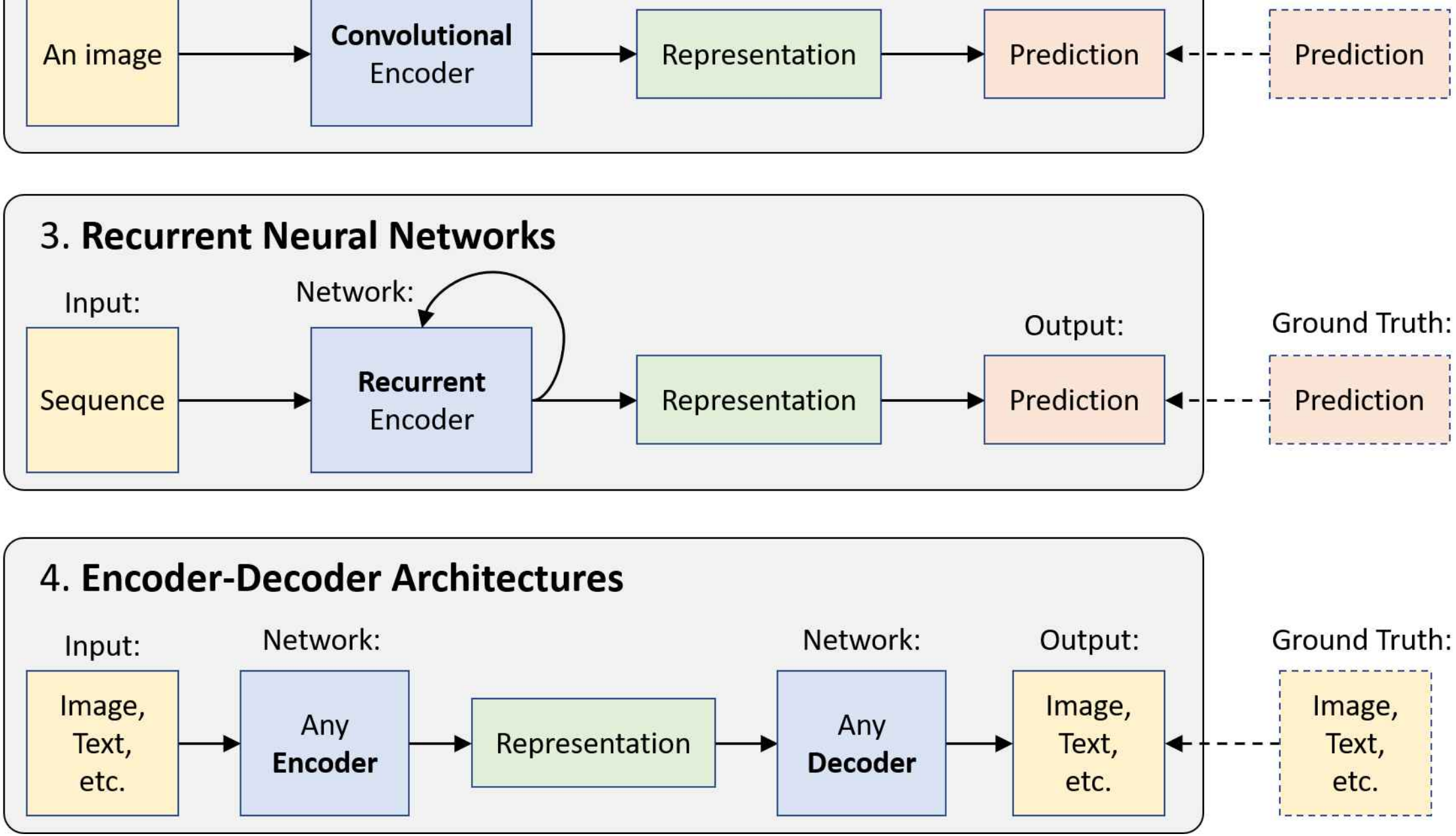


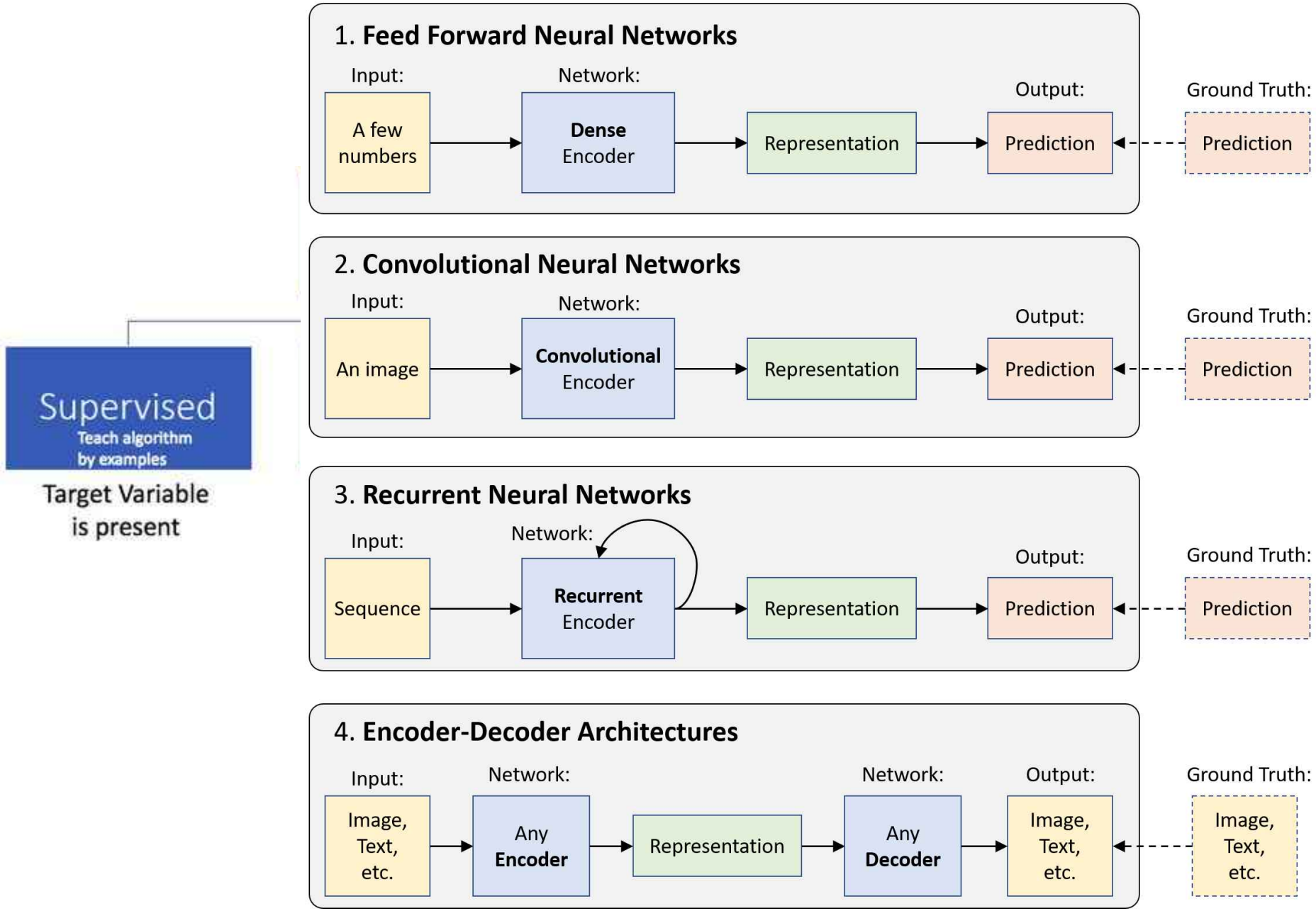
Machine

Supervised Learning



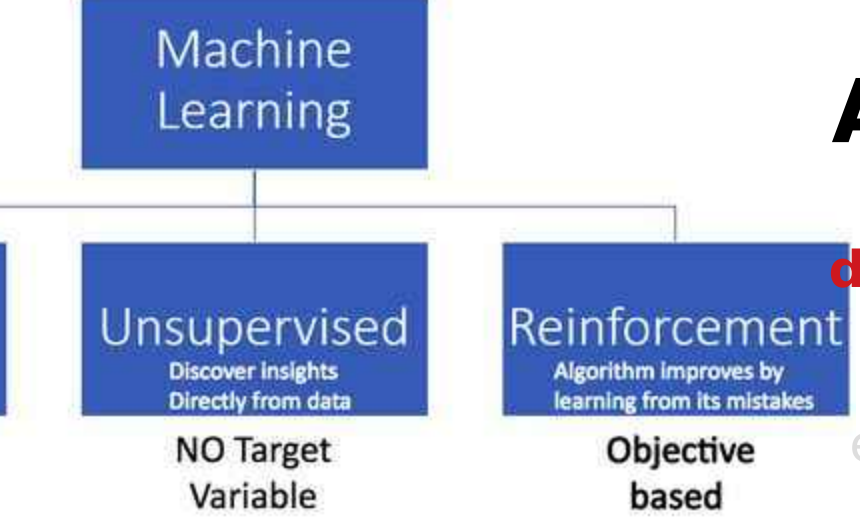

A few numbers
Dense Encoder
Representation
Prediction
Prediction
2. Convolutional Neural Networks
Input:
An image
Network:
Convolutional Encoder
Representation
Output:
Prediction
Ground Truth:
Prediction
3. Recurrent Neural Networks
Input:
Sequence
Network:
Recurrent Encoder
Representation
Output:
Prediction
Ground Truth:
Prediction


An image
Convolutional Encoder
Representation
Prediction
Prediction
3. Recurrent Neural Networks
Input:
Network:
Output:
Ground Truth:
Sequence
Recurrent Encoder
Representation
Prediction
Prediction
4. Encoder-Decoder Architectures
Input:
Network:
Network:
Output:
Ground Truth:
Image, Text, etc.
Any Encoder
Representation
Any Decoder
Image, Text, etc.
Image, Text, etc.



ВИДИ АЛГОРИТМІВ НАВЧАННЯ
-
АРХІТЕКТУРИ НЕЙРОМЕРЕЖ
-
БЕЗ НАГЛЯДУ

Алгоритми навчання

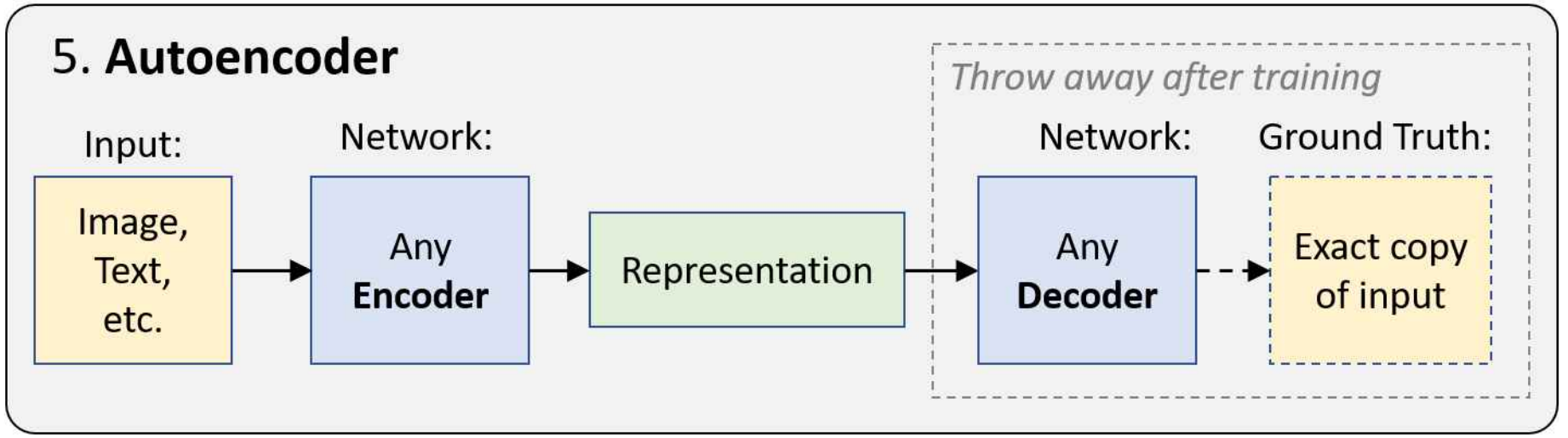


d Навчання—Нейронні мережі (NN):

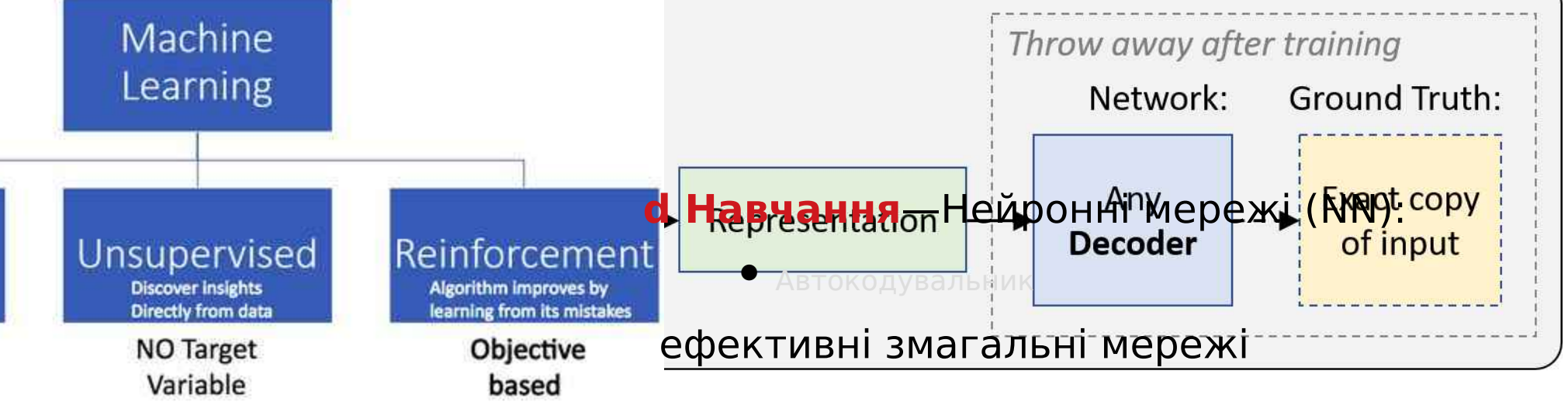
- Автокодувальник

ефективні змагальні мережі

Unsupervised Learning



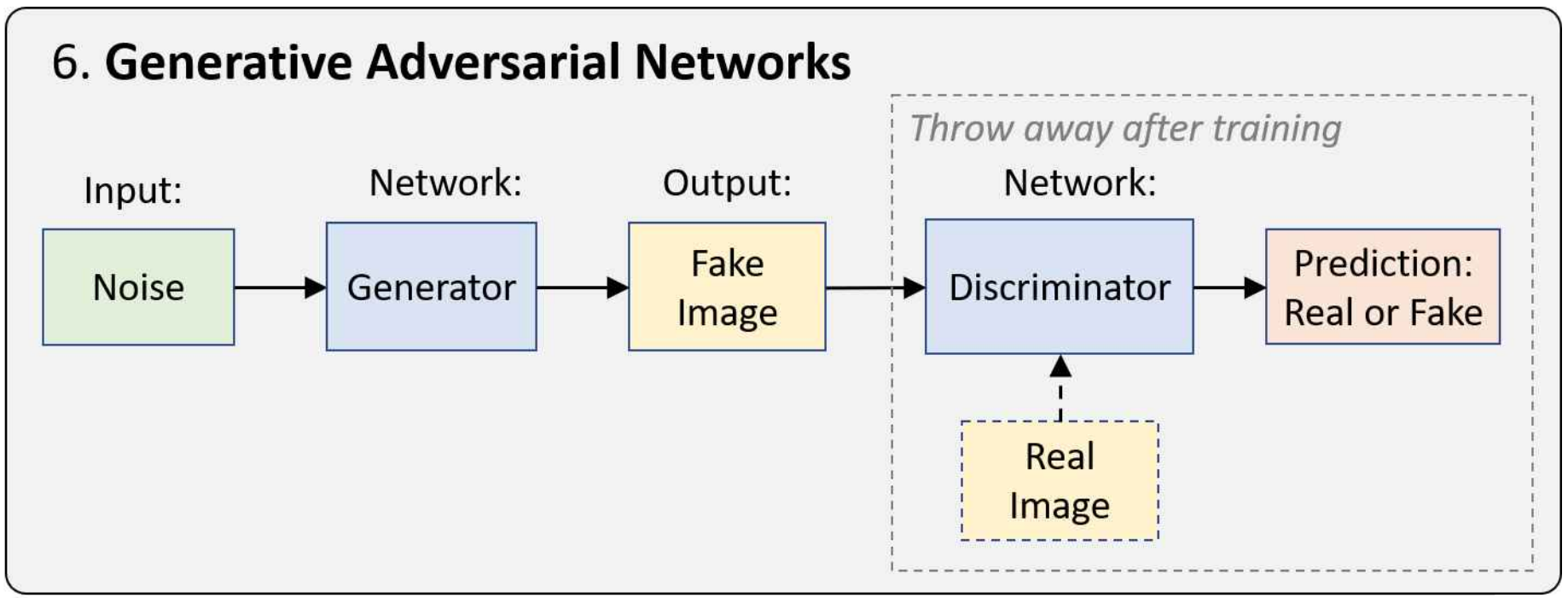
(C) Лекс Фрідман

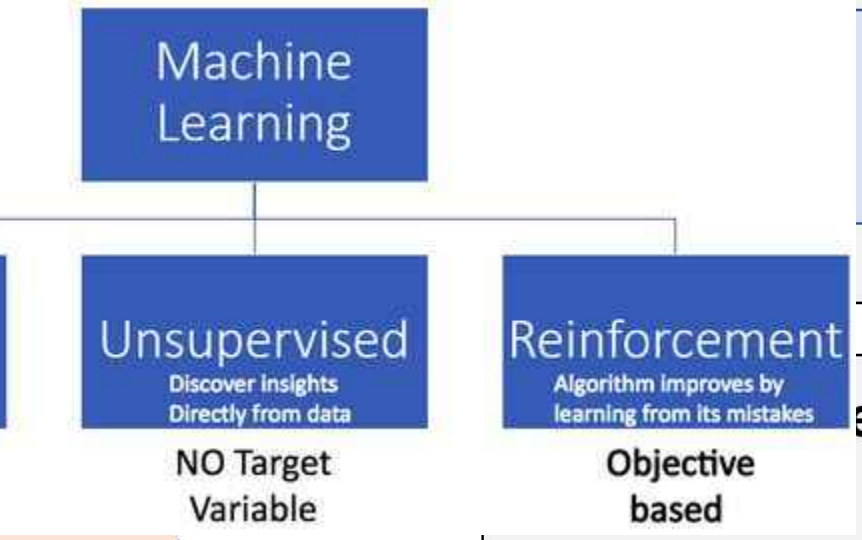


d. **Навчання** — Нейронні мережі (NN).

Автокодувальник

ефективні змагальні мережі





5. **Autoencoder**

Throw away after training

Input: | Network: | Network: | Ground Truth:

Any **Encoder** → Representation → Any **Decoder** → Exact copy of input

e Adversarial Networks

Throw away after training

Input: | Network: | Output: | Network:

Noise → Generator → Fake Image → Discriminator → Prediction: Real or Fake

Real Image → Discriminator

ВИДИ АЛГОРИТМІВ НАВЧАННЯ
-
АРХІТЕКТУРИ НЕЙРОМЕРЕЖ
-
АРМІРУВАННЯ

round Truth:

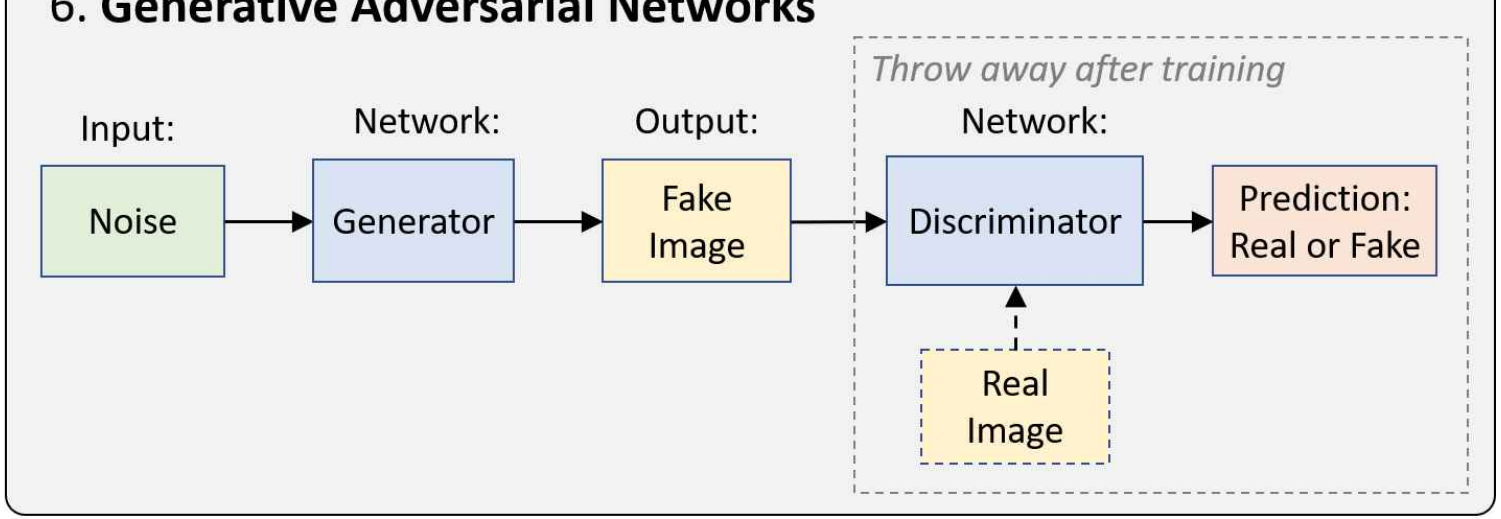
Prediction

round Truth:

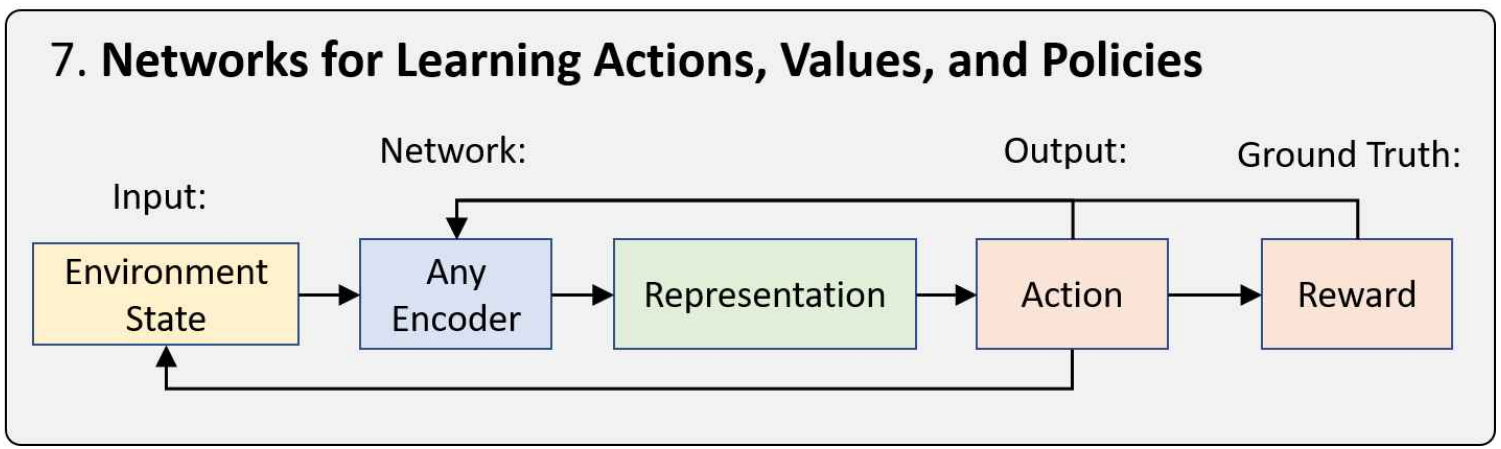
Prediction

round Truth:

Image,
Text,
etc.



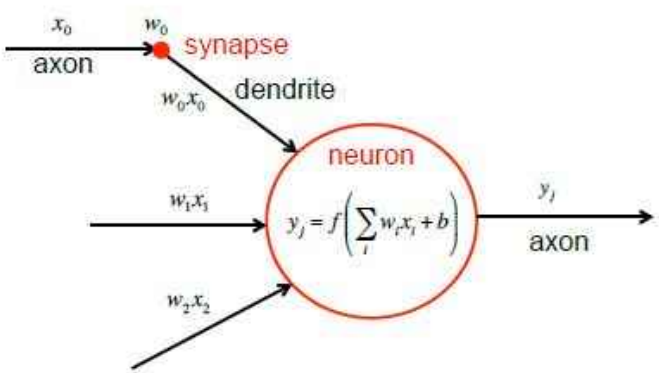
Reinforcement Learning

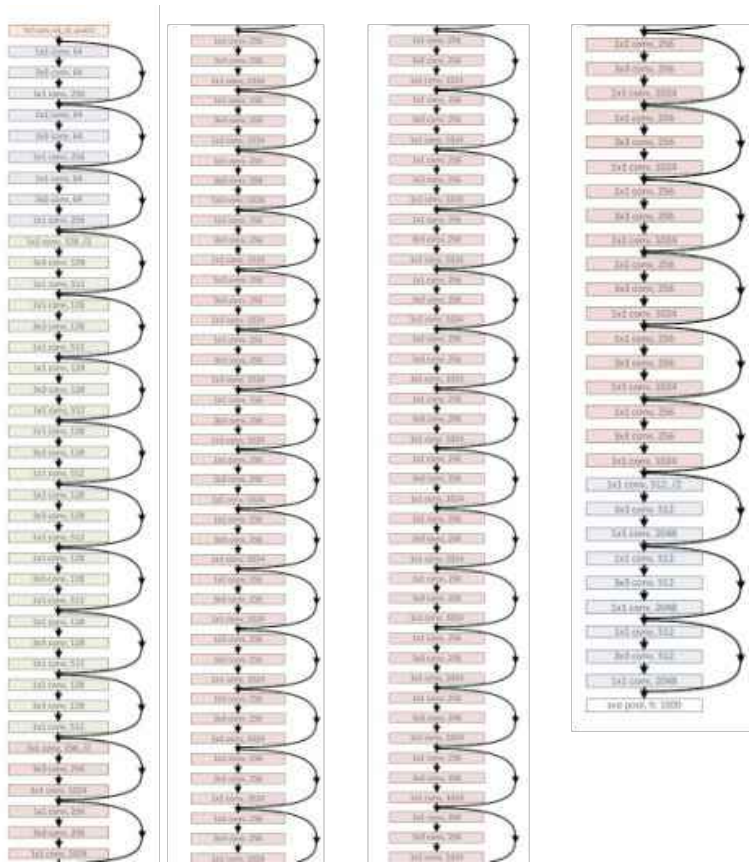


ЯК ВОНИ З'ЯВИЛИСЯ
-
МОТИВАЦІЯ

НЕЙРОМЕРЕЖІ НЕ НОВИНКА

І напрочуд простий як алгоритм





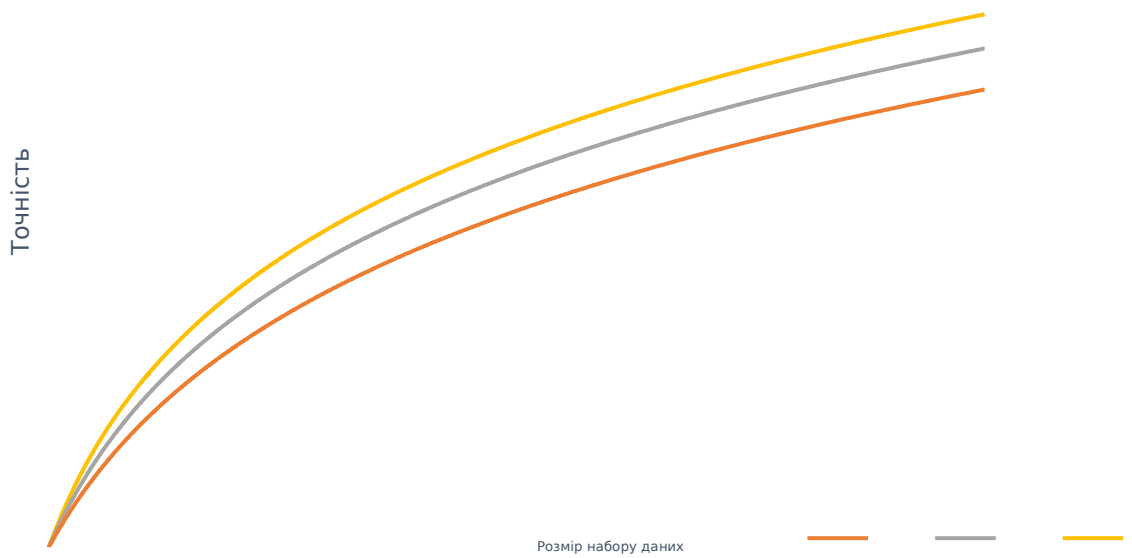
НЕЙРОМЕРЕЖІ НЕ НОВИНКА

Вони просто історично ніколи не працювали добре

Ендрю Нґ, «Гайки та болти застосування глибокого навчання», https://www.youtube.com/watch?v=F1ka6a13S9I

НЕЙРОМЕРЕЖІ НЕ НОВИНКА

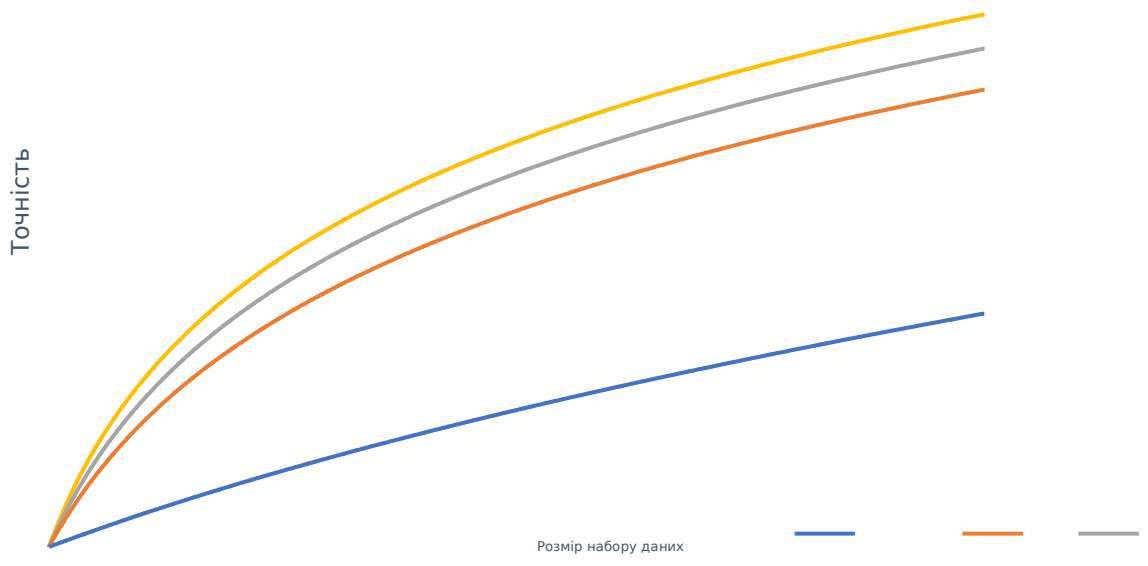
Вони просто історично ніколи не працювали добре



Ендрю Нґ, «Гайки та болти застосування глибокого навчання», https://www.youtube.com/watch?v=F1ka6a13S9I

НЕЙРОМЕРЕЖІ НЕ НОВИНКА

Вони просто історично ніколи не працювали добре

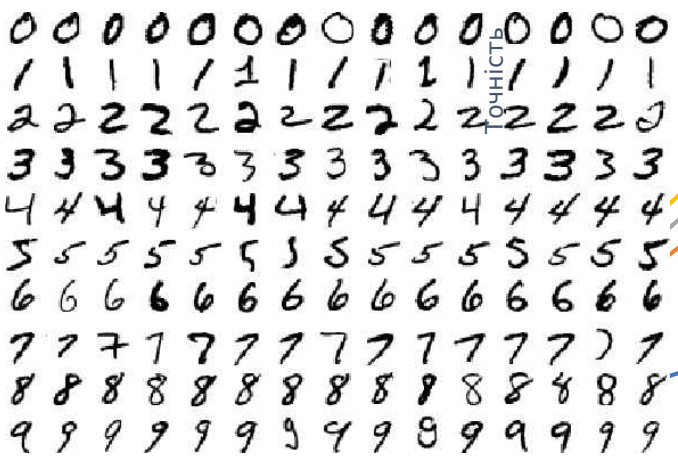


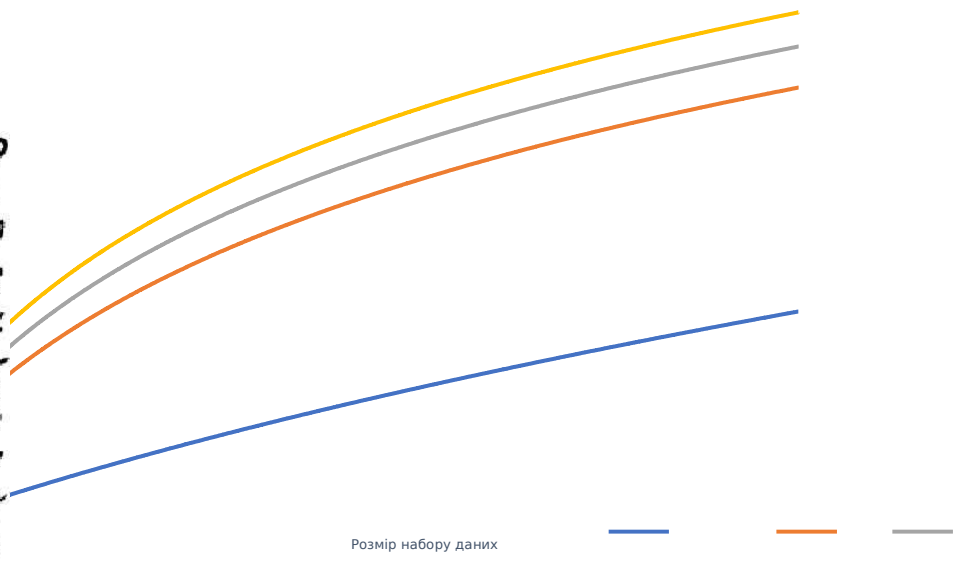
Ендрю Нґ, «Гайки та болти застосування глибокого навчання», https://www.youtube.com/watch?v=F1ka6a13S9I

НЕЙРОМЕРЕЖІ НЕ НОВИНКА

Історично у нас ніколи не було великих наборів даних чи комп’ютерів

База даних MNIST (1999) містить 60 000 навчальних зображень і 10 000 тестових зображень.

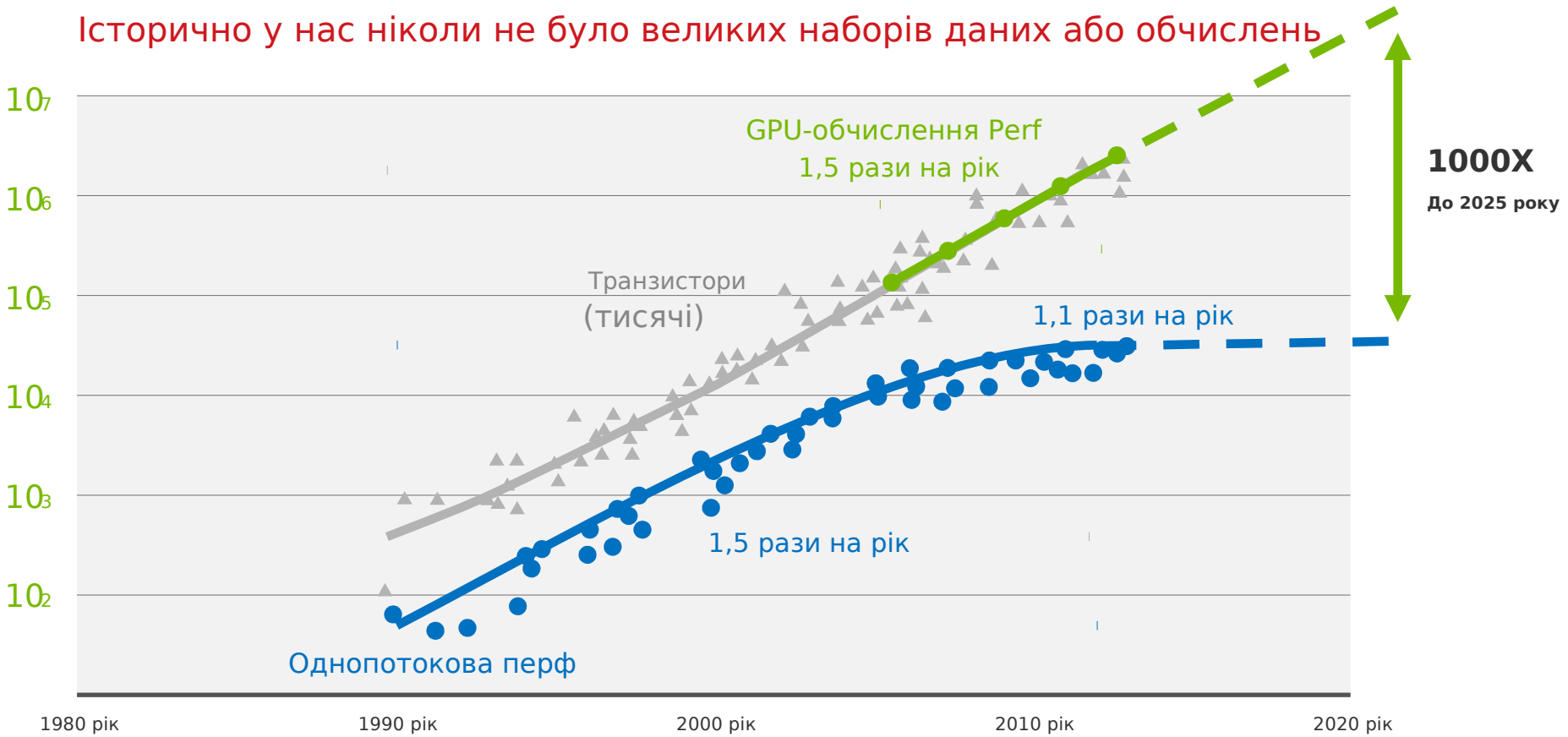




Ендрю Нґ, «Гайки та болти застосування глибокого навчання», https://www.youtube.com/watch?v=F1ka6a13S9I

ОБЧИСЛИТИ

Історично у нас ніколи не було великих наборів даних або обчислень



AI BIG BANG PILLARS:

- ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ

- ВЕЛИКІ ДАНІ

- МОДЕЛІ DEEP

AI BIG BANG PILLARS:

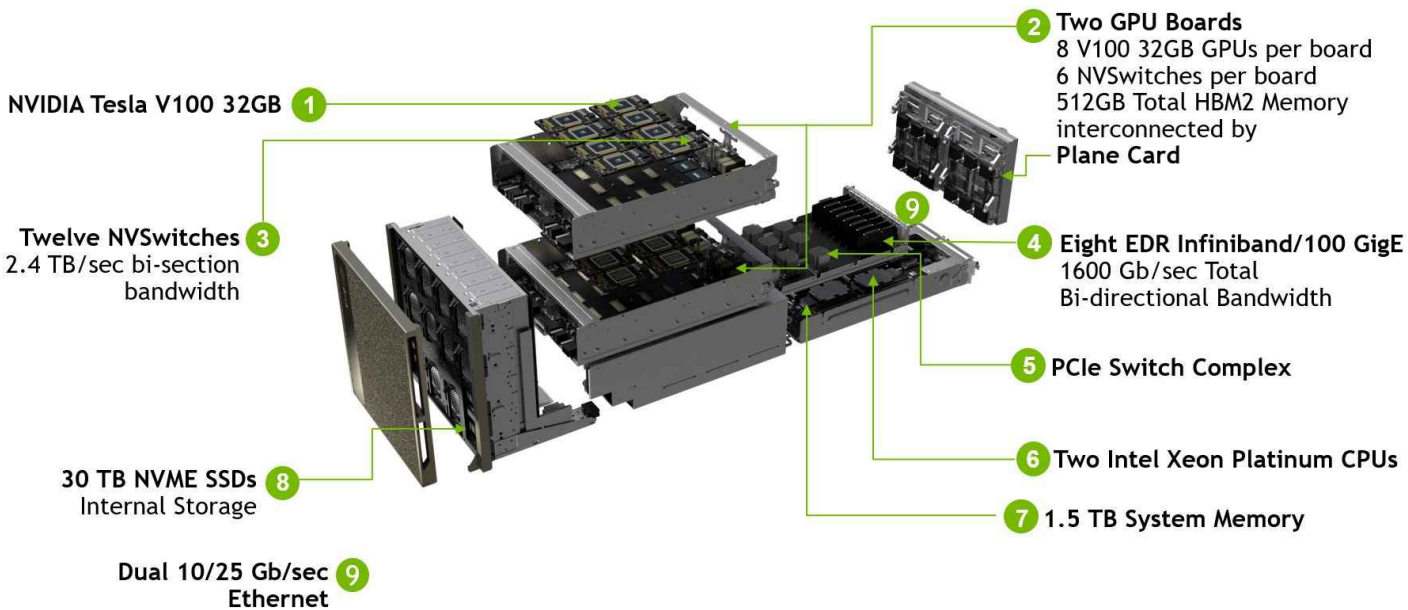
- ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ
- ВЕЛИКІ ДАНІ
- МОДЕЛІ DEEP

КОНТЕКСТ

1,759 petaFLOPs у листопаді2009 рік

КОНТЕКСТ

2 petaFLOPs -сьогодні



100 ЕКЗАФЛОПС

=

2 РОКИ НА ДВОЦЕСОРНОМУ СЕРВЕРІ

AI BIG BANG PILLARS:

- ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ

- ВЕЛИКІ ДАНІ

- МОДЕЛІ DEEP

РОЗШИРЕННЯ НАБОРОВ ДАНИХ

Логарифмічне співвідношення між розміром набору даних і точністю

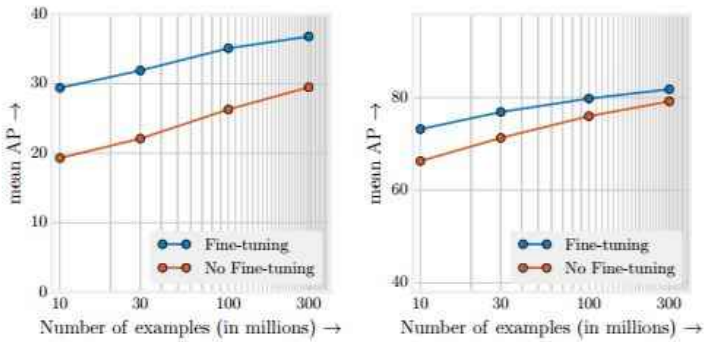


Figure 4. Object detection performance when initial checkpoints are pre-trained on different subsets of JFT-300M from scratch. x-axis is the data size in log-scale, y-axis is the detection performance in mAP@[.5,.95] on COCO minival* (left), and in mAP@.5 on PASCAL VOC 2007 test (right).

Initialization	mIOU
ImageNet	73.6
300M	75.3
ImageNet+300M	**76.5**

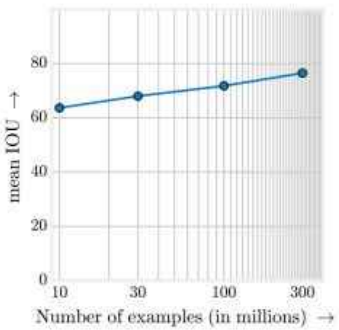
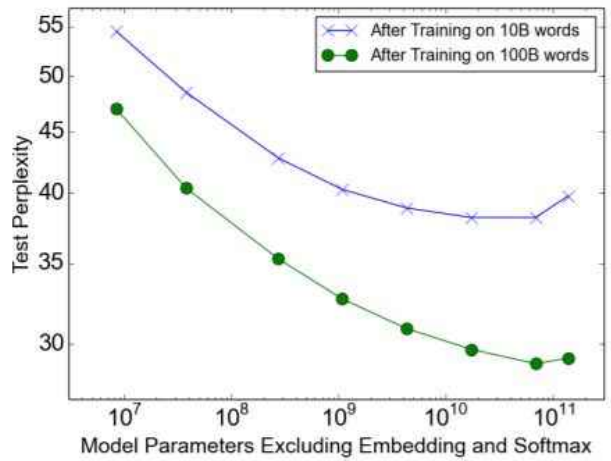


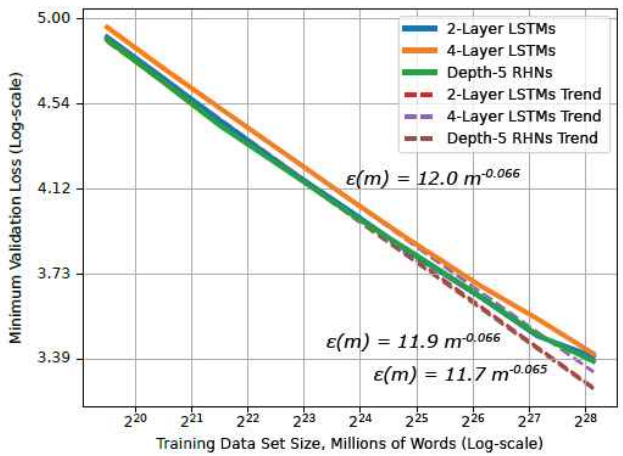
Figure 6. Semantic segmentation performance on Pascal VOC 2012 val set. (left) Quantitative performance of different initializations; (right) Impact of data size on performance.



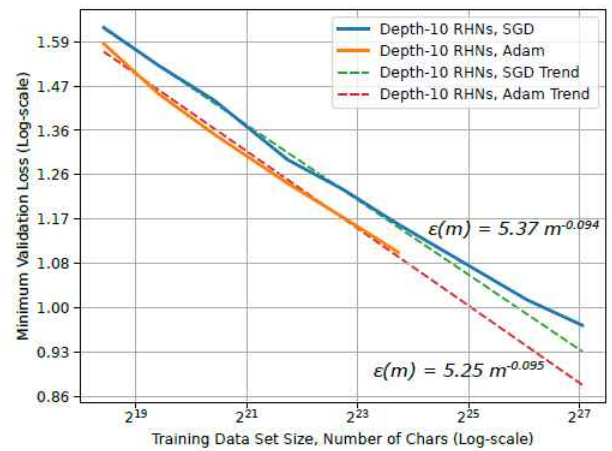
Сунь, Чен та ін. «Перегляд необґрунтованої ефективності даних в епоху глибокого навчання», препринт arXiv arXiv:1707.02968 (2017).
Шазір, Ноам та ін. "Надзвичайно великі нейронні мережі: шар змішаних експертів із розрідженим доступом". Препринт arXiv arXiv:1701.06538 (2017). Симонян, Карен та Ендрю Зіссерман. «Дуже глибокі згорткові мережі для розпізнавання великомасштабних зображень». Препринт arXiv arXiv:1409.1556 (2014).

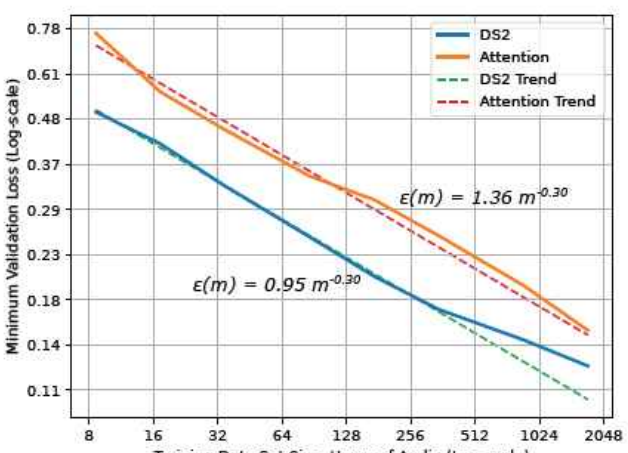
РОЗШИРЕННЯ НАБОРОВ ДАНИХ

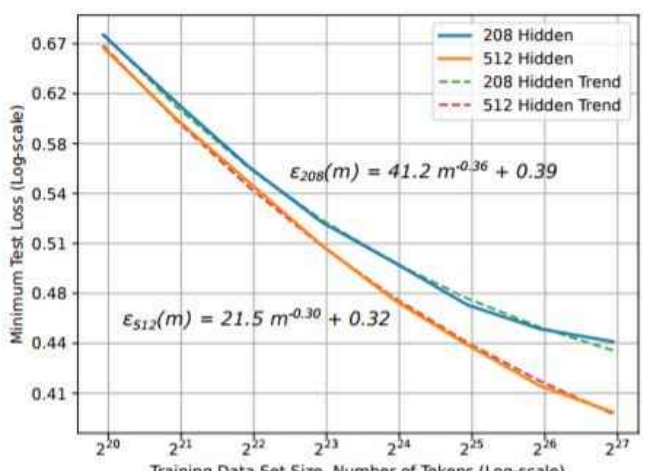
Логарифмічне співвідношення між розміром набору даних і точністю

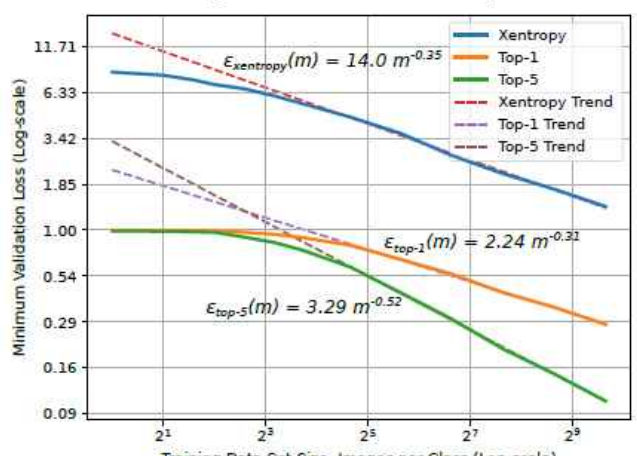


- Переклад
- Мовні моделі
- Моделі мови символів
- Класифікація зображень
- Моделі мовлення уваги





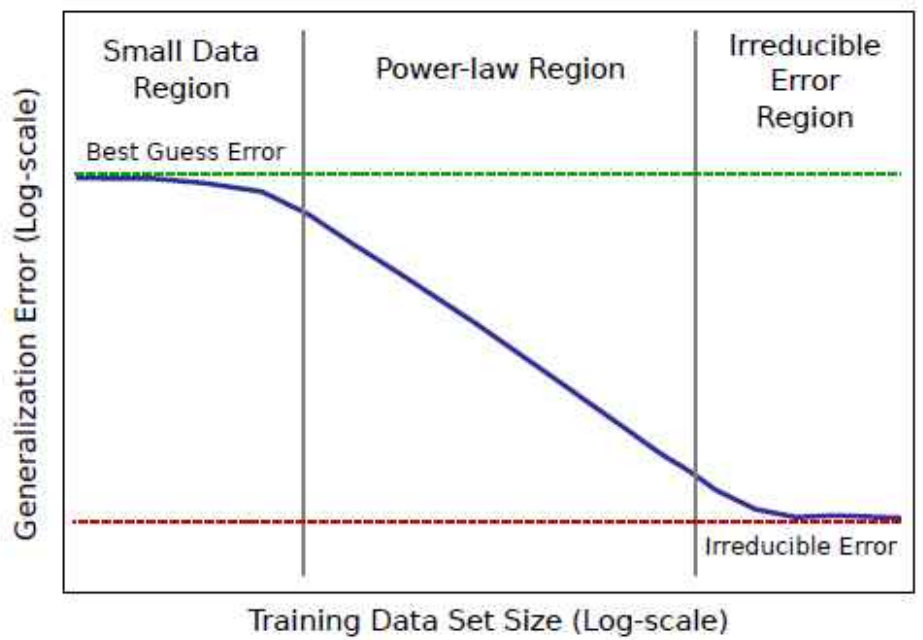




Хестнесс, Дж., Наранг, С., Ардалані, Н., Діамос, Г., Джун, Х., Кіанінеджад, Х., ... та Чжоу, Ю. (2017). Масштабування глибокого навчання є передбачуваним емпіричним шляхом. препринт arXiv arXiv:1712.00409.

РОЗШИРЕННЯ НАБОРОВ ДАНИХ

Логарифмічне співвідношення між розміром набору даних і точністю



Хестнесс, Дж., Наранг, С., Ардалані, Н., Діамос, Г., Джун, Х., Кіанінеджад, Х., ... та Чжоу, Ю. (2017). Масштабування глибокого навчання є передбачуваним емпіричним шляхом. препринт arXiv arXiv:1712.00409.

AI BIG BANG PILLARS:

- ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ

- ВЕЛИКІ ДАНІ

- МОДЕЛІ DEEP

СКЛАДНІСТЬ НЕЙРОМЕРЕЖІ ВИБУХАЄ

Щоб вирішувати дедалі складніші виклики

7 ExaFLOPS

2015 – Microsoft ResNet
Superhuman Image Recognition

20 ExaFLOPS

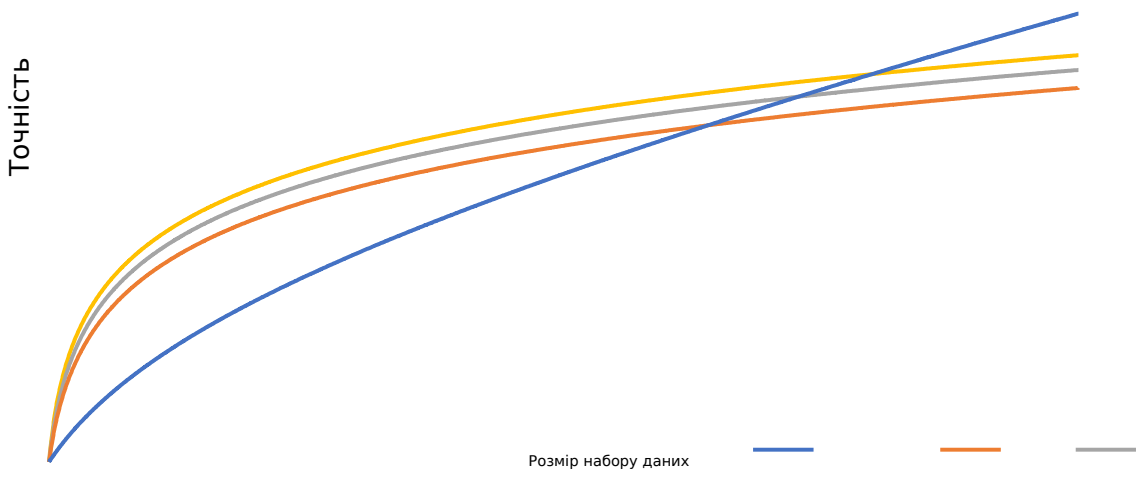
2016 – Baidu Deep Speech 2
Надлюдське розпізнавання голосу

100 ExaFLOPS
8700 мільйонів параметрів

2017 – Google Neural Machine Translation
Переклад майже людською мовою

НЕЙРОМЕРЕЖІ НЕ НОВИНКА

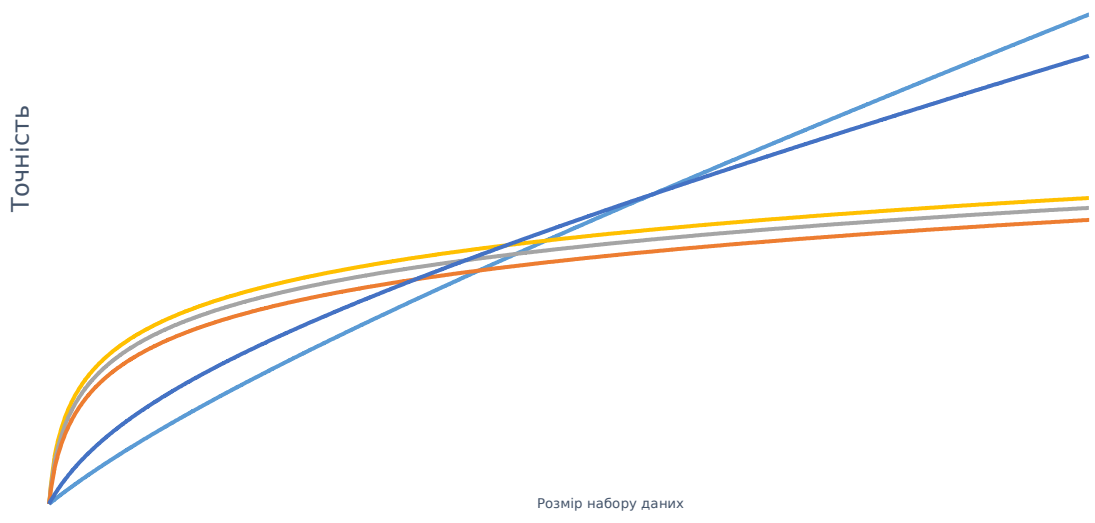
Але це змінило спосіб машинного навчання



Ендрю Нґ, «Гайки та болти застосування глибокого навчання», https://www.youtube.com/watch?v=F1ka6a13S9I

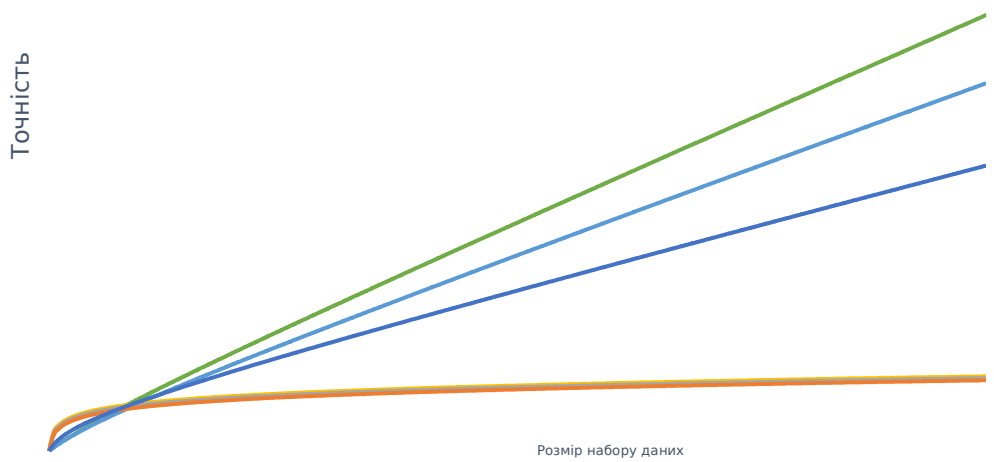
НЕЙРОМЕРЕЖІ НЕ НОВИНКА

Дані та розмір моделі – ключ до точності



НЕЙРОМЕРЕЖІ НЕ НОВИНКА

Перевищення продуктивності людського рівня



Ендрю Нґ, «Гайки та болти застосування глибокого навчання», https://www.youtube.com/watch?v=F1ka6a13S9I

СКЛАДНІСТЬ НЕЙРОМЕРЕЖІ ВИБУХАЄ

Щоб вирішувати дедалі складніші виклики

7 ExaFLOPS

2015 – Microsoft ResNet
Superhuman Image Recognition

20 ExaFLOPS

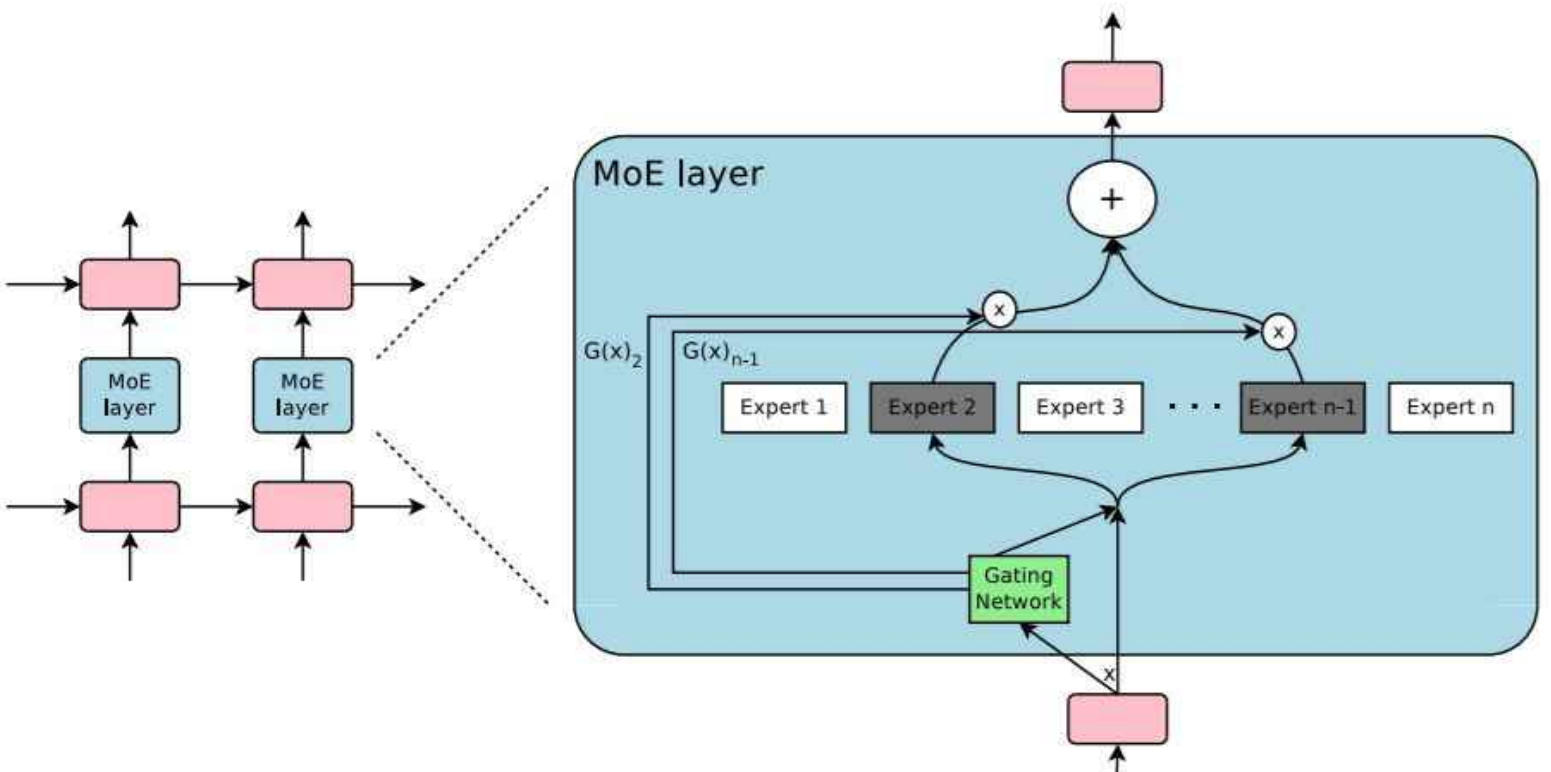
2016 – Baidu Deep Speech 2
Надлюдське розпізнавання голосу

100 ExaFLOPS
8700 мільйонів параметрів

2017 – Google Neural Machine Translation
Переклад майже людською мовою

РОЗВИШЕННЯ СКЛАДНОСТІ МОДЕЛІ

Можливі більші моделі

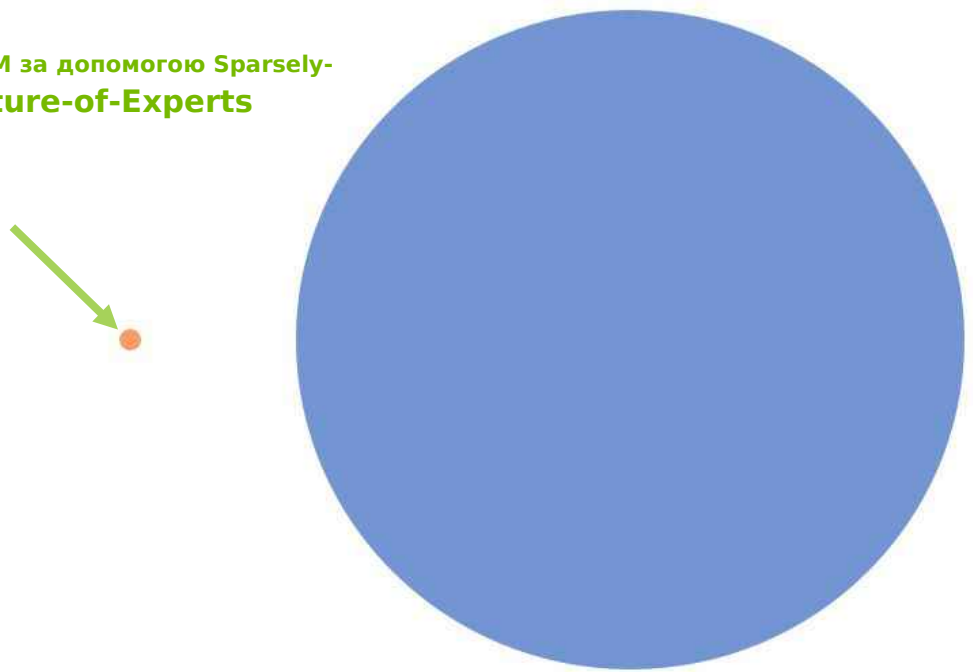


Шазір, Ноам та ін. "Надзвичайно великі нейронні мережі: шар змішаних експертів із розрідженим доступом". Препринт arXiv arXiv:1701.06538 (2017). Симонян, Карен та Ендрю Зіссерман. «Дуже глибокі згорткові мережі для розпізнавання великомасштабних зображень». Препринт arXiv arXiv:1409.1556 (2014). Ключові моменти застосування глибокого навчання, Ендрю Нг, 2016 р. - https://youtu.be/F1ka6a13S9I

РОЗВИШЕННЯ СКЛАДНОСТІ МОДЕЛІ

«Надзвичайно великі нейронні мережі» – розмір має значення

VGG 19 проти Google LSTM за допомогою Sparsely-
Шар Gated Mixture-of-Experts



Шазір, Ноам та ін. "Надзвичайно великі нейронні мережі: шар змішаних експертів із розрідженим доступом". Препринт arXiv arXiv:1701.06538 (2017). Симонян, Карен та Ендрю Зіссерман. «Дуже глибокі згорткові мережі для розпізнавання великомасштабних зображень». Препринт arXiv arXiv:1409.1556 (2014). Ключові моменти застосування глибокого навчання, Ендрю Нг, 2016 р. - https://youtu.be/F1ka6a13S9I

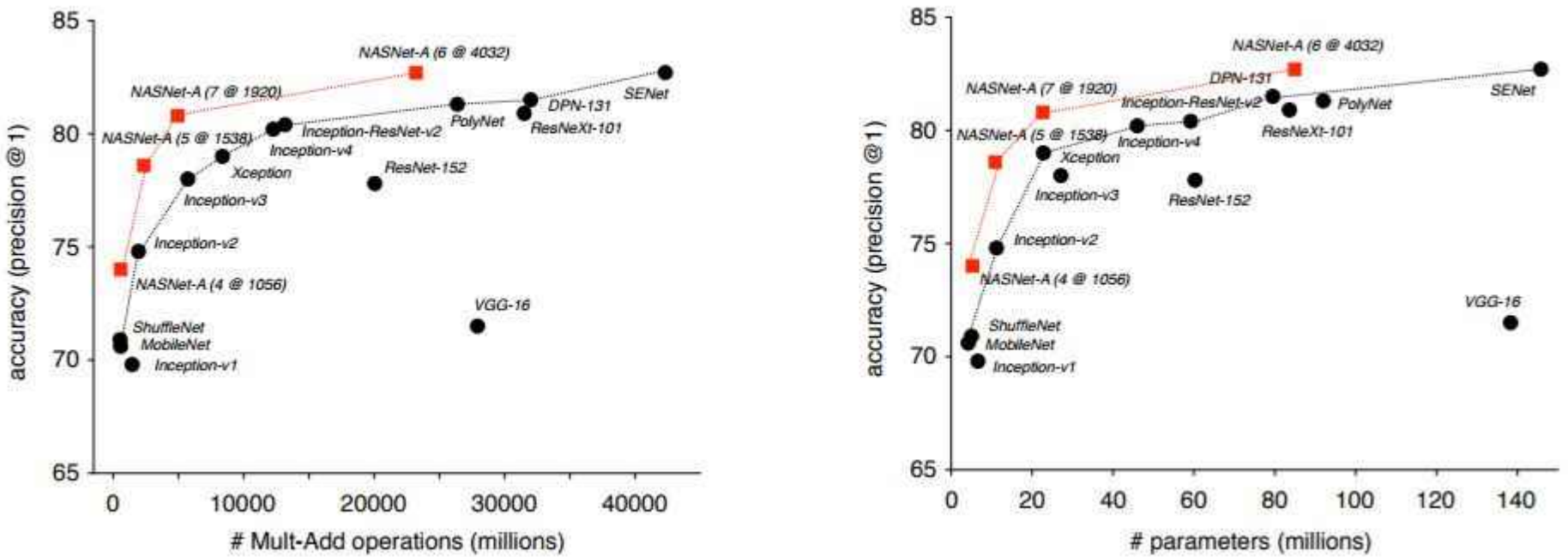
РОЗВИШЕННЯ СКЛАДНОСТІ МОДЕЛІ

Хороші новини – розмір моделі сублінійно масштабується

Хестнесс, Дж., Наранг, С., Ардалані, Н., Діамос, Г., Джун, Х., Кіанінеджад, Х., ... та Чжоу, Ю. (2017). Масштабування глибокого навчання є передбачуваним емпіричним шляхом. препринт arXiv arXiv:1712.00409.

ДОКАЗИ ОБРОБКИ ЗОБРАЖЕНЬ

Хороші новини – розмір моделі сублінійно масштабується



Zoph, Barret та ін. «Вивчення переносних архітектур для масштабованого розпізнавання зображень». Препринт arXiv arXiv:1707.07012 (2017).

НАСЛІДКИ

Полегшення складних завдань

Створення нерозв'язних проблем

дорогий

«Для будь-якого розміру даних це а гарна ідея завжди робити так, щоб дані виглядали маленькими, використовуючи величезну модель».

Джеффрі Хінтон

ФУНДАМЕНТАЛЬНІ ЗМІНИ В ЕКОНОМІЦІ

Вплив

China's Got a Huge Artificial Intelligence Plan

Bloomberg News
July 21, 2017, 4:04 AM GMT+1 *Updated on* July 21, 2017, 8:12 AM GMT+1

- Priorities are intelligent robotics, vehicles, virtual reality
- AI seen contributing up to $15.7 trillion worldwide by 2030

Microsoft just officially listed AI as one of its top priorities, replacing mobile

- Satya Nadella's "mobile-first and cloud-first world" line is out.
- The change comes after Microsoft formed the Artificial Intelligence and Research group.

Jordan Novet | @jordannovet
Published 5:48 PM ET Wed, 2 Aug 2017 | Updated 7:00 PM ET Fri, 4 Aug 2017
CNBC

BBC Menu
NEWS | PIDGIN
Home | Nigeria | Africa | World | Video | Audio | Sport | Entertainme

UAE: First minister of artificial intelligence don land

19 October 2017

REUTERS World Business Markets Politics TV

SCIENCE NEWS MARCH 29, 2018 / 2:36 PM / 10 DAYS AGO

France to spend $1.8 billion on AI to compete with U.S., China

Mathieu Rosemain, Michel Rose

EUROPEAN COMMISSION
Press Release Database
European Commission > Press releases database > Press Release details

European Commission - Press release

Commission proposes to invest EUR 1 billion in world-class European supercomputers

Brussels, 11 January 2018

Andrus **Ansip**, European Commission Vice-President for the Digital Single Market, said: *"Supercomputers are the engine to power the digital economy. It is a tough race and today the EU is lagging behind: we do not have any supercomputers in the world's top-ten. With the EuroHPC initiative we want to give European researchers and companies world-leading supercomputer capacity by 2020 – to develop technologies such as artificial intelligence and build the future's everyday applications in areas like health, security or engineering."*

TechRepublic.

INNOVATION

The 10 tech companies that have invested the most money in AI

Of the tech giants, Google is the biggest investor in AI by billions.

By Olivia Krauth | January 12, 2018, 1:12 PM PST

1. Google - $3.9 billion
2. Amazon - $871 million
3. Apple - $786 million
4. Intel - $776 million
5. Microsoft - $690 million
6. Uber - $680 million
7. Twitter - $629 million
8. AOL - $191.7 million
9. Facebook - $60 million
10. Salesforce - $32.8 million

NVIDIA DEEP LEARNING INSTITUTE

НАСЛІДКИ

Хороші і погані новини

- Хороша новина: вимоги передбачувані.
  - Ми можемо передбачити, скільки даних нам знадобиться
  - Ми можемо передбачити, скільки обчислювальної потужності нам знадобиться
- The**погано**новини: значення можуть бути значними.

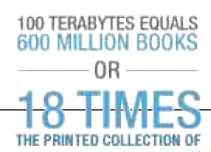
НАСЛІДКИ

Експериментальний характер глибокого навчання – неприйнятний час навчання

НАСЛІДКИ

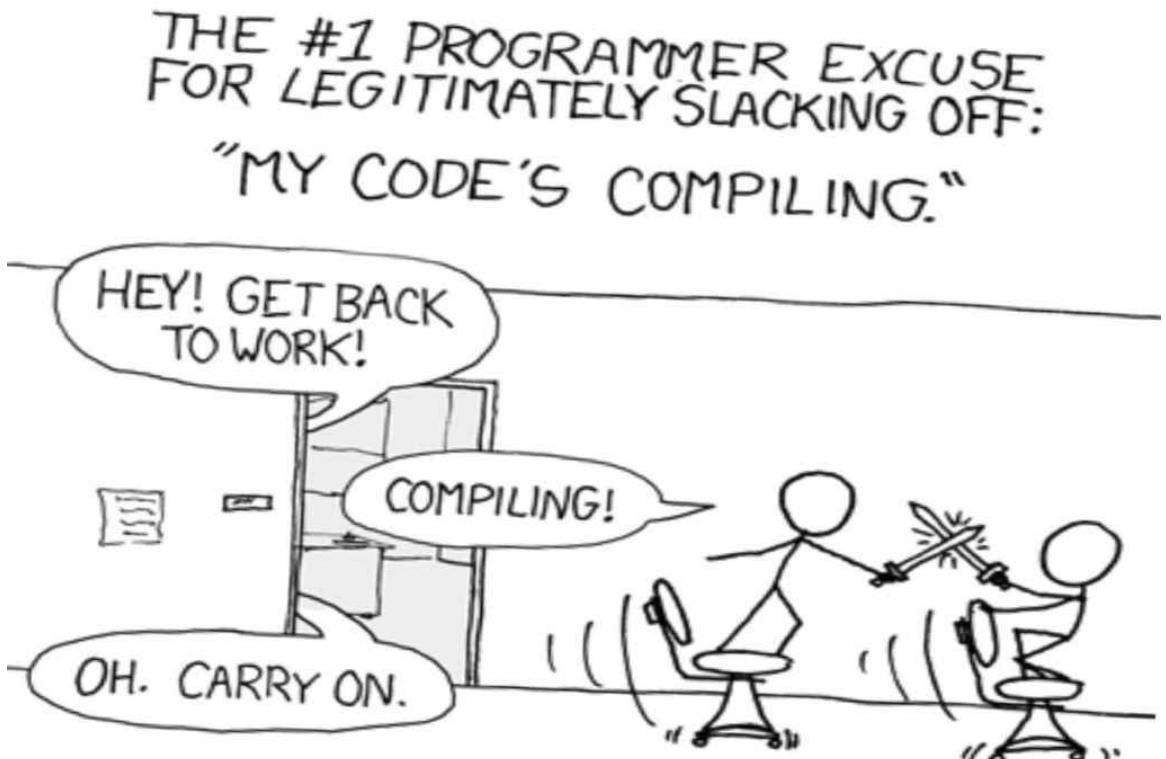
Автомобільний приклад

Більшість корисних задач занадто складні для навчання на одному GPU

	ДУЖЕ КОНСЕРВАТИВНИЙ	КОНСЕРВАТИВНИЙ
Розмір автопарку (збір даних за годину)	100 автомобілів / 1 ТБ/год	125 автомобілів / 1,5 ТБ/год
Тривалість збору даних	260 днів * 8 годин	325 днів * 10 годин
Фактор стиснення даних	0,0005	0,0008
Загальний навчальний набір	104 ТБ 	87,5 ТБ
Час навчання InceptionV3 (з графічним процесором 1 Pascal)	9,1 років	. 6 років 
Час навчання AlexNet (з 1 графічним процесором Pascal)	 1,1 року 	 5,4 років 

ВИСНОВКИ

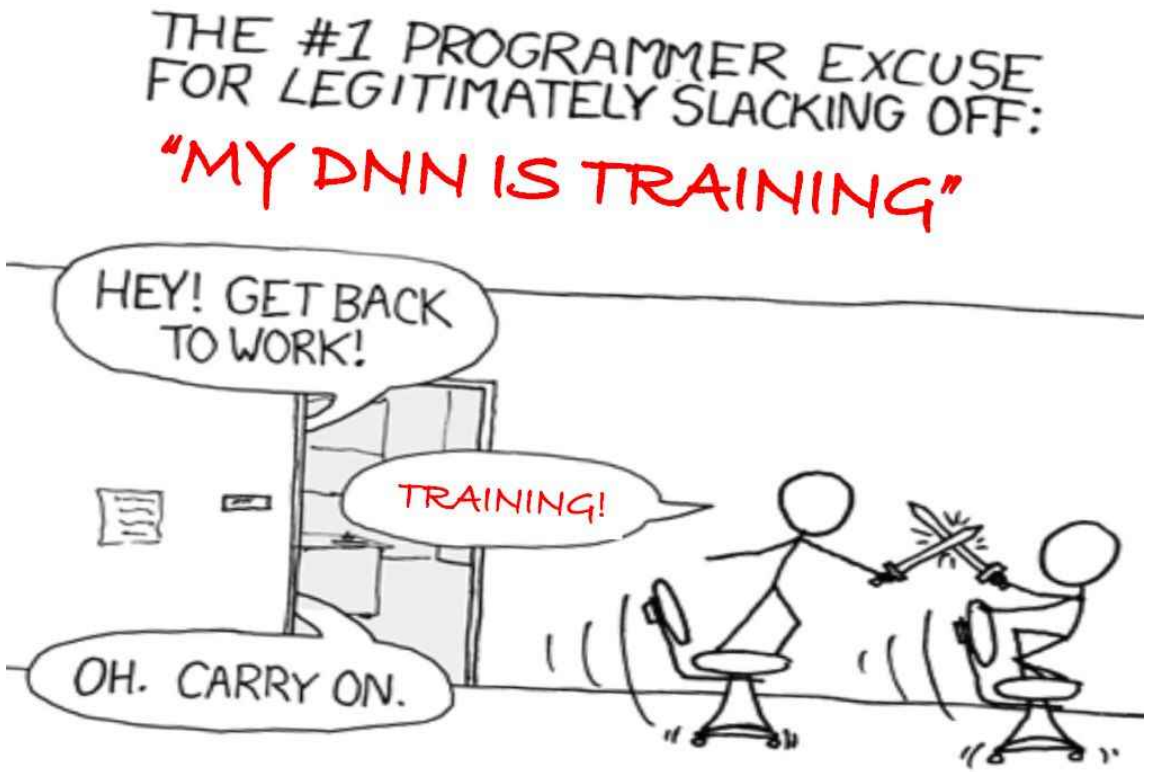
Що ваша команда робить тим часом



Xkcd.com

ВИСНОВКИ

Що ваша команда робить тим часом



Xkcd.com

ВИСНОВКИ

Потрібно масштабувати навчальний процес під одну роботу

1 NVIDIA DGX-1

10 NVIDIA DGX-1

	ДУЖЕ КОНСЕРВАТИВНИЙ	**КОНСЕРВАТИВНИЙ**
Загальний навчальний набір	104 ТБ	487,5 ТБ
InceptionV3 (один DGX-1V)	166 днів (5+**місяців**)	778 днів (2+**років**)
AlexNet (один DGX-1V)	21 день (3**тижнів**)	98 днів (3**місяців**)
Початок V3 (10 DGX-1V)	16 днів (2+**тижнів**)	77 днів (11**тижнів**)
AlexNet (10 DGX-1V)	2,1 дня	9,8 днів

Навчання

Від

Місяці або роки

до

Тижні або дні

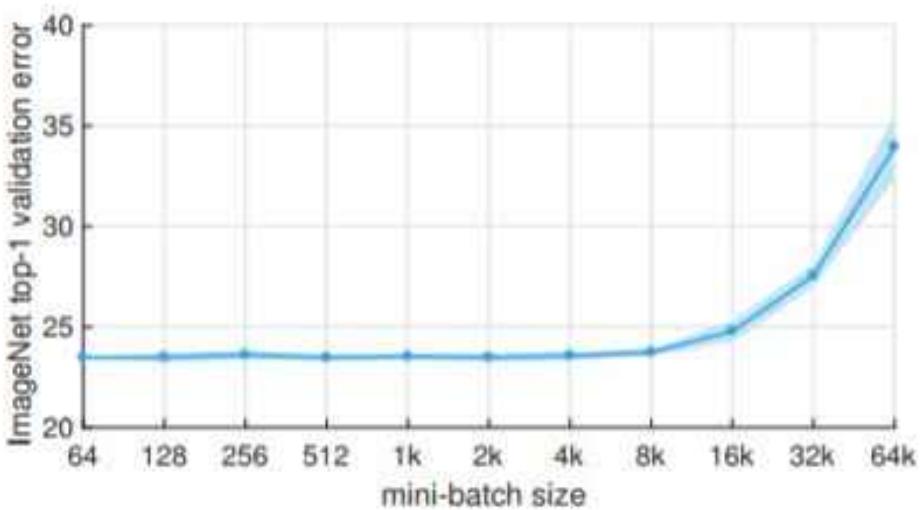
NVIDIA DEEP LEARNING INSTITUTE

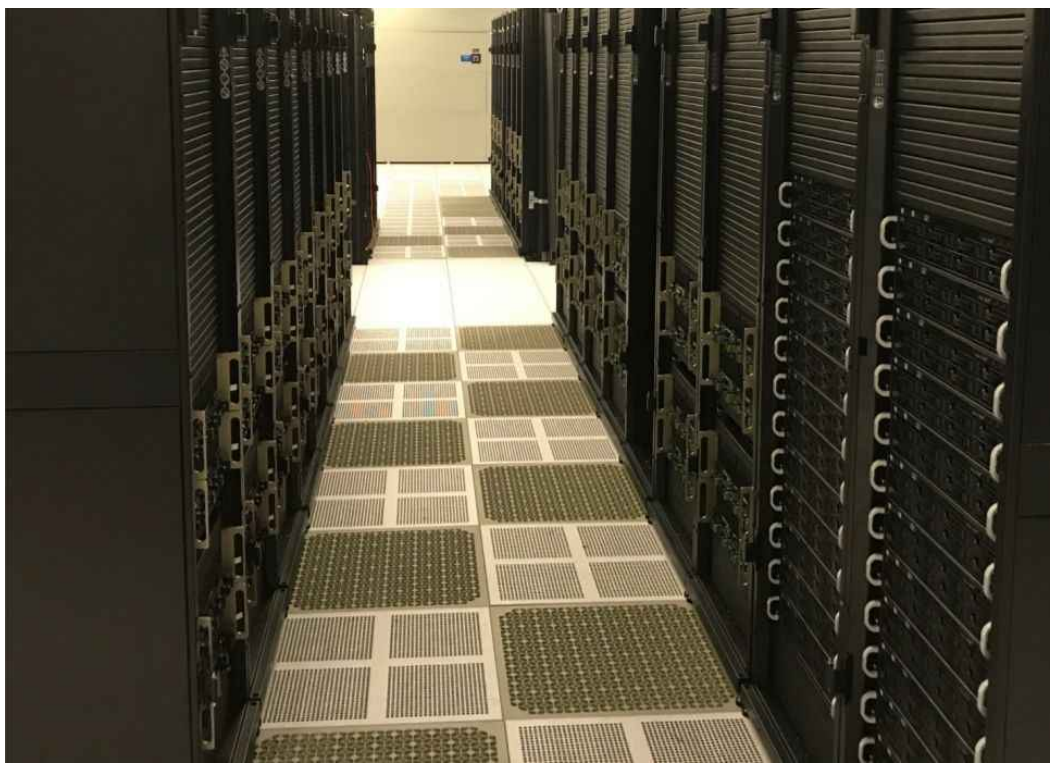
ПРАКТИЧНІ ПРИКЛАДИ ВЕЛИКОГО МАСШТАБУ НАВЧАННЯ

FACEBOOK

Навчання ImageNet із ResNet 50 за 1 годину

- 128 * DGX-1
- 10,5 PFLOPS загальний FP32
- 21 PFLOPS разом FP16
- Незлипаюча тканина IB



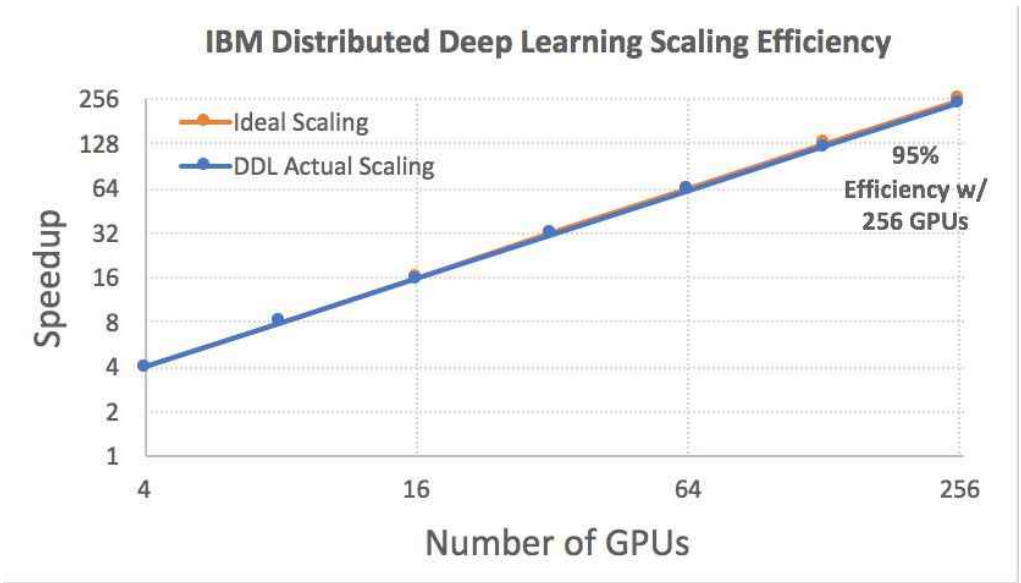


Гоял, П., Доллар, П., Гіршик, Р., Нордхуіс, П., Весоловскі, Л., Кірола, А., ... і Хе, К. (2017). Точна, велика міні серія SGD: навчання ImageNet за 1 годину. Препринт arXiv arXiv:1706.02677

IBM

Навчання ImageNet із ResNet 101 за 7 годин

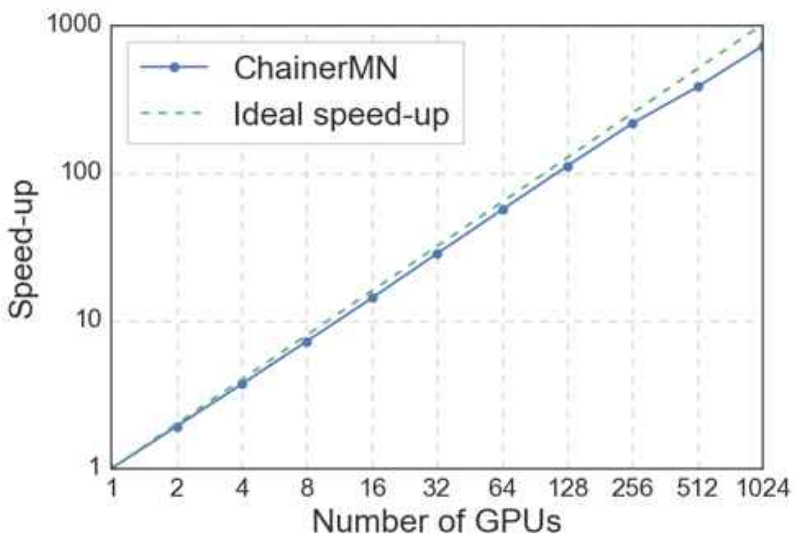
- 64 IBM Power Systems



ПЕРЕВАЖЕНІ МЕРЕЖІ

Навчання ImageNet за 15 хвилин

- Він складається з 128 вузлів з 8 графічним процесором NVIDIA P100 кожен**Всього 1024 GPU**.
- Вузли з’єднані двома лініями FDR Infiniband (56 Гбіт/с x 2).

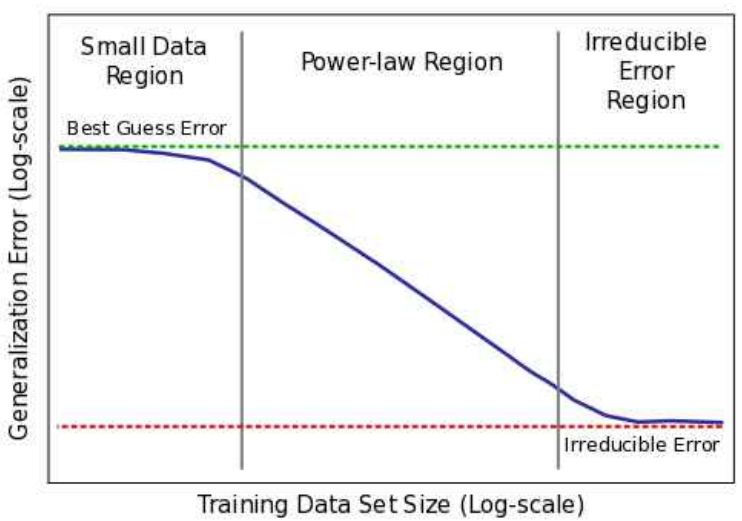


Акіба Т., Сузукі С. та Фукуда К. (2017). Надзвичайно велика міні-серія sgd: навчання resnet-50 на imagenet за 15 хвилин. Препринт arXiv arXiv:1711.04325

БАЙДУ СВАІЛ

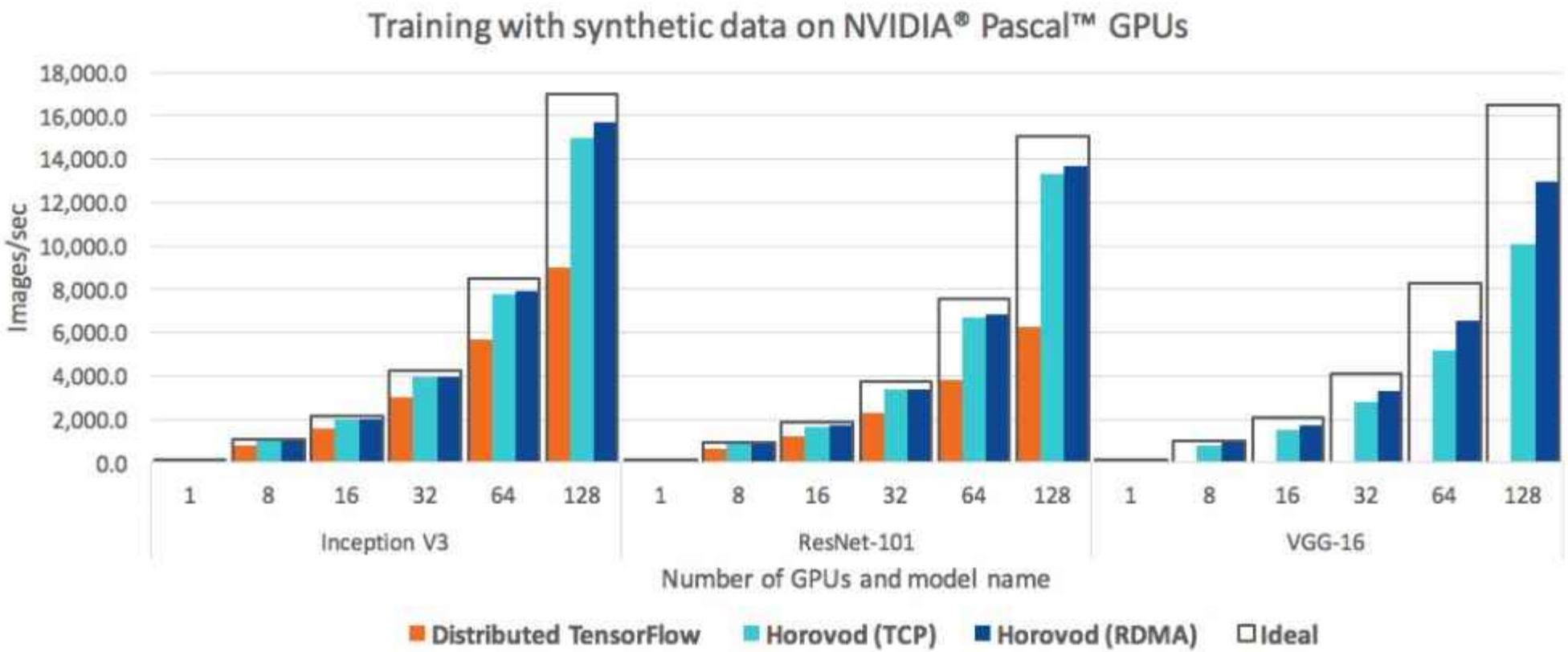
Дослідження логарифмічного лінійного характеру зв'язку між набором даних розмір і точність узагальнення

- 11 PFLOPS на 1500 GPU



UBER

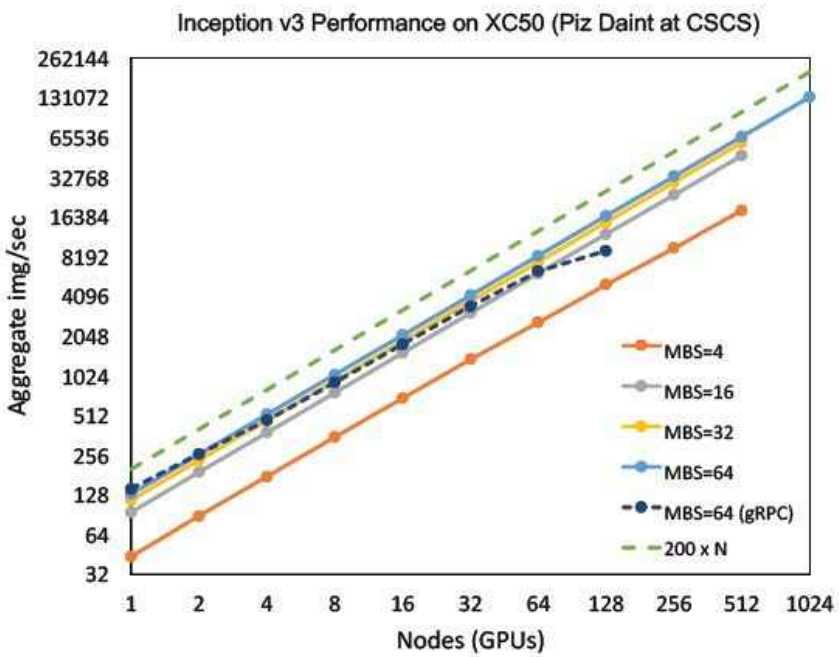
Значні інвестиції в масштабованість Deep Learning

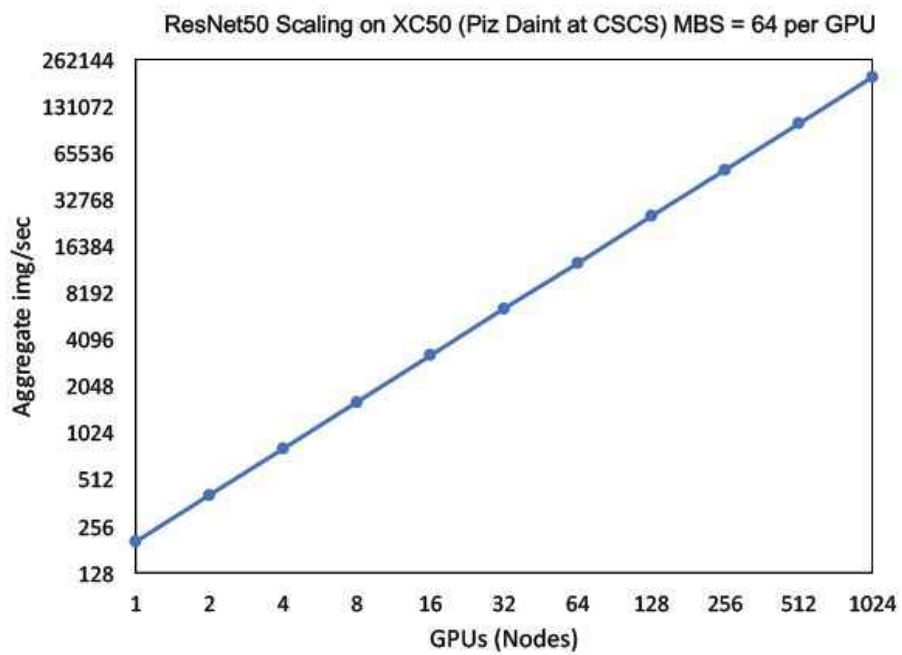


https://github.com/uber/horovod

КРЕЙ

Piz Daint в CSCS





САТУРН V

660 вузлів DGX-1 Volta

- 660 вузлів із загальною кількістю графічних процесорів Volta 5280
- 660 PFLOP для навчання ШІ

ЧАС ІТЕРАЦІЇ

Короткий час ітерації є основою успіху

незважаючи на

'ЧОРНА СКРИНЬКА'

ВНУТРІШНІ

-

КОМПЛЕКСНА ТЕОРІЯ

ПОЗАДУ

Є

ПІД

ДОСЛІДЖУЙТЕ ЗАРАЗ

Стохастичний градієнтний спуск

Більше даних і паралелізм моделей

Адаптація для Edge Computing

Масштабування з кількома GPU

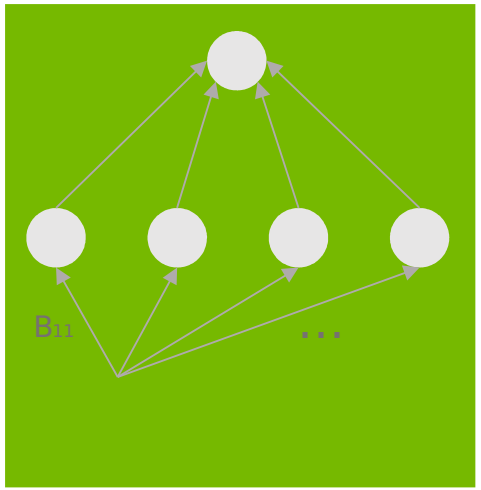
. . .

ОПТИМІЗАЦІЯ НА ОСНОВІ ГРАДІЄНТА

СТОХАСТИЧНИЙ ГРАДІЄНТНИЙ СПУСТ (ТА ЙОГО ВАРІАНТИ)

ОПТИМІЗАЦІЯ

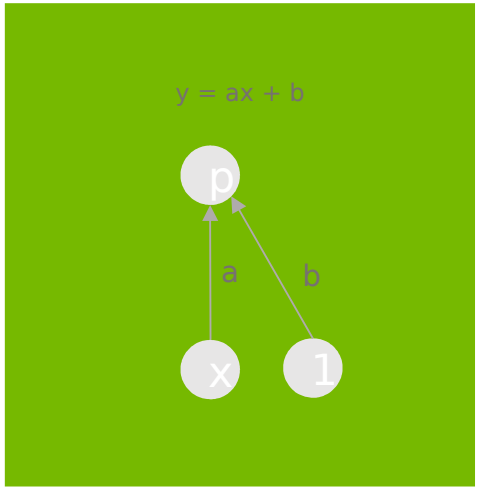
Як ми знаходимо параметри нейронної мережі в першому місце?

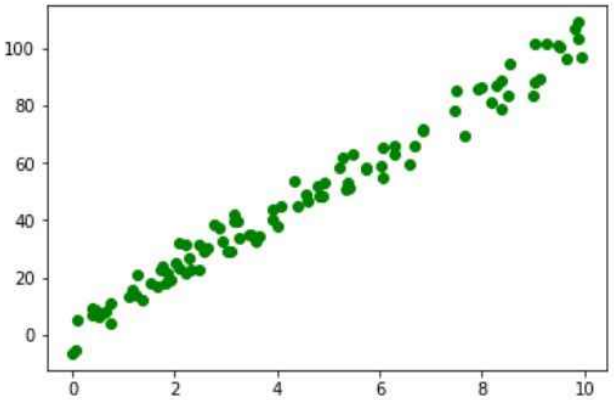


ОПТИМІЗАЦІЯ

Почнемо з найпростішої задачі – лінійного нейрона

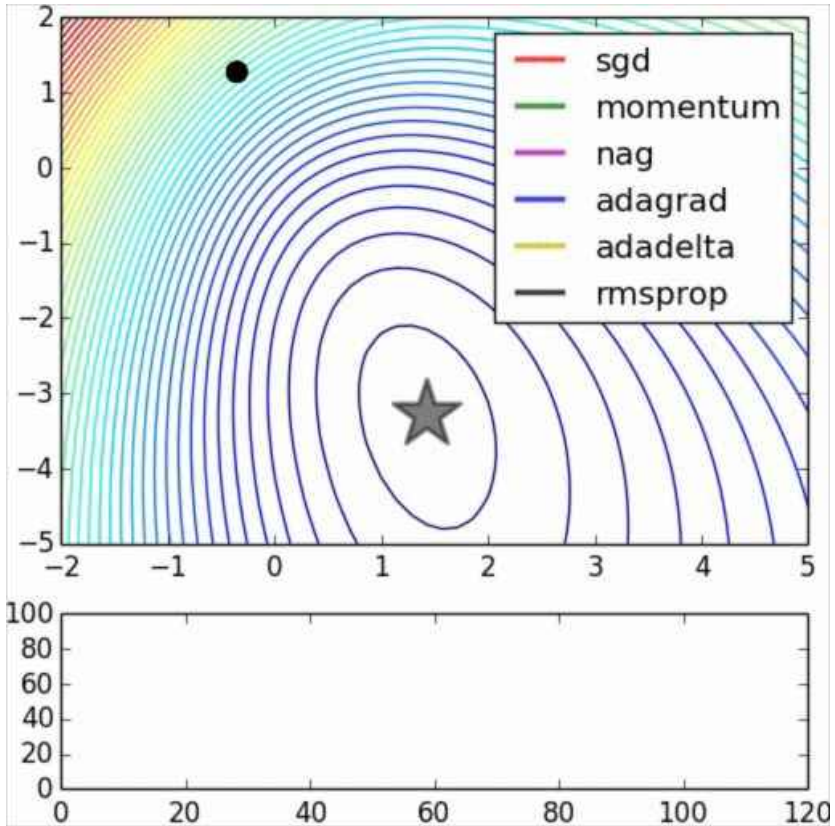
Наша мета - знайти найкращі параметри моделі (комбінація a і b), щоб підібрати дані





БІЛЬШЕ, НІЖ ПРОСТО SGD

Існує широкий спектр алгоритмів оптимізації



SGD ДЛЯ СКЛАДНІШИХ НЕЙРОМЕРЕЖ

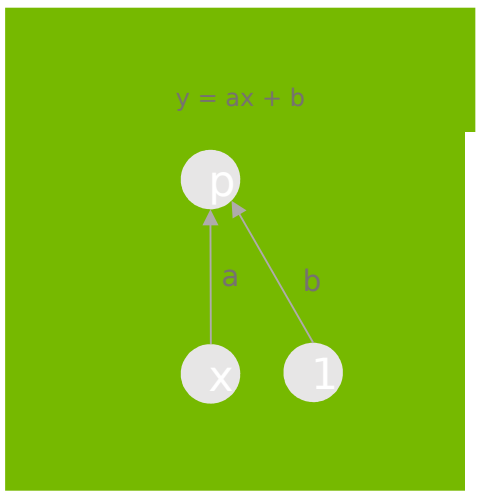
СУЧАСНІ НЕЙРОМЕРЕЖІ

Чим вони відрізняються від нашого тривіального прикладу?

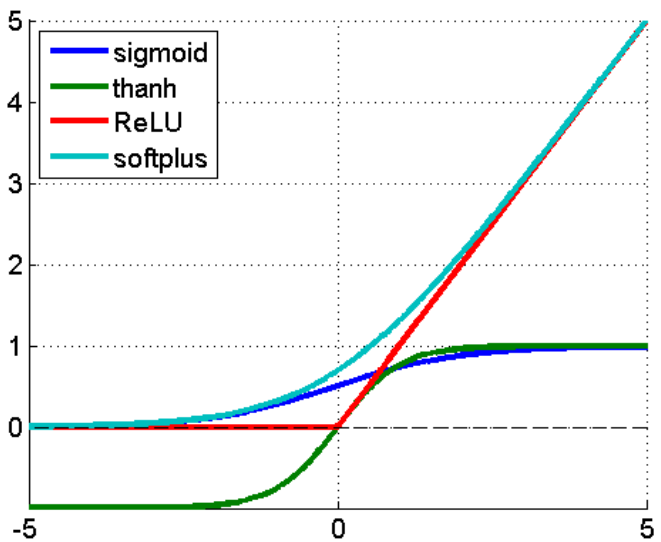
Не істотно!

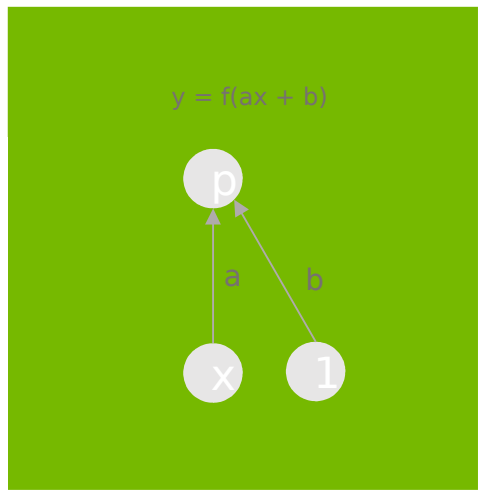
СУЧАСНІ НЕЙРОМЕРЕЖІ

Чим вони відрізняються від нашого тривіального прикладу?



Нелінійність

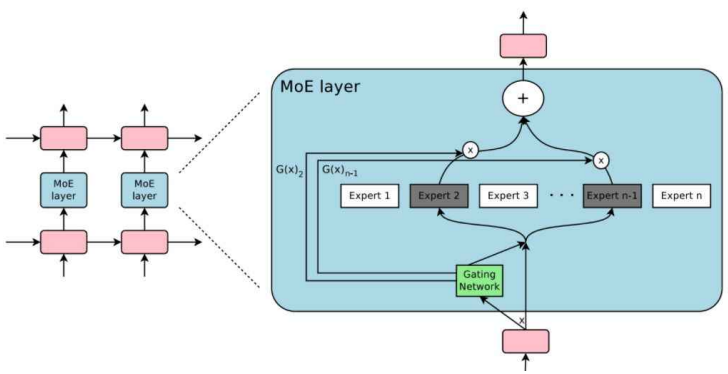


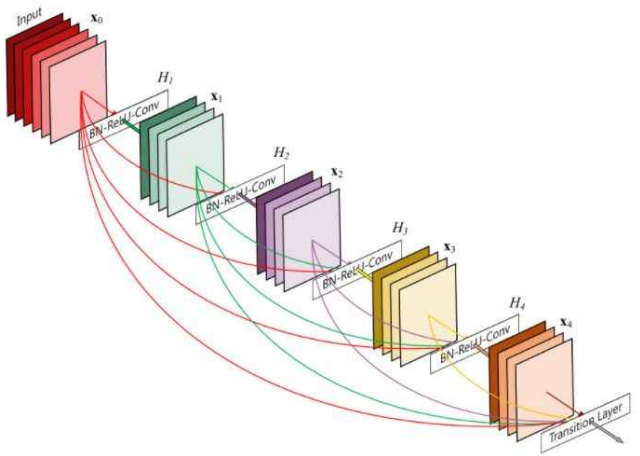


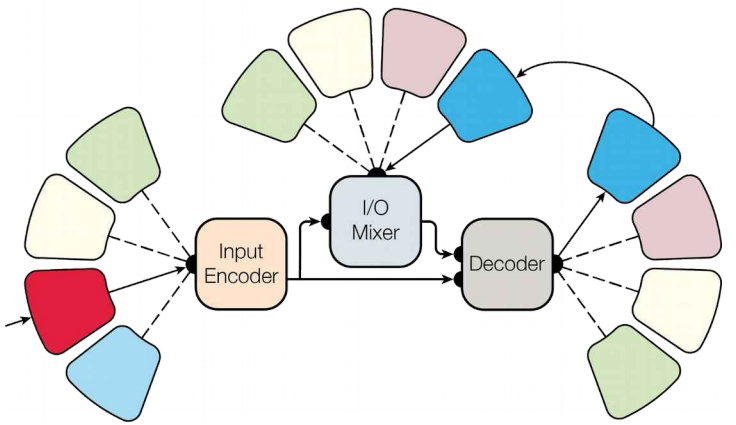
СУЧАСНІ НЕЙРОМЕРЕЖІ

Чим вони відрізняються від нашого тривіального прикладу?

Більш складні взаємозв'язки та багато іншого параметри







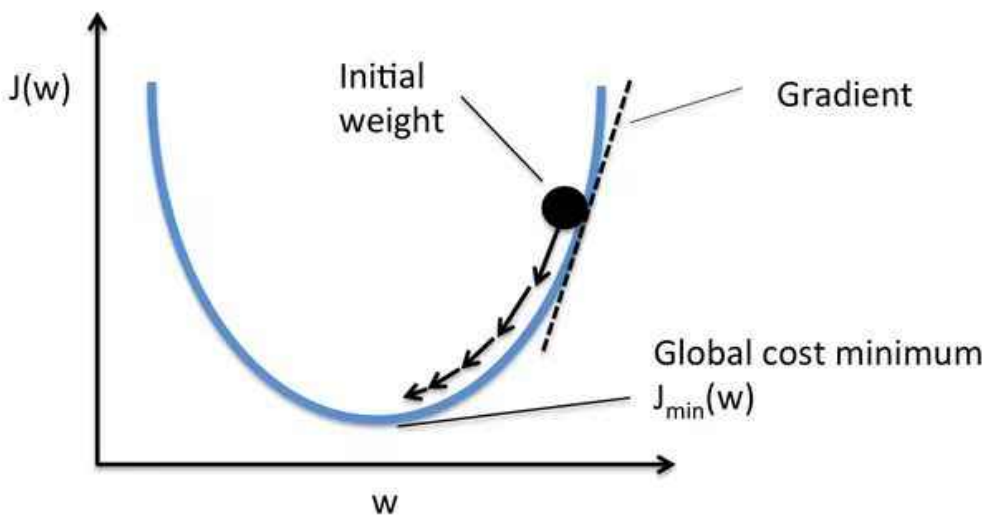
Кайзер, Л., Гомес, А.Н., Шазір, Н., Васвані, А., Пармар, Н., Джонс, Л., та Ускорейт, Дж. (2017). Одна модель, щоб вивчити їх усіх. Препринт arXiv arXiv:1706.05137

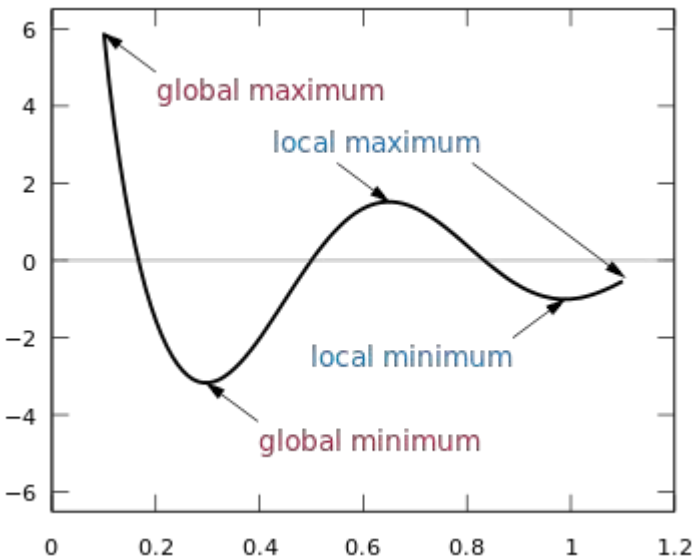
Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., & Keutzer, K. (2014). Densenet: впровадження ефективних пірамід дескрипторів мережі. Препринт arXiv arXiv:1404.1869.

Шазір, Н., Міргосейні, А., Мазіарц, К., Девіс, А., Ле, К., Хінтон, Г., і Дін, Дж. (2017). Надзвичайно великі нейронні мережі: розріджена суміш експертів. Препринт arXiv arXiv:1701.06538

НЕОПУКЛІ ФУНКЦІЇ ВАРТОСТІ

Ці відмінності значно ускладнюють проблему оптимізації

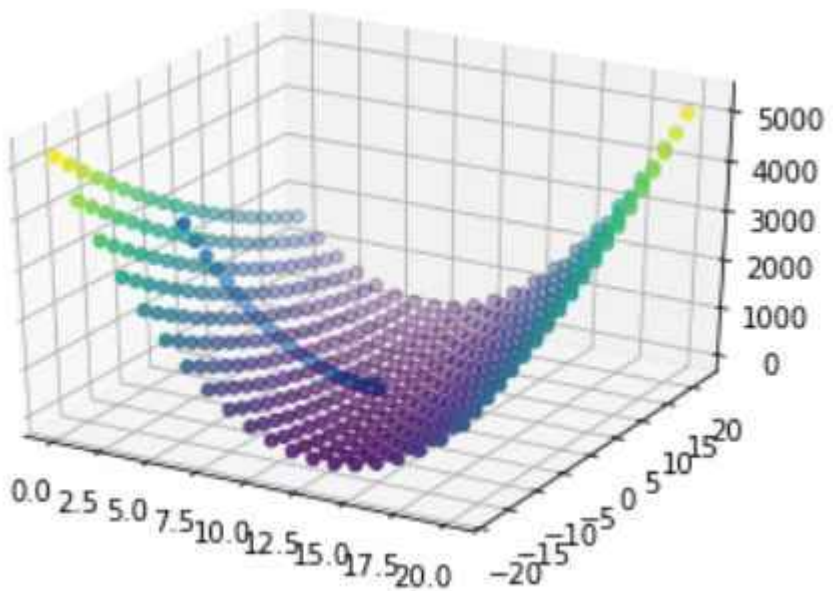




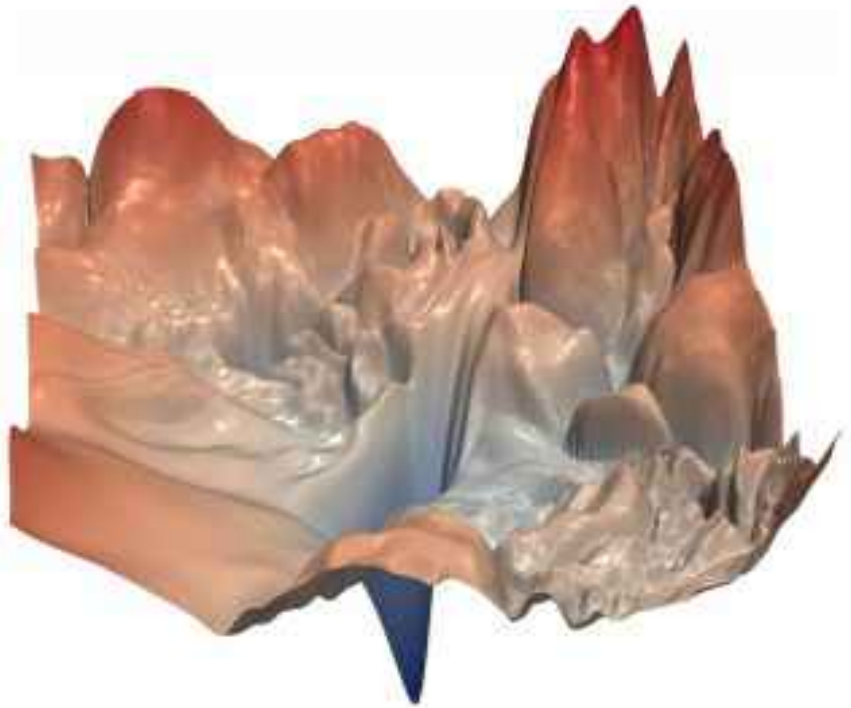
НЕОПУКЛІ ФУНКЦІЇ ВАРТОСТІ

Ці відмінності значно ускладнюють проблему оптимізації

Лінійна функція вартості нейрона



Проекція функції витрат ResNet 56 у 3D – без пропуску з'єднання



Лі Х., Сю З., Тейлор Г. та Гольдштейн Т. (2017). Візуалізація ландшафту втрат нейронних мереж.Препринт arXiv arXiv:1712.09913

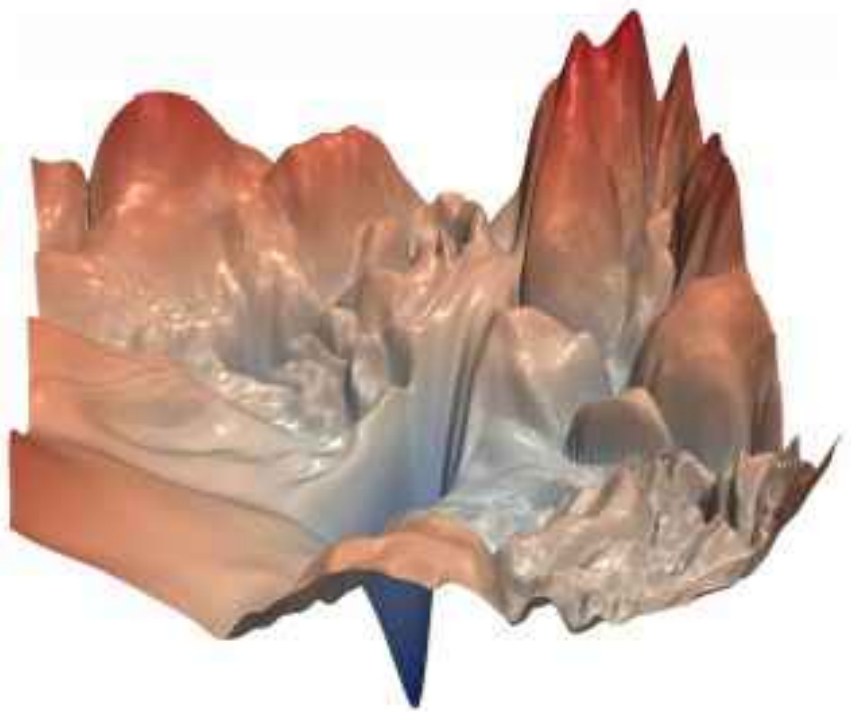
НЕОПУКЛІ ФУНКЦІЇ ВАРТОСТІ

Ці відмінності значно ускладнюють проблему оптимізації

Чому нам вдається знайти

хороші місцеві мінімуми?

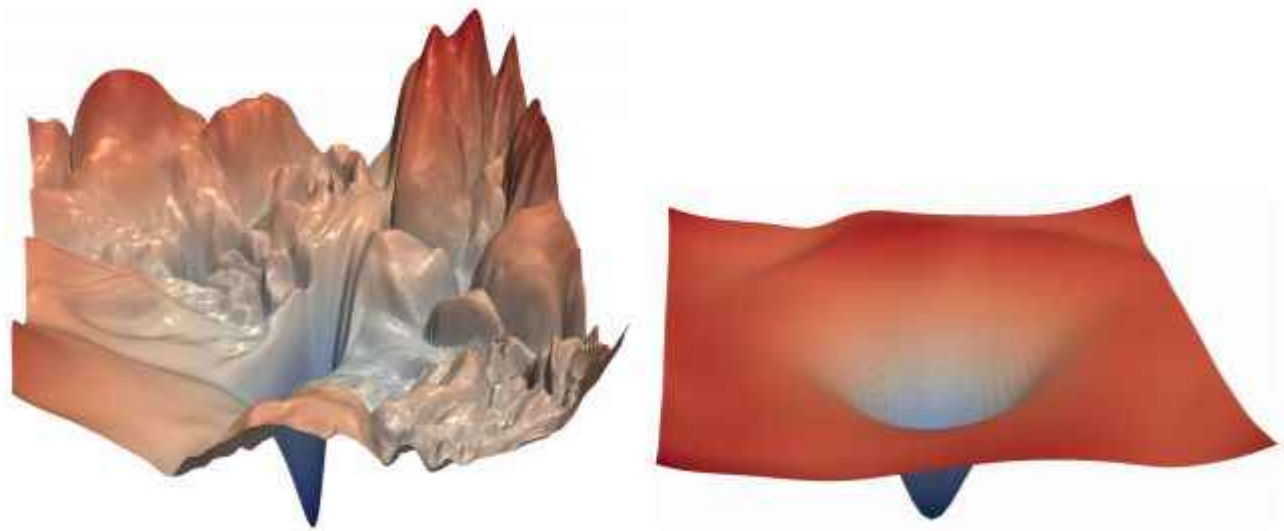
Проекція функції витрат ResNet 56 у 3D – без пропуску з'єднання



Лі Х., Сю З., Тейлор Г. та Гольдштейн Т. (2017). Візуалізація ландшафту втрат нейронних мереж.Препринт arXiv arXiv:1712.09913

НЕОПУКЛІ ФУНКЦІЇ ВАРТОСТІ

Останні досягнення, такі як залишкові з’єднання, спрощують задача оптимізації



(a) without skip connections (b) with skip connections

Лі Х., Сю З., Тейлор Г. та Гольдштейн Т. (2017). Візуалізація ландшафту втрат нейронних мереж.Препринт arXiv arXiv:1712.09913

ОПТИМІЗАЦІЯ

Хоча підхід той самий

- Визначити модель (багатошарова нейронна мережа)
- Визначте функцію витрат (конкретна проблема)
- Ітеративно:
  - Обчислити градієнт функції витрат (алгоритм отримання градієнта називається зворотне поширення)
  - Оновіть параметри моделі (знову використовуючи один із багатьох алгоритмів оптимізації)

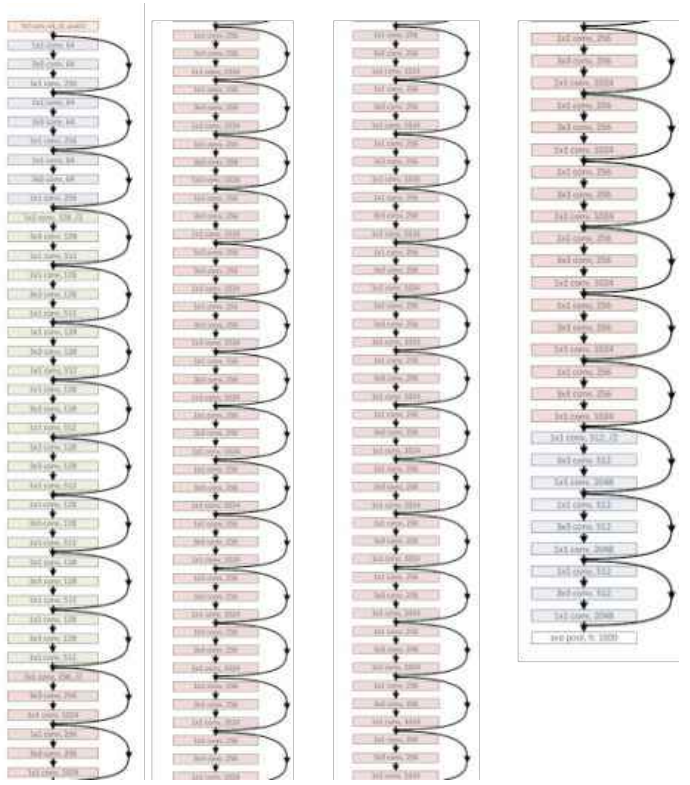
ЗВОРОТНЕ ПОШИРЕННЯ

Як ми обчислюємо градієнт такої складної функції?

$$y = f(u),\ u = g(x)$$

Therefore, $y = (f \circ g)(x)$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$



ЗВОРОТНЕ ПОШИРЕННЯ

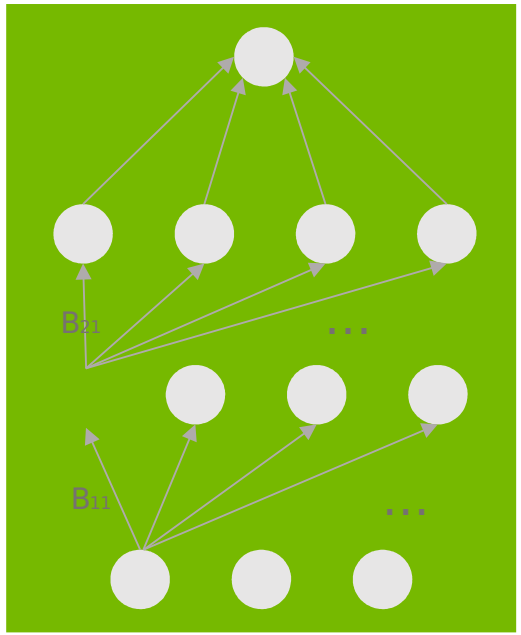
Автоматична диференціація

- На практиці це рідко чи взагалі робиться вручну, оскільки всі фреймворки глибокого навчання постачаються з:
  - Набір попередньо створених алгоритмів оптимізації
  - Функція автоматичного розрізнення
- Дуже корисним побічним ефектом є те, що ви можете вставити БУДЬ-ЯКИЙ ВІДМІННИЙ код у свою нейронну мережу!

ОПТИМІЗАЦІЯ

Почнемо з найпростішої нейронної мережі – багатошарового перцептрона

- Ми побудуємо просту (2 приховані шари) нейронну систему мережа – багатошарова перцептрон (ні нелінійність)
- Ми будемо працювати з набором даних MNIST
- Наша мета — знайти найкращі параметри моделі, які відповідають даним




NVIDIA
DEEP
LEARNING
INSTITUTE
www.nvidia.com/dli

CPU version - MNIST digit classification in TensorFlow 2.0

IMPORTANT: Runtime -> Change runtime -> None

Now, we will see how can we perform the MNIST handwritten digits classification using tensorflow 2.0. It hardly a few lines of code compared to the tensorflow 1.x. As we learned, tensorflow 2.0 uses as keras as its high-level API, we just need to add tf.keras to the keras code.

Enabling and testing the environment

```
! cat /sys/class/dmi/id/product_name
```

```
Google Compute Engine
```

```
! cat  /sys/class/dmi/id/sys_vendor
```

```
Google
```

```
! lscpu
```

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  2
Core(s) per socket:  1
Socket(s):           1
NUMA node(s):        1
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               49
Model name:          AMD EPYC 7B12
Stepping:            0
CPU MHz:             2249.998
BogoMIPS:            4499.99
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            16384K
NUMA node0 CPU(s):   0,1
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
```

```
! grep MemTotal /proc/meminfo
```

```
MemTotal:       13333596 kB
```

```
! df -h
```

```
Filesystem      Size  Used Avail Use% Mounted on
overlay         108G   31G   78G  28% /
tmpfs            64M     0   64M   0% /dev
tmpfs           6.4G     0  6.4G   0% /sys/fs/cgroup
shm             5.9G     0  5.9G   0% /dev/shm
tmpfs           6.4G   28K  6.4G   1% /var/colab
/dev/sda1       114G   32G   83G  28% /etc/hosts
tmpfs           6.4G     0  6.4G   0% /proc/acpi
tmpfs           6.4G     0  6.4G   0% /proc/scsi
tmpfs           6.4G     0  6.4G   0% /sys/firmware
```

```
! nvidia-smi
```

```
NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver.
```

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
---------------------------------------------------------------------------
--------
SystemError                               Traceback (most recent
call last)
<ipython-input-7-d1680108c58e> in <module>()
      2 device_name = tf.test.gpu_device_name()
      3 if device_name != '/device:GPU:0':
----> 4   raise SystemError('GPU device not found')
      5 print('Found GPU at: {}'.format(device_name))

SystemError: GPU device not found
```

Import the libraries:

```
import warnings
warnings.filterwarnings('ignore')

import tensorflow as tf
```

Check Tensorflow version

```
print(tf.__version__)
```

```
2.4.1
```

Load the dataset:

```
mnist =  tf.keras.datasets.mnist
```

Create a train and test set:

```
(x_train,y_train), (x_test, y_test) = mnist.load_data()
```

Normalize data ...

... the x values by diving with maximum value of x which is 255 and convert them to float:

```
x_train, x_test = tf.cast(x_train/255.0, tf.float32), tf.cast(x_test/255.0, tf.flo
```

convert y values to int:

```
y_train, y_test = tf.cast(y_train,tf.int64),tf.cast(y_test,tf.int64)
```

Create the model

Define the sequential model:

Define the sequential model:

```
model = tf.keras.models.Sequential()
```

Add the layers - We use a three-layered network. We apply ReLU activation at the first two layers and in the final output layer we apply softmax function:

```
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dense(128, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Compile the model with Stochastic Gradient Descent, that is 'sgd' (we will learn about this in the next chapter) as optimizer and sparse_categorical_crossentropy as loss function and with

accuracy as a metric:

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['a
```

- List item
- List item

Train

Train the model for 10 epochs with batch_size as 32:

```
history = model.fit(x_train, y_train, batch_size=32, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 5s 2ms/step - loss: 1.0382 - acc
Epoch 2/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.3054 - acc
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2385 - acc
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2010 - acc
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1730 - acc
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1538 - acc
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1348 - acc
Epoch 8/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1239 - acc
Epoch 9/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1116 - acc
Epoch 10/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1017 - acc
```

Show the structure of the model

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (32, 784)                 0
_________________________________________________________________
dense (Dense)                (32, 256)                 200960
_________________________________________________________________
dense_1 (Dense)              (32, 128)                 32896
_________________________________________________________________
dense_2 (Dense)              (32, 10)                  1290
=================================================================
Total params: 235,146
```

```
Trainable params: 235,146
Non-trainable params: 0
_________________________________________________________________
```

Evaluate

Evaluate the model on test sets:

```
model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 1s 1ms/step - loss: 0.1082 - accur
[0.10815522819757462, 0.9682999849319458]
```

GPU version - MNIST digit classification in TensorFlow 2.0

IMPORTANT: Runtime -> Change runtime -> GPU

Now, we will see how can we perform the MNIST handwritten digits classification using tensorflow 2.0. It hardly a few lines of code compared to the tensorflow 1.x. As we learned, tensorflow 2.0 uses as keras as its high-level API, we just need to add tf.keras to the keras code.

Enabling and testing the GPU

```
! nvidia-smi
```

```
Tue Feb 23 13:03:10 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.39       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   37C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

Import the libraries:

```
import warnings
warnings.filterwarnings('ignore')
```

```
import tensorflow as tf
```

Check Tensorflow version

```
print(tf.__version__)
```

```
2.4.1
```

Load the dataset:

```
mnist =  tf.keras.datasets.mnist
```

Create a train and test set:

```
(x_train,y_train), (x_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
11493376/11490434 [==============================] - 0s 0us/step
```

Normalize data ...

... the x values by diving with maximum value of x which is 255 and convert them to float:

```
x_train, x_test = tf.cast(x_train/255.0, tf.float32), tf.cast(x_test/255.0, tf.flo
```

convert y values to int:

```
y_train, y_test = tf.cast(y_train,tf.int64),tf.cast(y_test,tf.int64)
```

Create the model

Define the sequential model:

Define the sequential model:

```
model = tf.keras.models.Sequential()
```

Add the layers - We use a three-layered network. We apply ReLU activation at the first two layers

```
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dense(128, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Compile the model with Stochastic Gradient Descent, that is 'sgd' (we will learn about this in the next chapter) as optimizer and sparse_categorical_crossentropy as loss function and with accuracy as a metric:

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['a
```

Show the structure of the model

Train

Train the model for 10 epochs with batch_size as 32:

```
history = model.fit(x_train, y_train, batch_size=32, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 5s 2ms/step - loss: 1.0041 - acc
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2966 - acc
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2382 - acc
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2003 - acc
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1774 - acc
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1537 - acc
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1370 - acc
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1232 - acc
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1089 - acc
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1030 - acc
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (32, 784)                 0
```

```
_________________________________________________________________
dense (Dense)                (32, 256)                 200960
_________________________________________________________________
dense_1 (Dense)              (32, 128)                 32896
_________________________________________________________________
dense_2 (Dense)              (32, 10)                  1290
=================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
_________________________________________________________________
```

Evaluate

Evaluate the model on test sets:

```
model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1084 - accur
[0.10837740451097488, 0.96670001745224]
```

TPU version - MNIST digit classification in TensorFlow 2.0

IMPORTANT: Runtime -> Change runtime -> TPU

Now, we will see how can we perform the MNIST handwritten digits classification using tensorflow 2.0. It hardly a few lines of code compared to the tensorflow 1.x. As we learned, tensorflow 2.0 uses as keras as its high-level API, we just need to add tf.keras to the keras code.

Enabling and testing the TPU

First, you'll need to enable TPUs for the notebook:

- Navigate to Edit→Notebook Settings
- select TPU from the Hardware Accelerator drop-down

Next, we'll check that we can connect to the TPU:

```
%tensorflow_version 2.x
import tensorflow as tf
print("Tensorflow version " + tf.__version__)

try:
  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()  # TPU detection
  print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
except ValueError:
  raise BaseException('ERROR: Not connected to a TPU runtime; please see the previ

tf.config.experimental_connect_to_cluster(tpu)
tf.tpu.experimental.initialize_tpu_system(tpu)
tpu_strategy = tf.distribute.experimental.TPUStrategy(tpu)
```

```
Tensorflow version 2.4.1
Running on TPU  ['10.15.163.122:8470']
INFO:tensorflow:Initializing the TPU system: grpc://10.15.163.122:8470
INFO:tensorflow:Initializing the TPU system: grpc://10.15.163.122:8470
INFO:tensorflow:Clearing out eager caches
INFO:tensorflow:Clearing out eager caches
INFO:tensorflow:Finished initializing TPU system.
INFO:tensorflow:Finished initializing TPU system.
WARNING:absl:`tf.distribute.experimental.TPUStrategy` is deprecated, please u
INFO:tensorflow:Found TPU system:
INFO:tensorflow:Found TPU system:
INFO:tensorflow:*** Num TPU Cores: 8
INFO:tensorflow:*** Num TPU Cores: 8
INFO:tensorflow:*** Num TPU Workers: 1
INFO:tensorflow:*** Num TPU Workers: 1
INFO:tensorflow:*** Num TPU Cores Per Worker: 8
INFO:tensorflow:*** Num TPU Cores Per Worker: 8
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:localhost/replic
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:localhost/replic
```

```
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
INFO:tensorflow:*** Available Device: _DeviceAttributes(/job:worker/replica:
```

Import the libraries:

```
import warnings
warnings.filterwarnings('ignore')

import tensorflow as tf
```

Check Tensorflow version

```
print(tf.__version__)
```

```
2.4.1
```

Load the dataset:

```
mnist =  tf.keras.datasets.mnist
```

Create a train and test set:

```
(x_train,y_train), (x_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dat
11493376/11490434 [==============================] - 0s 0us/step
```

Normalize data ...

... the x values by diving with maximum value of x which is 255 and convert them to float:

```
x_train, x_test = tf.cast(x_train/255.0, tf.float32), tf.cast(x_test/255.0, tf.flo
```

convert y values to int:

```
y_train, y_test = tf.cast(y_train,tf.int64),tf.cast(y_test,tf.int64)
```

Create the model

Define the sequential model:

```
model = tf.keras.models.Sequential()
```

Add the layers - We use a three-layered network. We apply ReLU activation at the first two layers and in the final output layer we apply softmax function:

```
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dense(128, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Compile the model with Stochastic Gradient Descent, that is 'sgd' (we will learn about this in the next chapter) as optimizer and sparse_categorical_crossentropy as loss function and with accuracy as a metric:

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['a
```

Train

Train the model for 10 epochs with batch_size as 32:

```
history = model.fit(x_train, y_train, batch_size=32, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.9990 - a
Epoch 2/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.2995 - a
Epoch 3/10
```

```
1875/1875 [==============================] - 20s 11ms/step - loss: 0.2413 - a
Epoch 4/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.2023 - a
Epoch 5/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1679 - a
Epoch 6/10
1875/1875 [==============================] - 22s 12ms/step - loss: 0.1526 - a
Epoch 7/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.1340 - a
Epoch 8/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.1230 - a
Epoch 9/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.1105 - a
Epoch 10/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.1019 - a
```

Show the structure of the model

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (32, 784)                 0
_________________________________________________________________
dense (Dense)                (32, 256)                 200960
_________________________________________________________________
dense_1 (Dense)              (32, 128)                 32896
_________________________________________________________________
dense_2 (Dense)              (32, 10)                  1290
=================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
_________________________________________________________________
```

Compare with training results on GPU

Epoch 1/10 1875/1875 [==============================] - 6s 2ms/step - loss: 0.9975 - accuracy: 0.7304

Epoch 2/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.3008 - accuracy: 0.9134

Epoch 3/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.2401 - accuracy: 0.9320

Epoch 4/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.2071 - accuracy: 0.9423

Epoch 5/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.1794 - accuracy: 0.9492

Epoch 6/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.1576 - accuracy: 0.9548

Epoch 7/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.1393 - accuracy: 0.9613

Epoch 8/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.1309 - accuracy: 0.9628

Epoch 9/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.1119 - accuracy: 0.9680

Epoch 10/10 1875/1875 [==============================] - 4s 2ms/step - loss: 0.1042 -

Evaluate

Evaluate the model on test sets:

```
model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 3s 9ms/step - loss: 0.1062 - accur
[0.10615649074316025, 0.968500018119812]
```

Compare with training results on GPU

313/313 [==============================] - 1s 2ms/step - loss: 0.1077 - accuracy: 0.9671

[0.10774651169776917, 0.9671000242233276]

The results are nearly the same up to 3rd (loss) and 4th (accuracy) significant number after the decimal point.

```

```

Colab paid products - Cancel contracts here

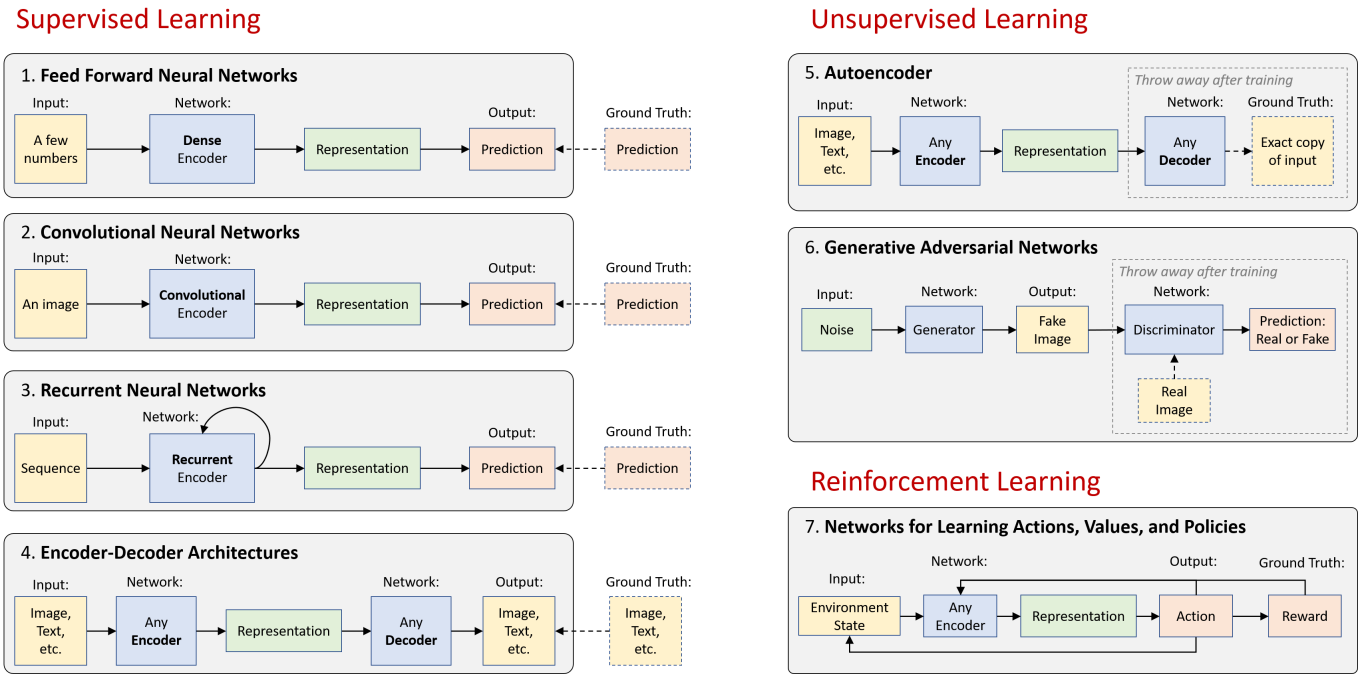
Deep Learning Basics

Main Types of Deep Neural Networks

based on (C) MIT Deep Learning course

This tutorial accompanies the lecture on Deep Learning Basics given as part of MIT Deep Learning. Acknowledgement to amazing people involved is provided throughout the tutorial and at the end. You can watch the video on YouTube:

In this tutorial, we mention seven important types/concepts/approaches in deep learning, introducing the first 2 and providing pointers to tutorials on the others. Here is a visual representation of the seven:



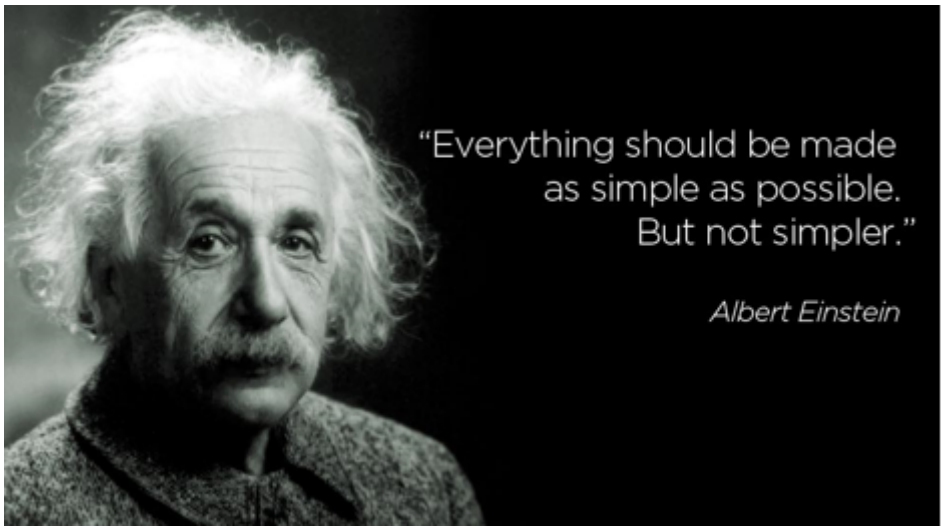
At a high-level, neural networks are either encoders, decoders, or a combination of both. Encoders find patterns in raw data to form compact, useful representations. Decoders generate new data or high-resolution useful infomation from those representations. As the lecture describes, deep learning discovers ways to **represent** the world so that we can reason about it. The rest is clever methods that help use deal effectively with visual information, language, sound (#1-6) and even act in a world based on this information and occasional rewards (#7).

1. **Feed Forward Neural Networks (FFNNs)** - classification and regression based on features. See Part 1 of this tutorial for an example.
2. **Convolutional Neural Networks (CNNs)** - image classification, object detection, video action recognition, etc. See Part 2 of this tutorial for an example.
3. **Recurrent Neural Networks (RNNs)** - language modeling, speech recognition/generation, etc. See this TF tutorial on text generation for an example.
4. **Encoder Decoder Architectures** - semantic segmentation, machine translation, etc. See our tutorial on semantic segmentation for an example.
5. **Autoencoder** - unsupervised embeddings, denoising, etc.
6. **Generative Adversarial Networks (GANs)** - unsupervised generation of realistic images, etc. See this TF tutorial on DCGANs for an example.
7. **Deep Reinforcement Learning** - game playing, robotics in simulation, self-play, neural arhitecture search, etc. We'll be releasing notebooks on this soon and will link them here.

There are selective omissions and simplifications throughout these tutorials, hopefully without losing the essence of the underlying ideas. See Einstein quote...

Part 0: Prerequisites:

We recommend that you run this this notebook in the cloud on Google Colab (see link with icon at the top) if you're not already doing so. It's the simplest way to get started. You can also install TensorFlow locally. But, again, simple is best (with caveats):



tf.keras is the simplest way to build and train neural network models in TensorFlow. So, that's what we'll stick with in this tutorial, unless the models neccessitate a lower-level API.

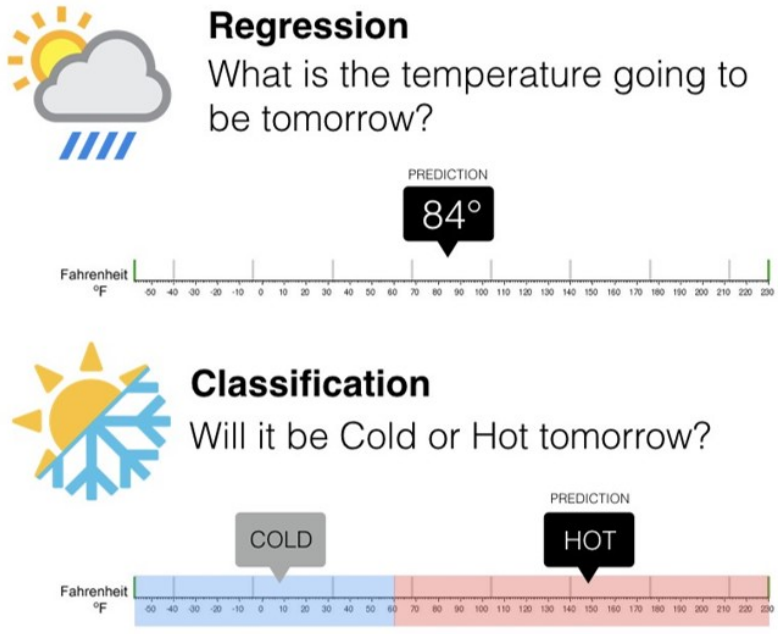
Note that there's tf.keras (comes with TensorFlow) and there's Keras (standalone). You should be using tf.keras because (1) it comes with TensorFlow so you don't need to install anything extra and (2) it comes with powerful TensorFlow-specific features.

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense

# Commonly used modules
import numpy as np
import os
import sys

# Images, plots, display, and visualization
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import cv2
import IPython
from six.moves import urllib

print(tf.__version__)
```

```
2.4.1
```

Part 1: Boston Housing Price Prediction with Feed Forward Neural Networks

Let's start with using a fully-connected neural network to do predict housing prices. The following image highlights the difference between regression and classification (see part 2). Given an observation as input, **regression** outputs a continuous value (e.g., exact temperature) and classificaiton outputs a class/category that the observation belongs to.



For the Boston housing dataset, we get 506 rows of data, with 13 features in each. Our task is to build a regression model that takes these 13 features as input and output a single value prediction of the "median value of owner-occupied homes (in $1000)."

Now, we load the dataset. Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the *training set*—the data the model uses to learn.
- The model is tested against the *test set*, the `test_images`, and `test_labels` arrays.

```
(train_features, train_labels), (test_features, test_labels) = keras.datasets.bost

# get per-feature statistics (mean, standard deviation) from the training set to n
train_mean = np.mean(train_features, axis=0)
train_std = np.std(train_features, axis=0)
train_features = (train_features - train_mean) / train_std
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
57344/57026 [==============================] - 0s 0us/step
```

Build the model

Building the neural network requires configuring the layers of the model, then compiling the model. First we stack a few layers together using `keras.Sequential`. Next we configure the loss function, optimizer, and metrics to monitor. These are added during the model's compile step:

- *Loss function* - measures how accurate the model is during training, we want to minimize this with the optimizer.
- *Optimizer* - how the model is updated based on the data it sees and its loss function.
- *Metrics* - used to monitor the training and testing steps.

Let's build a network with 1 hidden layer of 20 neurons, and use mean squared error (MSE) as the loss function (most common one for regression problems):

```
def build_model():
    model = keras.Sequential([
        Dense(20, activation=tf.nn.relu, input_shape=[len(train_features[0])]),
        Dense(1)
    ])

    # for TF1
    #model.compile(optimizer=tf.train.AdamOptimizer(),
    # for TF2: tf.train.AdamOptimizer() => tf.optimizers.Adam()
    model.compile(optimizer=tf.optimizers.Adam(),
                  loss='mse',
                  metrics=['mae', 'mse'])
    return model
```

```
# this helps makes our output less verbose but still shows progress
class PrintDot(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        if epoch % 100 == 0: print('')
```

```
        print('.', end='')

model = build_model()
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 20)                280
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 21
=================================================================
Total params: 301
Trainable params: 301
Non-trainable params: 0
_________________________________________________________________
```

Train the model

Training the neural network model requires the following steps:

1. Feed the training data to the model—in this example, the `train_features` and `train_labels` arrays.
2. The model learns to associate features and labels.
3. We ask the model to make predictions about a test set—in this example, the `test_features` array. We verify that the predictions match the labels from the `test_labels` array.

To start training, call the `model.fit` method—the model is "fit" to the training data:

```
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=50)
history = model.fit(train_features, train_labels, epochs=1000, verbose=0, validati
                    callbacks=[early_stop, PrintDot()])

hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
```

```
....................................................................................
....................................................................................
....................................................................................
....................................................................................
....................................................................................
....................................................................................
```

```
hist.head()
```

	loss	mae	mse	val_loss	val_mae	val_mse	epo
0	587.850525	22.342070	587.850525	497.338501	21.296099	497.338501	
1	578.270081	22.107761	578.270081	488.346130	21.056395	488.346130	
2	568.495544	21.872797	568.495544	479.016754	20.807596	479.016754	
3	558.563049	21.626896	558.563049	469.127014	20.536999	469.127014	

```
# show RMSE measure to compare to Kaggle leaderboard on https://www.kaggle.com/c/b
rmse_final = np.sqrt(float(hist['val_mse'].tail(1)))
print()
print('Final Root Mean Square Error on validation set: {}'.format(round(rmse_final
```

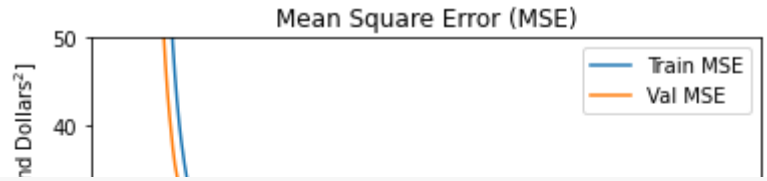
```
    Final Root Mean Square Error on validation set: 2.383
```

```
# show RMSE measure to compare to Kaggle leaderboard on https://www.kaggle.com/c/b
rmse_final = np.sqrt(float(hist['val_mae'].tail(1)))
print()
print('Final Root Mean Average Error on validation set: {}'.format(round(rmse_fina
```

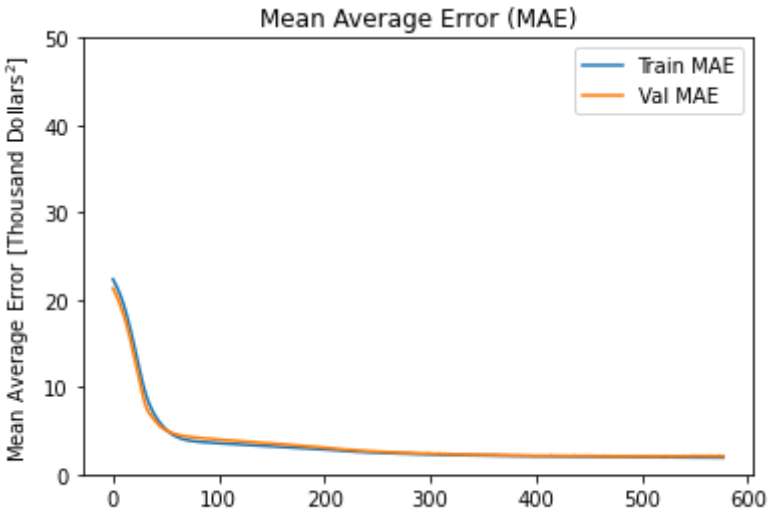
```
    Final Root Mean Average Error on validation set: 1.441
```

Now, let's plot the loss function measure on the training and validation sets. The validation set is used to prevent overfitting (learn more about it here). However, because our network is small, the training convergence without noticeably overfitting the data as the plot shows.

```
def plot_history():
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Mean Square Error [Thousand Dollars$^2$]')
    plt.plot(hist['epoch'], hist['mse'], label='Train MSE')
    plt.plot(hist['epoch'], hist['val_mse'], label = 'Val MSE')
    plt.legend()
    plt.title("Mean Square Error (MSE)")
    plt.ylim([0,50])

plot_history()
```




```
def plot_history():
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Mean Average Error [Thousand Dollars$^2$]')
    plt.plot(hist['epoch'], hist['mae'], label='Train MAE')
    plt.plot(hist['epoch'], hist['val_mae'], label = 'Val MAE')
    plt.legend()
    plt.title("Mean Average Error (MAE)")
    plt.ylim([0,50])

plot_history()
```



Next, compare how the model performs on the test dataset:

```
print(model.metrics_names)
```

```
['loss', 'mae', 'mse']
```

```
test_features_norm = (test_features - train_mean) / train_std
loss, mae, mse = model.evaluate(test_features_norm, test_labels)
print('Loss on test set: {}'.format(round(loss, 3)))
print('Mean Average Error on test set: {}'.format(round(mae, 3)))
print('Mean Square Error on test set: {}'.format(round(mse, 3)))
rmse = np.sqrt(mse)
print('Root Mean Square Error on test set: {}'.format(round(rmse, 3)))
```

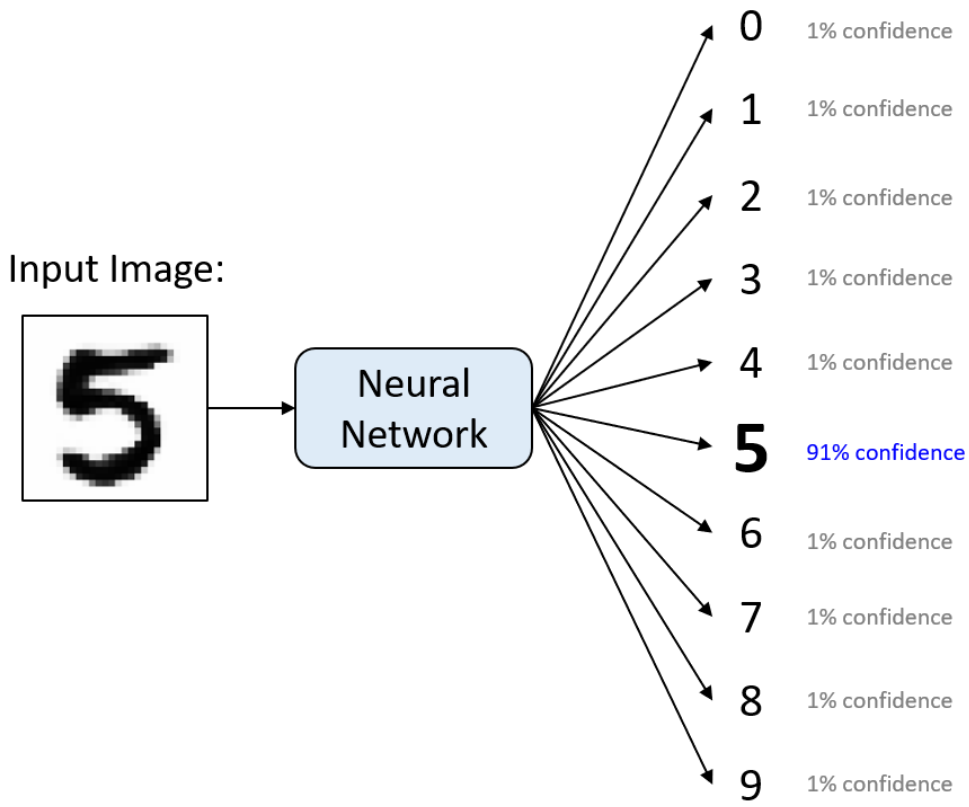
```
4/4 [==============================] - 0s 4ms/step - loss: 18.3065 - mae: 2.6
Loss on test set: 18.306
Mean Average Error on test set: 2.686
Mean Square Error on test set: 18.306
Root Mean Square Error on test set: 4.279
```

Compare the RMSE measure you get to the Kaggle leaderboard. An RMSE of 4.105 puts us in 24th place.

Part 2: Classification of MNIST Dreams with Convolutional Neural Networks

Next, let's build a convolutional neural network (CNN) classifier to classify images of handwritten digits in the MNIST dataset with a twist where we test our classifier on high-resolution hand-written digits from outside the dataset.

The MNIST dataset containss 70,000 grayscale images of handwritten digits at a resolution of 28 by 28 pixels. The task is to take one of these images as input and predict the most likely digit contained in the image (along with a relative confidence in this prediction):



Now, we load the dataset. The images are 28x28 NumPy arrays, with pixel values ranging between 0 and 255. The *labels* are an array of integers, ranging from 0 to 9.

```
(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.lo

# reshape images to specify that it's a single channel
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1)
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
```

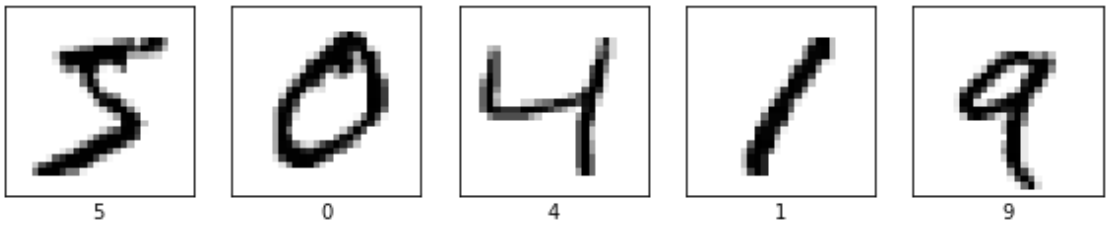
```
11493376/11490434 [==============================] - 0s 0us/step
```

We scale these values to a range of 0 to 1 before feeding to the neural network model. For this, we divide the values by 255. It's important that the *training set* and the *testing set* are preprocessed in the same way:

```
def preprocess_images(imgs): # should work for both a single image and multiple im
    sample_img = imgs if len(imgs.shape) == 2 else imgs[0]
    assert sample_img.shape in [(28, 28, 1), (28, 28)], sample_img.shape # make su
    return imgs / 255.0

train_images = preprocess_images(train_images)
test_images = preprocess_images(test_images)
```

Display the first 5 images from the *training set* and display the class name below each image. Verify that the data is in the correct format and we're ready to build and train the network.

```
plt.figure(figsize=(10,2))
for i in range(5):
    plt.subplot(1,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i].reshape(28, 28), cmap=plt.cm.binary)
    plt.xlabel(train_labels[i])
```



Build the model

Building the neural network requires configuring the layers of the model, then compiling the model. In many cases, this can be reduced to simply stacking together layers:

```
model = keras.Sequential()
# 32 convolution filters used each of size 3x3
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1
# 64 convolution filters used each of size 3x3
model.add(Conv2D(64, (3, 3), activation='relu'))
# choose the best features via pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
# randomly turn neurons on and off to improve convergence
model.add(Dropout(0.25))
```

```
# flatten since too many dimensions, we only want a classification output
model.add(Flatten())
# fully connected to get all relevant data
model.add(Dense(128, activation='relu'))
# one more dropout
model.add(Dropout(0.5))
# output a softmax to squash the matrix into output probabilities
model.add(Dense(10, activation='softmax'))
```

Before the model is ready for training, it needs a few more settings. These are added during the model's *compile* step:

- *Loss function* - measures how accurate the model is during training, we want to minimize this with the optimizer.
- *Optimizer* - how the model is updated based on the data it sees and its loss function.
- *Metrics* - used to monitor the training and testing steps. "accuracy" is the fraction of images that are correctly classified.

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320       
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496     
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0         
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 64)        0         
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 128)               1179776   
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290      
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

```
model.compile(optimizer=tf.optimizers.Adam(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the model

Training the neural network model requires the following steps:

1. Feed the training data to the model—in this example, the `train_images` and `train_labels` arrays.
2. The model learns to associate images and labels.
3. We ask the model to make predictions about a test set—in this example, the `test_images` array. We verify that the predictions match the labels from the `test_labels` array.

To start training, call the `model.fit` method—the model is "fit" to the training data:

```
history = model.fit(train_images, train_labels, epochs=5, validation_split = 0.1)
```

```
Epoch 1/5
1688/1688 [==============================] - 11s 3ms/step - loss: 0.3720 - ac
Epoch 2/5
1688/1688 [==============================] - 6s 3ms/step - loss: 0.0842 - acc
Epoch 3/5
1688/1688 [==============================] - 5s 3ms/step - loss: 0.0678 - acc
Epoch 4/5
1688/1688 [==============================] - 5s 3ms/step - loss: 0.0535 - acc
Epoch 5/5
1688/1688 [==============================] - 6s 3ms/step - loss: 0.0418 - acc
```

```
print(model.metrics_names)
```

```
['loss', 'accuracy']
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 98.68% on the training data.

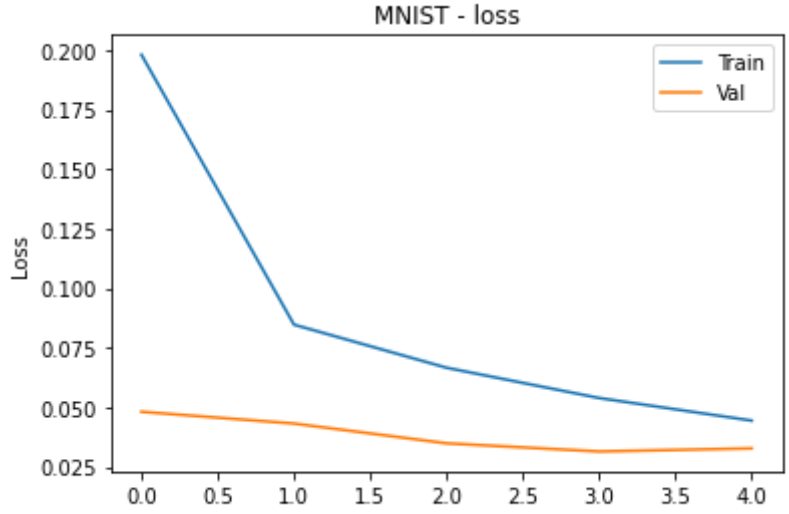
```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
```

```
hist.head()
```

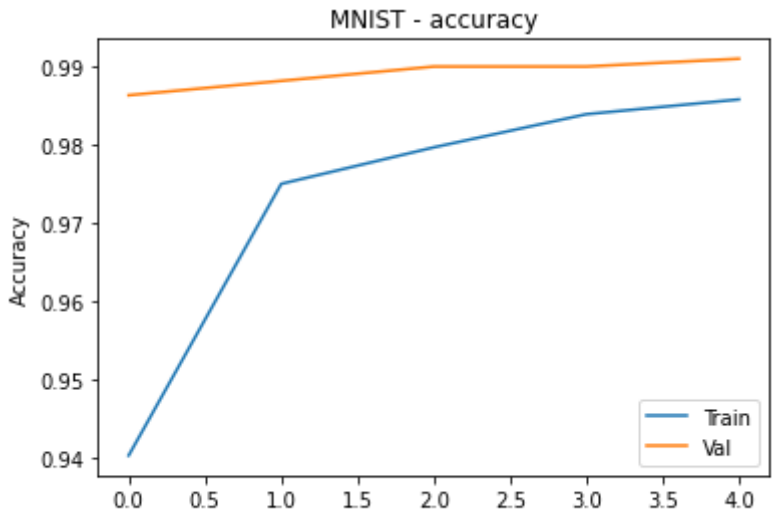
	loss	accuracy	val_loss	val_accuracy	epoch
0	0.198095	0.940278	0.048189	0.986333	0
1	0.084753	0.975000	0.043236	0.988167	1
2	0.066670	0.979667	0.034934	0.990000	2
3	0.053950	0.983907	0.031506	0.990000	3
4	0.044473	0.985796	0.032814	0.991000	4

```
def plot_history():
```

```
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.plot(hist['epoch'], hist['loss'], label='Train')
    plt.plot(hist['epoch'], hist['val_loss'], label = 'Val')
    plt.legend()
    plt.title("MNIST - loss")
    #plt.ylim([0,50])

plot_history()
```




```
def plot_history():
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.plot(hist['epoch'], hist['accuracy'], label='Train')
    plt.plot(hist['epoch'], hist['val_accuracy'], label = 'Val')
    plt.legend()
    plt.title("MNIST - accuracy")
    #plt.ylim([0,50])

plot_history()
```



Evaluate accuracy

Next, compare how the model performs on the test dataset:

```
print(test_images.shape)
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('MNIST - Test accuracy:', test_acc)
```

```
(10000, 28, 28, 1)
313/313 [==============================] - 1s 2ms/step - loss: 0.0291 - accur
MNIST - Test accuracy: 0.9905999898910522
```

Compare with ... after 5 epochs

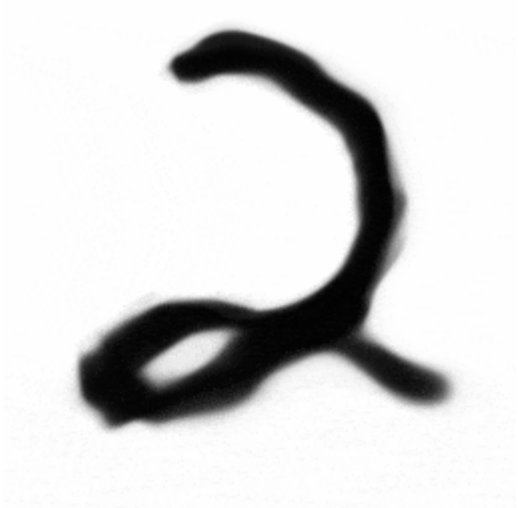
(10000, 28, 28, 1) 313/313 [==============================] - 1s 2ms/step - loss: 0.0302 - accuracy: 0.9910

Test accuracy: 0.9909999966621399

Often times, the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between training accuracy and test accuracy is an example of *overfitting*. In our case, the accuracy is better at 99.19%! This is, in part, due to successful regularization accomplished with the Dropout layers.

Make predictions

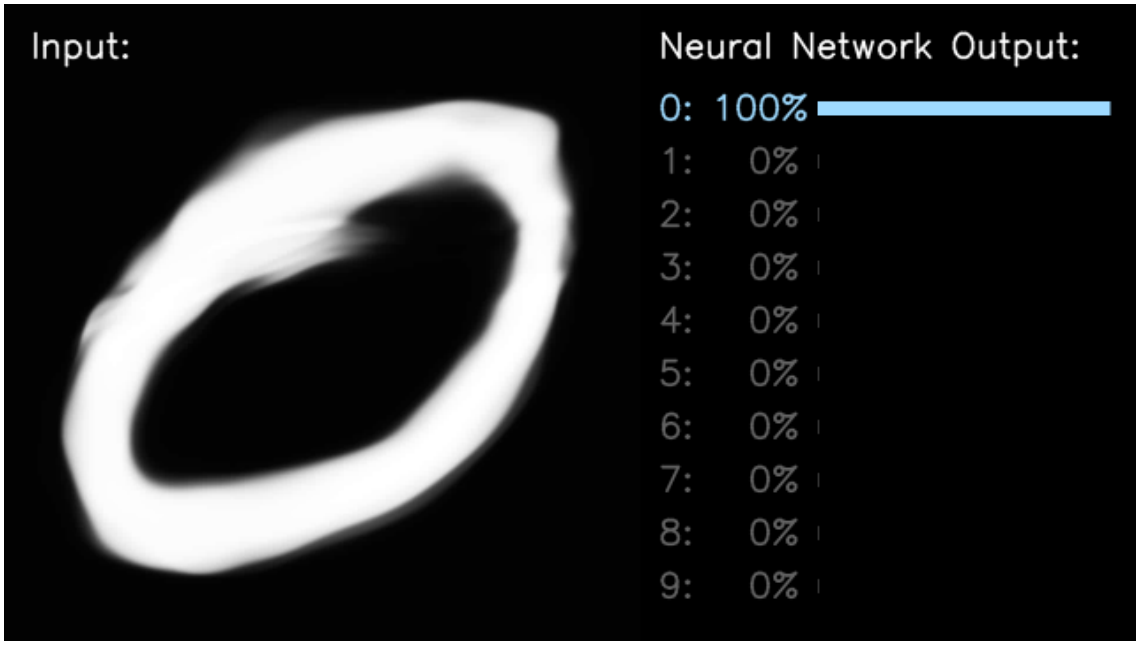
With the model trained, we can use it to make predictions about some images. Let's step outside the MNIST dataset for that and go with the beautiful high-resolution images generated by a mixture of CPPN, GAN, VAE. See great blog post by hardmaru for the source data and a description of how these morphed animations are generated:



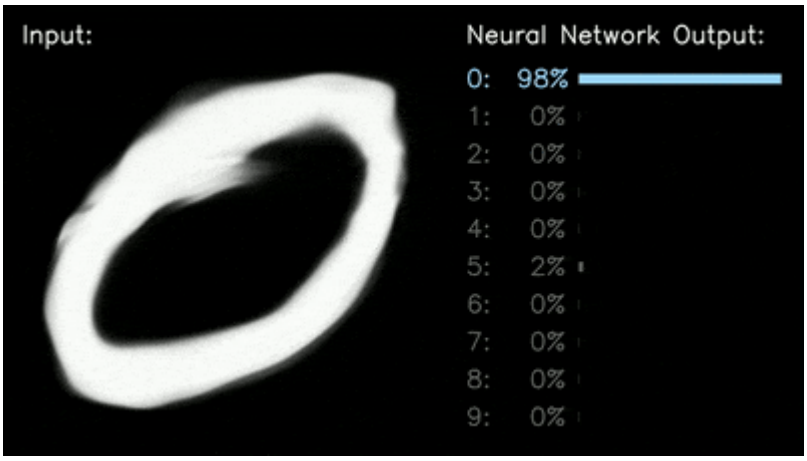
```
# Set common constants
this_repo_url = 'https://github.com/lexfridman/mit-deep-learning/raw/master/'
this_tutorial_url = this_repo_url + 'tutorial_deep_learning_basics'
```

```
mnist_dream_path = 'images/mnist_dream.mp4'
mnist_prediction_path = 'images/mnist_dream_predicted.mp4'

# download the video if running in Colab
if not os.path.isfile(mnist_dream_path):
    print('downloading the sample video...')
    vid_url = this_tutorial_url + '/' + mnist_dream_path

    mnist_dream_path = urllib.request.urlretrieve(vid_url)[0]

def cv2_imshow(img):
    ret = cv2.imencode('.png', img)[1].tobytes()
    img_ip = IPython.display.Image(data=ret)
    IPython.display.display(img_ip)

cap = cv2.VideoCapture(mnist_dream_path)
vw = None
frame = -1 # counter for debugging (mostly), 0-indexed

# go through all the frames and run our classifier on the high res MNIST images as
while True: # should 481 frames
    frame += 1
    ret, img = cap.read()
    if not ret: break

    assert img.shape[0] == img.shape[1] # should be a square
    if img.shape[0] != 720:
        img = cv2.resize(img, (720, 720))

    #preprocess the image for prediction
    img_proc = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_proc = cv2.resize(img_proc, (28, 28))
    img_proc = preprocess_images(img_proc)
    img_proc = 1 - img_proc # inverse since training dataset is white text with bl

    net_in = np.expand_dims(img_proc, axis=0) # expand dimension to specify batch
    net_in = np.expand_dims(net_in, axis=3) # expand dimension to specify number o

    preds = model.predict(net_in)[0]
    guess = np.argmax(preds)
    perc = np.rint(preds * 100).astype(int)

    img = 255 - img
    pad_color = 0
    img = np.pad(img, ((0,0), (0,1280-720), (0,0)), mode='constant', constant_valu
```

```
    line_type = cv2.LINE_AA
    font_face = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 1.3
    thickness = 2
    x, y = 740, 60
    color = (255, 255, 255)

    text = "Neural Network Output:"
    cv2.putText(img, text=text, org=(x, y), fontScale=font_scale, fontFace=font_fa
                color=color, lineType=line_type)

    text = "Input:"
    cv2.putText(img, text=text, org=(30, y), fontScale=font_scale, fontFace=font_f
                color=color, lineType=line_type)

    y = 130
    for i, p in enumerate(perc):
        if i == guess: color = (255, 218, 158)
        else: color = (100, 100, 100)

        rect_width = 0
        if p > 0: rect_width = int(p * 3.3)

        rect_start = 180
        cv2.rectangle(img, (x+rect_start, y-5), (x+rect_start+rect_width, y-20), c

        text = '{}: {:>3}%'.format(i, int(p))
        cv2.putText(img, text=text, org=(x, y), fontScale=font_scale, fontFace=fon
                    color=color, lineType=line_type)
        y += 60

    # if you don't want to save the output as a video, set this to False
    save_video = True

    if save_video:
        if vw is None:
            codec = cv2.VideoWriter_fourcc(*'DIVX')
            vid_width_height = img.shape[1], img.shape[0]
            vw = cv2.VideoWriter(mnist_prediction_path, codec, 30, vid_width_heigh
        # 15 fps above doesn't work robustly so we right frame twice at 30 fps
        vw.write(img)
        vw.write(img)

    # scale down image for display
    img_disp = cv2.resize(img, (0,0), fx=0.5, fy=0.5)
    cv2_imshow(img_disp)
    IPython.display.clear_output(wait=True)

cap.release()
if vw is not None:
    vw.release()
```



The above shows the prediction of the network by choosing the neuron with the highest output. While the output layer values add 1 to one, these do not reflect well-calibrated measures of "uncertainty". Often, the network is overly confident about the top choice that does not reflect a learned measure of probability. If everything ran correctly you should get an animation like this:



Acknowledgements

The contents of this tutorial is based on and inspired by the work of TensorFlow team), our MIT Human-Centered AI team, and individual pieces referenced in the MIT Deep Learning course slides.

Colab paid products - Cancel contracts here

Нейронні мережі

Лекція_04

********** **********

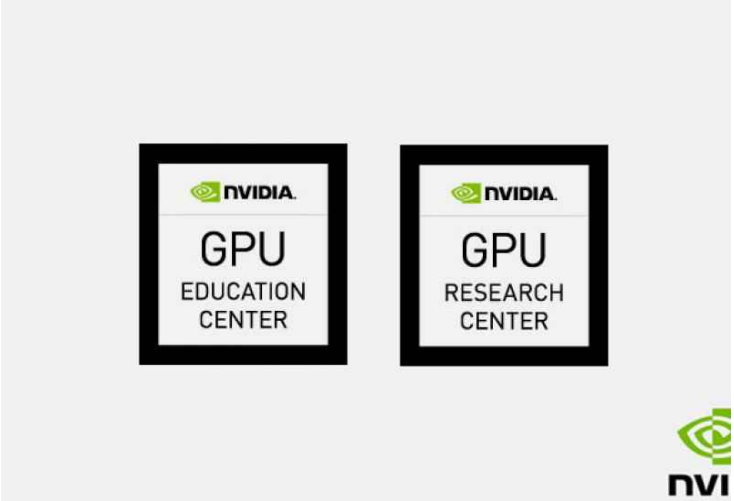
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 04- Нейронні мережі в TensorFlow

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1 -ЦП
Робочий процес – приклад у Colab – версія ЦП
https://drive.google.com/file/d/16VOFnz5QOaWFsPvp4NBsUkIrPfnoLTjB/view?usp=sharing

ДЕМО1 - графічний процесор
Робочий процес - приклад у Colab -ГВерсія PU
https://drive.google.com/file/d/15u94L3o26NuAakCe9CNazM22X_6VSkqE/view?usp=sharing

ДЕМО1 - ТПУ
Робочий процес - приклад у Colab -ТВерсія PU
https://drive.google.com/file/d/1U5oXDZdChzX0RFppNhbiAciaJgqjr5F_/view?usp=sharing

ДЕМО2
Приклад робочого процесу із зовнішніми даними, тензорною панеллю, двійковою класифікацією
https://drive.google.com/file/d/1pqTVTUKgAwpiWbpKBdfiXXD2LXOvVT1B/view?usp=sharing

ДЕМО3А
Приклад робочого процесу з Keras,TFDS,CNN_2_4_6, cifar10,якласифікація маг
https://drive.google.com/file/d/1iMmbvklyNDbRtNiu2J486dM380eoDWgH/view?usp=sharing

ДЕМО3Б
Приклад робочого процесу cМодель відTF2 концентратор, MobileNetV2,ImageNet, якласифікація маг
https://drive.google.com/file/d/1w2XcgucVjybTwKVjJFK2kjaUWNr0lvG4/view?usp=sharing

Нейронні мережі
-

Лекція 04. Глибинні нейронні мережі у TensorFlow

(на основі (C) F.Colliet, Lex Fridman, ... та інших)

Зміст

Рекомендовані джерела — Книги

Книги (наукові):

Гудфелло, І., Бенгіо, Ю., Курвіль, А. (2016). Глибоке навчання. Кембридж: MIT Press

Цитовано в 23692 джерелах.

Книги (з кодами на github):

Алан Фонтейн (2018). Освоєння інтелектуальної аналітики scikit-learn і TensorFlow. Packt Publishing.

Танай Агравал (2021). Оптимізація гіперпараметрів у машинному навчанні: зробіть своє машинне навчання глибоким. Ефективніші моделі навчання. Apress

Рекомендовані джерела — папери

Imagenet

Pan, SJ, & Yang, Q. (2009). Опитування щодо трансферного навчання. IEEE Transactions on knowledge and data engineering, 22(10), 1345-1359.

Хінтон, Г., Віньялс, О., і Дін, Дж. (2015). Перегонка знань в нейронній мережі. препринт arXiv arXiv:1503.02531.

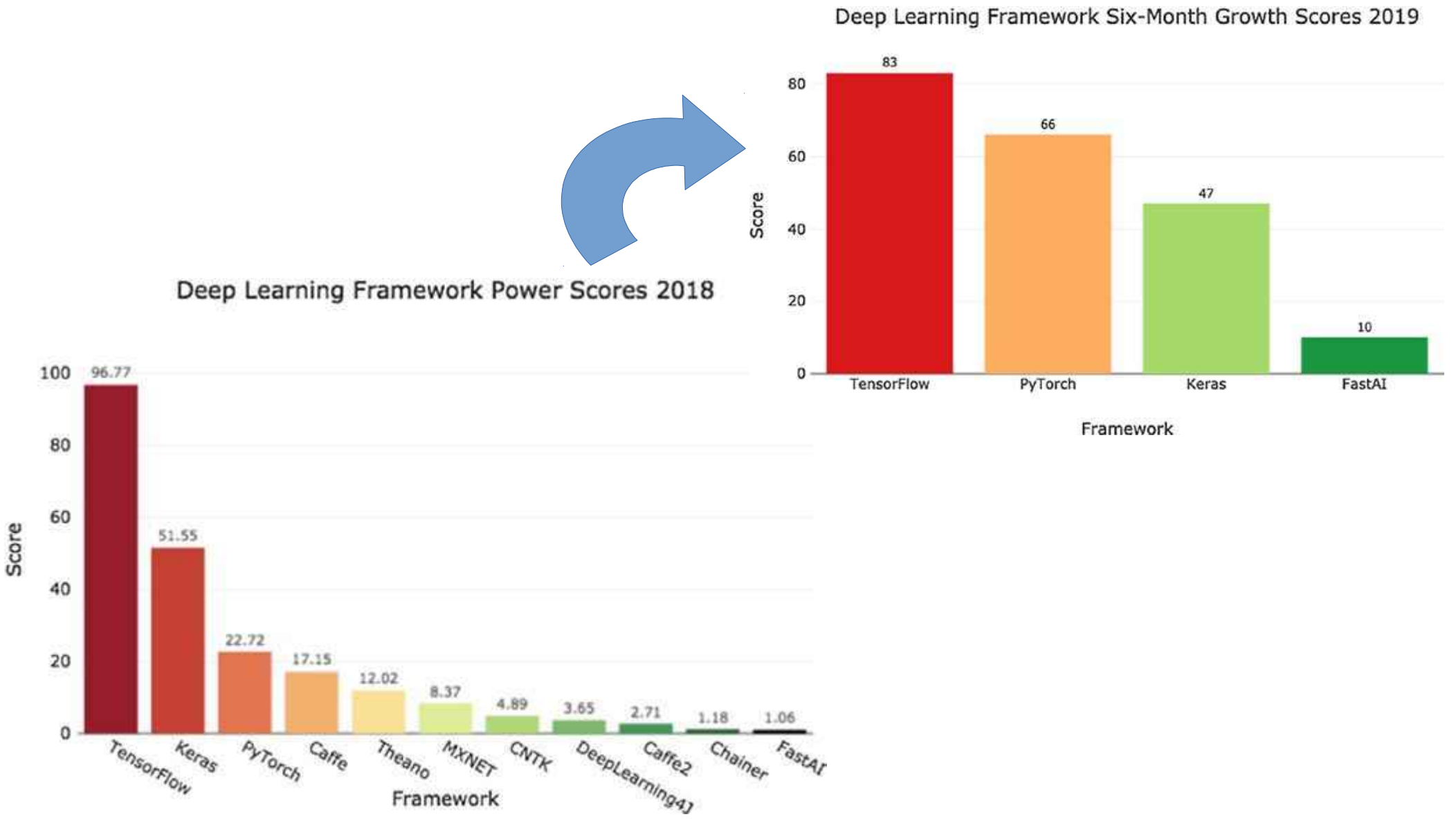
Конечний Дж., Макмехан Х.Б., Ю. Ф.Х., Ріхтарік П., Сурeш А.Т. & Бекон, Д. (2016). Інтегроване навчання: стратегії вдосконалення ефективність спілкування. препринт arXiv arXiv:1610.05492.

DL Frameworks

-

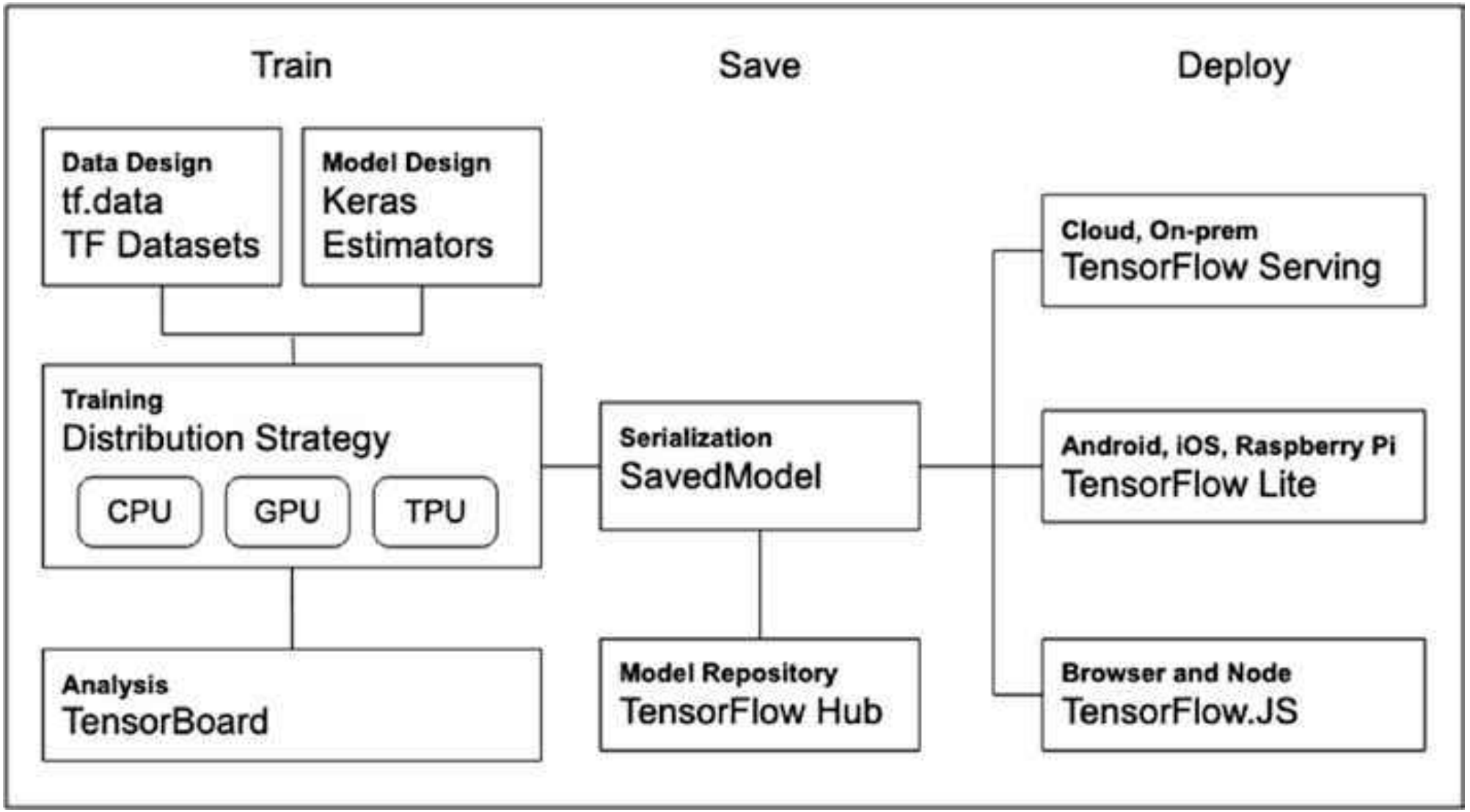
основи

DL Frameworks —Еволюція



https://www.kdnuggets.com/2019/05/which-deep-learning-framework-growing-

DL Framework — Екосистема (Приклад Tensorflow)



DL Framework —компоненти (Приклад Tensorflow)

Обладнання **Навколишнє середовище** (локальні, хмарні, периферійні обчислення, ...):

-ЦП

-GPU

- ТПУ

-Edge Computing Devices...

програмне забезпечення **Навколишнє середовище** (локальні, хмарні, периферійні обчислення, ...):

Операційна система: macOS (>=10.12.6), Ubuntu (>=16.04), Windows (>=7), Raspbian (>=9.0), ... **Мо**
програмування: C, C++, C#, Java, Go, Julia, Ruby, Scala, але ... Python 3

Бібліотеки: ... численні ...

DL Frameworks

-

робочий процес

DL робочий процес

(Tensorflowy Google Cloudприклад)

-**Налаштування обладнання** **Навколишнє середовище (**у Google VM**)**:

- вибрати**Тип виконання**: CPU, GPU, TPU —**ДЕМО 1**.

Програмне забезпечення для налаштування **Навколишнє середовище**:

Операційна система: Ubuntu (вже попередньо встановлено).

Мова програмування: Python 3 (вже попередньо встановлено)

Бібліотеки: Python-libs (багато з них уже попередньо встановлено)

Налаштувати **(отримати) набір даних**: локальний, хмарний (AWS, GC, ... Kaggle, ...)

отримати **(визначити або завантажити) модель**: локальний, TF Hub, хмара, ...

Скомпілювати(налаштувати гіперпараметри) модель: функція втрат, метод оптимізації, метрики, тривалість, зворотні виклики, ...

поїзді**перевірити**модель ->**Прогнозування->виробництво**?**Ще ні!**

ДЕМО 1: робочий процес у TF2

DEMO_1_Workflow_Example_CPU.ipynb
DEMO_1_Workflow_Example_GPU.ipynb
DEMO_1_Workflow_Example_TPU.ipynb

ДЕМО 2:

Як контролювати робочий процес у TF2

DEMO_2_External_Data_Tensorboard_Binary_Classification_Example.ipynb

Робочий процес DL — типи навчання/навчання

- **Навчання з нуля**:
- від**випадковий ініціал**параметри (вага, ...)
- від**раніше навчені**спроби.

- **Трансфер навчання**—полягає в удосконаленні навчання новому завданню шляхом передачі знань із суміжного завдання, яке вже вивчено.

- **Перегонка знань**—навчання - це процес перенесення знань з великої моделі на меншу.

- **Федеративне навчання**(також відомий як спільне навчання) — це техніка машинного навчання, яка навчає алгоритм на кількох децентралізованих периферійних пристроях або серверах, що містять локальні зразки даних без обміну ними.

Робочий процес DL — типи навчання/навчання

— з нуля ... на наборах даних

Навчання з нулявід**випадковий ініціал**параметри (ваги, ...) є дуже ресурсомістким завданням.

Навіть для**MNIST**(**60 тис**зображень), потрібен час, щоб навчити модель до точності 80-90%. Для більш високої точності, буде потрібно більше зображень. DNN **вчиться кращезвищий обсяг**даних.

неMNIST, FashionMNIST, ...

https://www.kaggle.com/yoctoman/graffiti-st-sophia-cathedral-київ

CIFAR10, CIFAR100, ...

Microsoft Common Objects in Context (COCO),

PASCAL Visual Object Classes (PASCAL VOC),

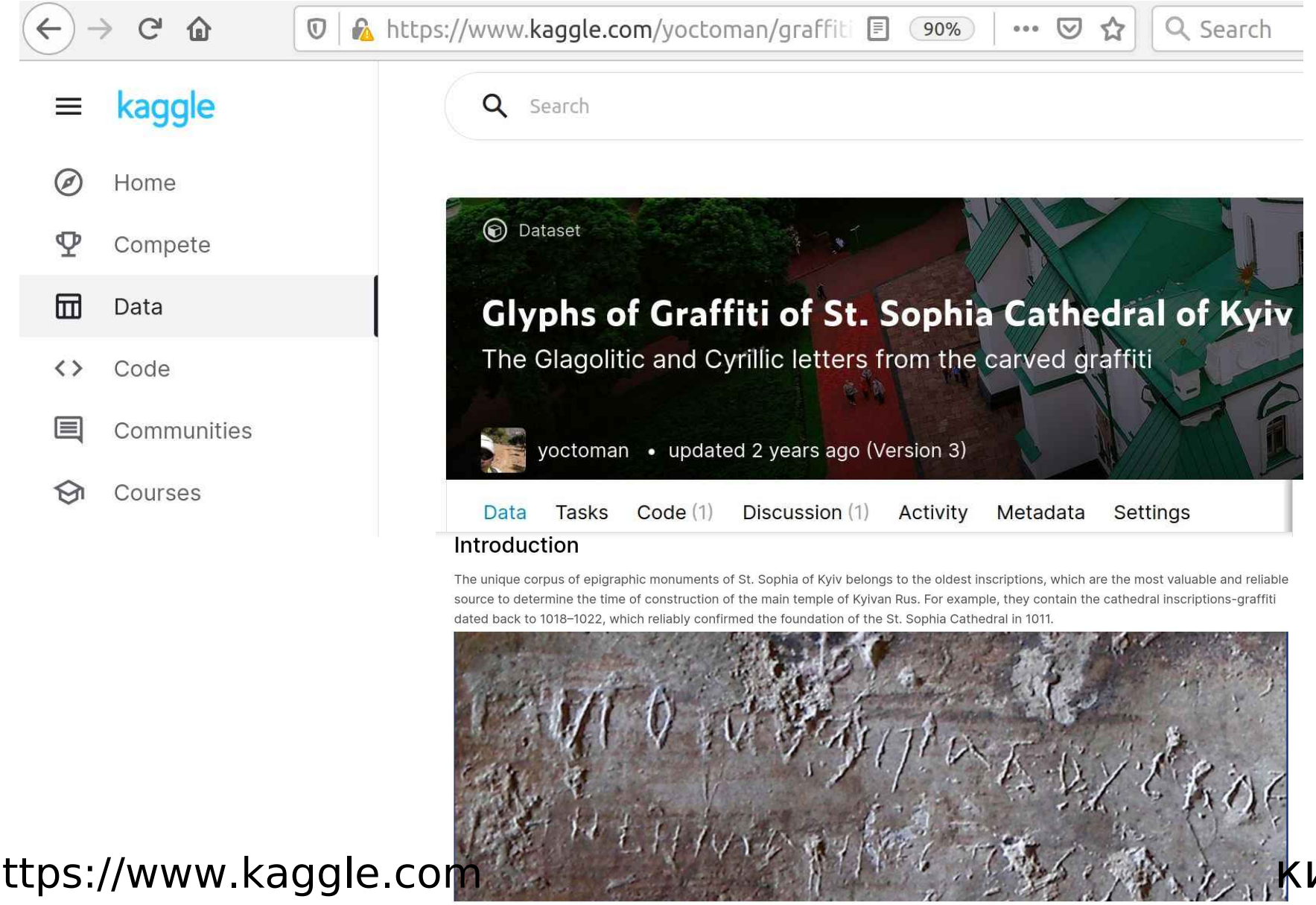
. . .

ImageNet!!!

Робочий процес DL — типи навчання/навчання

— з нуля ... на наборах даних

Ще один датасет зі старовинними кириличними літерами ... з Києва



https://www.kaggle.com

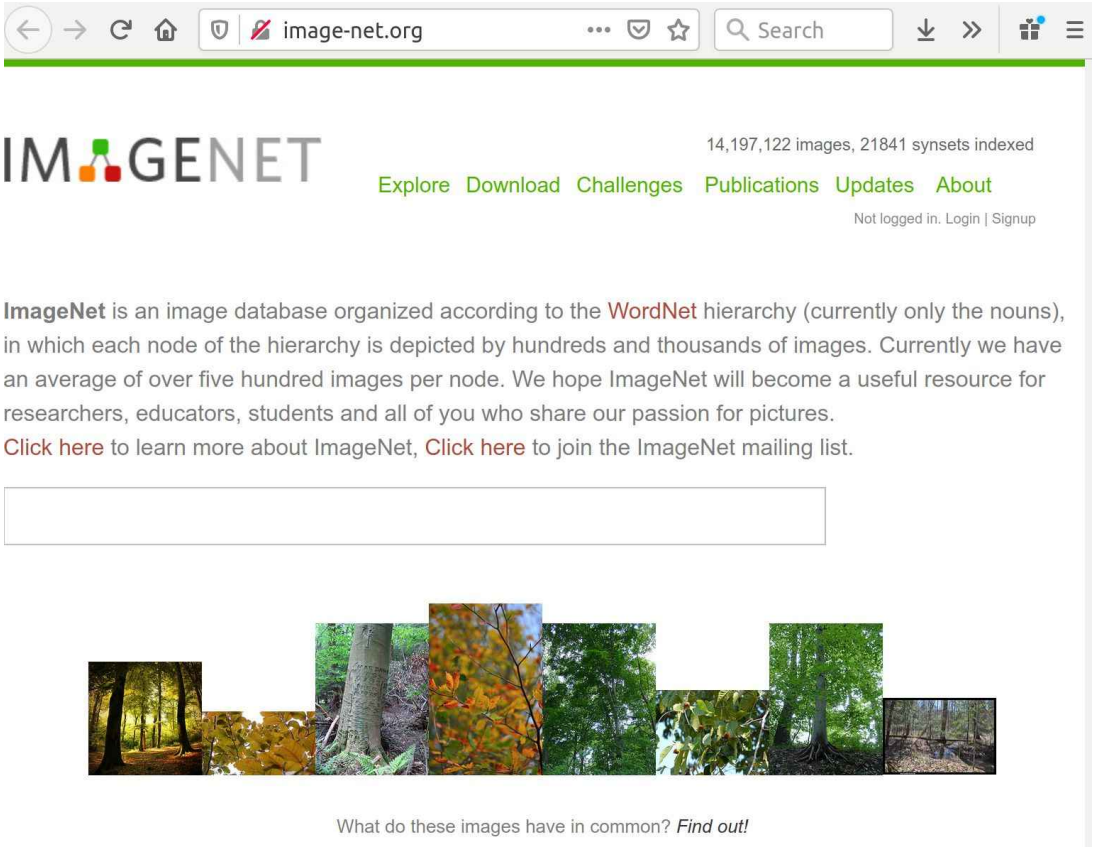
КИЇВ

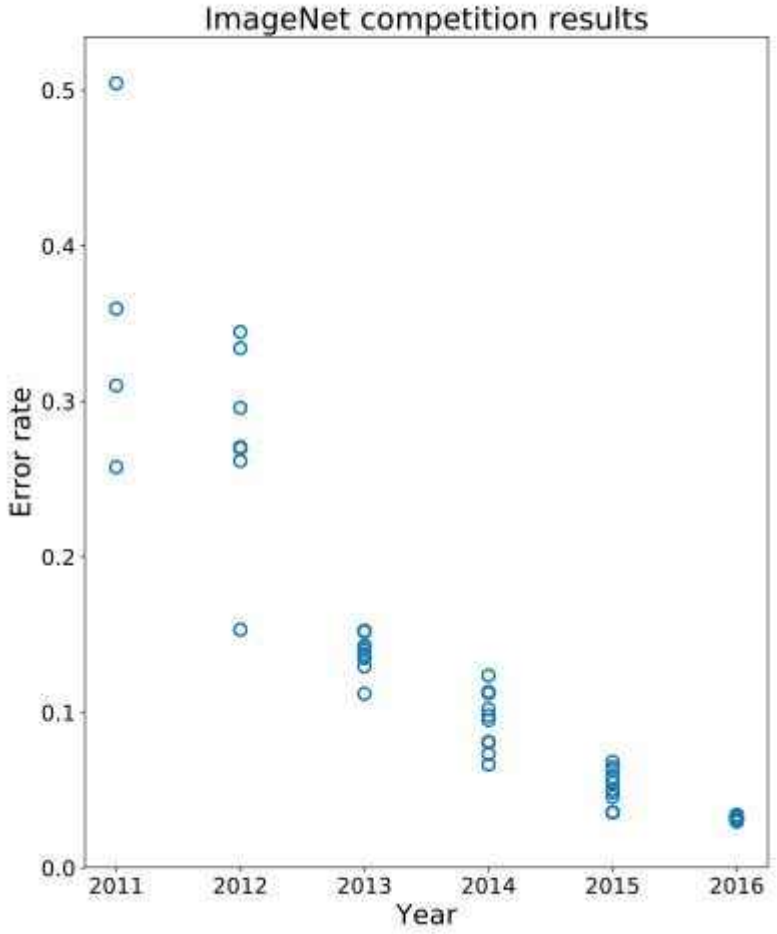
Робочий процес DL — типи навчання/навчання

— з нуля … на Imagenet

ImageNet(http://image-net.org):**14,197,122**зображення в **21,841**підкатегорії в**27**піддерева.

Щоб класифікувати зображення в ImageNet, було розроблено багато моделей ML/DL. У 2017 році одна модель досягла рівня помилок у 2,3%





Робочий процес DL — типи навчання/навчання

— Передача навчання

Transfer learning — це вдосконалення навчання в новому завдання через передачу знань із пов’язаного завдання що вже навчилися.

1. Використання попередньо підготовленої моделі

2. Навчання моделі повторному використанню

Для вирішення завдання A у вас є обмежені дані для навчання DNN. Один із способів: знайти пов’язане завдання B із великою кількістю даних.

Навчіть DNN на завдання B і використовуйте модель як вихідну бал за розв'язання задачі A.

3. Витяг функцій

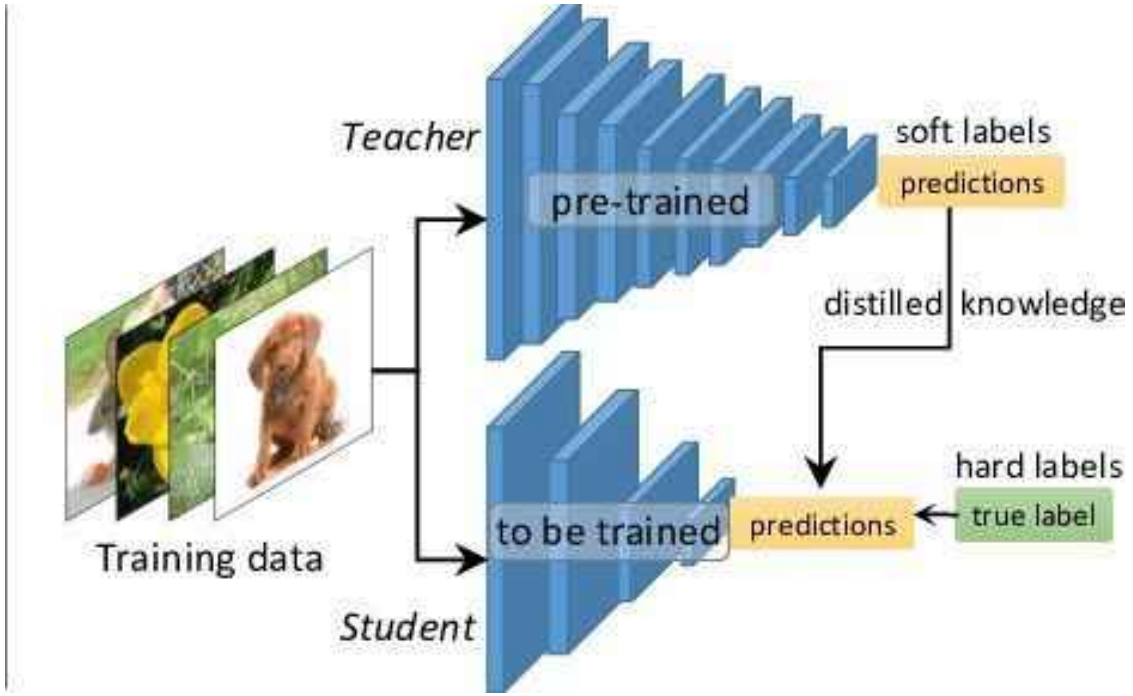
Інший підхід полягає у використанні DNN для пошуку найкращого **представництво**(найважливіші особливості) вашої проблеми, і використовувати їх.

Робочий процес DL — типи навчання/навчання -

Дистиляція знань

Маленьку модель навчають імітувати попередньо підготовлену більшу модель (або групу моделей). Цю навчальну установку іноді називають «викладач-учень», де**великий**модель є**викладач**і **маленький**модель є**студент**.

Було навіть помічено, що класифікатори**навчитися**багато**швидше**і більше**надійно**якщо тренуватися з**виходи**іншого класифікатора як **м'який**мітки, а не з жорстких міток (основні дані правдивості).

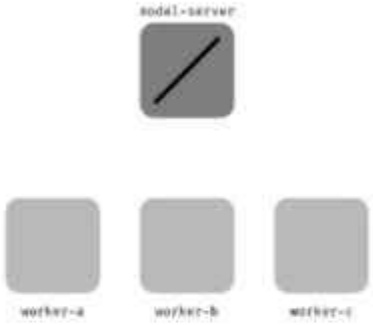
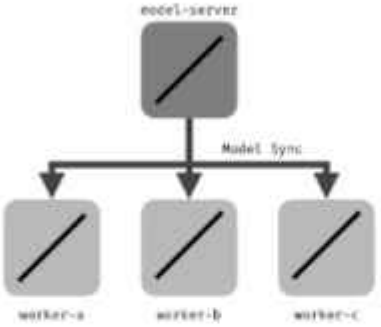
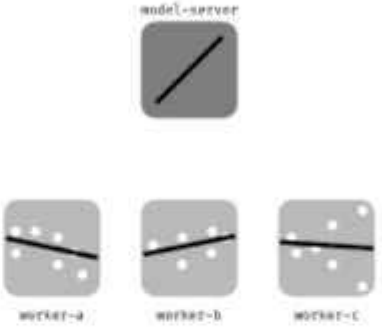
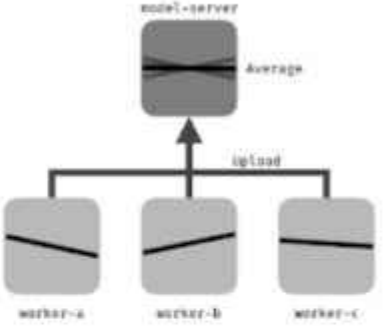


Робочий процес DL — типи навчання/навчання

— Федеративне навчання

. . . є**у контрасті**до

- традиційні**централізований**методи навчання, де всі **місцевий** набори даних є**завантажено на один сервер**,
- і деякі**класична децентралізована**методи навчання, де локальні вибірки даних однаково розподілені.

Step 1	Step 2	Step 3	Step 4
			
Central server chooses a statistical model to be trained	Central server transmits the initial model to several nodes	Nodes train the model locally with their own data	Central server pools model results and generate one global mode without accessing any data

DL робочий процес

-

Передача навчання

ДЕМО 3А:
Навчання з нуля в TF2

DEMO_3A_Model_from_TF2_Keras_CNN_2_4_6_cifar10_imageClassification.ipynb

DL Workflow —Передача навчання

Трансфер навчання

— в загальному сенсі : покращення навчання в новому завданні шляхом передачі знань із пов’язаного завдання що вже вивчено,

— у контексті ML/DL : це важлива техніка **передача знань**від одного до іншого**Завдання ML/DL**.

приклади:

-У розробці програмного забезпечення : люди використовують бінарні бібліотеки для повторно використовувати код.

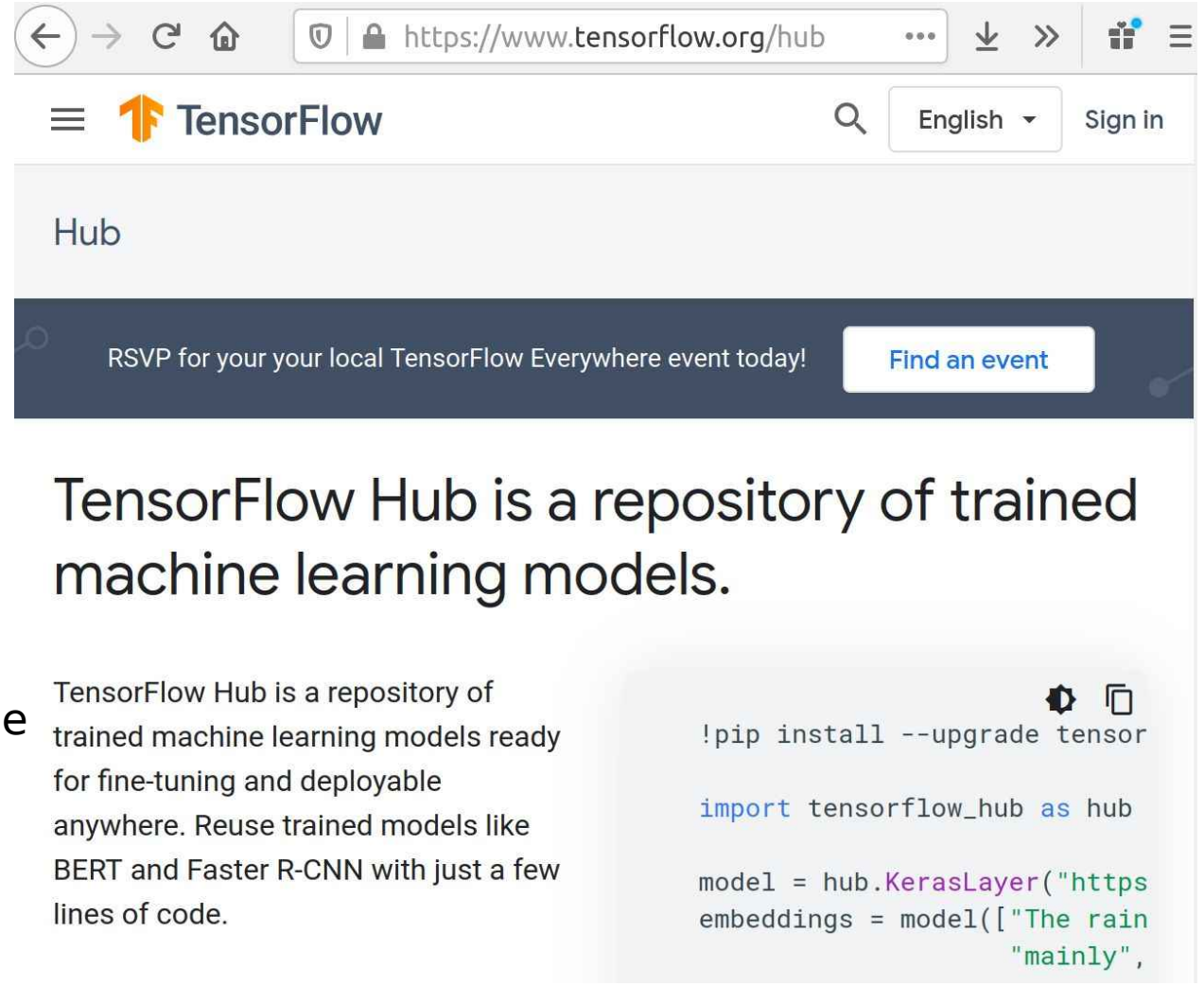
-У ML/DL : навчені моделі містять**алгоритми**, **даних**, **потужність обробки**, і**предметні знання експерта**. Все це має бути**переданий**до**новий** модель. Саме це забезпечує трансферне навчання.

Передача навчання —Джерела моделей

Попередньо навчені моделі можуть можна знайти всюди:
- від вашого**колеги**,
- **github**,
- **Kaggle**,

. . .

але**Концентратор Tensorflow**це найширший і найпростіший джерело попереднього навчання і надійний DNN моделі.



https://www.tensorflow.org/hub

ДЕМО 3B:

Перенести навчання в TF2

DEMO_3B_Model_from_TF2_Hub_MobileNetV2_ImageNet_imageClassification.ipy

DL Workflow - Example at Colab - CPU version

Let's consider the components and workflow of DL

Setup Hardware Environment:

selectby menu command **Runtime -> Change Runtime Type** above:

- None (CPU),
- GPU,
- TPU

Setup Software Environment

Import required libraries

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Set up (get) dataset

Generate data here

```
number_of_datapoints = 1000

# for reproducibility of results
#  every time you call the numpy's other random function, the result will be the s
np.random.seed(1941)

# generate random x values in the range -5 to +5
x = np.random.uniform(low = -5 , high = 5 , size = (number_of_datapoints, 1))
# generate random y values in the range -5 to +5
y = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
# generate some random error in the range -1 to +1
noise = np.random.uniform(low =-1 , high =1, size = (number_of_datapoints, 1))
#z = 7 * x + 6 * y + 5 + noise

a = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
z = 7 * x + 6 * y + 3 * a + 5 + noise
```

Print x, y and z sample values for manual verification

```
x[:5,:].round(2)
```

```
array([[-1.51],
       [ 0.06],
       [-0.87],
       [-1.67],
       [-2.13]])
```

```
y[:2,:].round(2)
```

```
array([[-3.76],
       [ 2.25]])
```

```
z[:2,:]
```

```
array([[-21.85820882],
       [ 18.21713423]])
```

```
z[:2,:].round(2)
```

```
array([[-21.86],
       [ 18.22]])
```

```
z.shape
```

```
(1000, 1)
```

Stack x and y arrays for inputting to neural network

```
#input = np.column_stack((x,y))
input = np.column_stack((x,y,a))
```

Print few values of input array for demonstration purpose.

```
input[:2,:].round(2)
```

```
array([[-1.51, -3.76,  1.89],
       [ 0.06,  2.25, -0.27]])
```

Get (define or load) model

Define by Keras tools here

Create a sequential model consisting of single layer with a single neuron.

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1)])
```

Compile (configure hyperparameters) model

Compile the model with the spcified optimizier, loss function and error metrics.

```
model.compile(optimizer = 'sgd' , loss = 'mean_squared_error' , metrics = ['mse']
```

Train and validate the model

Create callbacks (for various aims - to save training history here)

Import History module to record loss and accuracy on each epoch during training

```
from tensorflow.keras.callbacks import History
history = History()
```

Start training/validation ...

The first parameter specifies the input, the second specifies the corresponding target values. The epochs defines the number of iterations in training. The validation split specifies what percentage of input data would be used for validating the model. The callbacks parameter specifies the variable were the collected statistics would be accumulated.

```
%%time
model.fit(input, z , epochs = 15 , verbose = 1, validation_split=0.2, callbacks=[h
```

```
Epoch 1/15
25/25 [==============================] - 1s 13ms/step - loss: 126.1613 - mse:
Epoch 2/15
25/25 [==============================] - 0s 8ms/step - loss: 6.6176 - mse: 6.
Epoch 3/15
25/25 [==============================] - 0s 6ms/step - loss: 2.5691 - mse: 2.
Epoch 4/15
25/25 [==============================] - 0s 4ms/step - loss: 1.1390 - mse: 1.
Epoch 5/15
25/25 [==============================] - 0s 5ms/step - loss: 0.6109 - mse: 0.
```

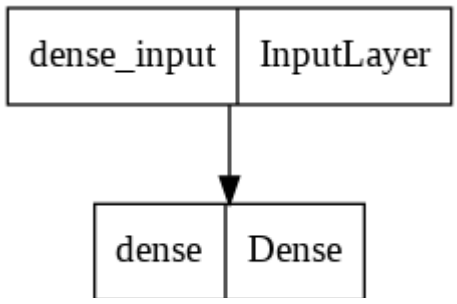
```
Epoch 6/15
25/25 [==============================] - 0s 3ms/step - loss: 0.4211 - mse: 0.
Epoch 7/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3502 - mse: 0.
Epoch 8/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3252 - mse: 0.
Epoch 9/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3162 - mse: 0.
Epoch 10/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3140 - mse: 0.
Epoch 11/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3109 - mse: 0.
Epoch 12/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3109 - mse: 0.
Epoch 13/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3114 - mse: 0.
Epoch 14/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3107 - mse: 0.
Epoch 15/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3122 - mse: 0.
CPU times: user 1.86 s, sys: 98.7 ms, total: 1.96 s
Wall time: 3.16 s
<keras.callbacks.History at 0x7fbba31aa6d0>
```

Visualize the model structure

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 1)                 4

=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

```
tf.keras.utils.plot_model(model, 'simple_model.png')
```



Check the available metrics

Print keys in the history just to know their names. These will be used for plotting the metrics.

```
print(history.history.keys())
```

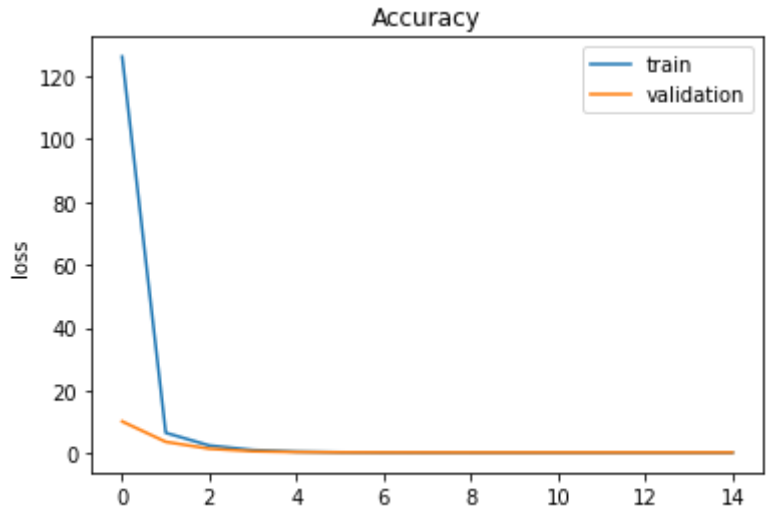
```
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

Visualize the training/validation history

Loss

Plot the loss metric on both training and validation datasets.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Accuracy')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

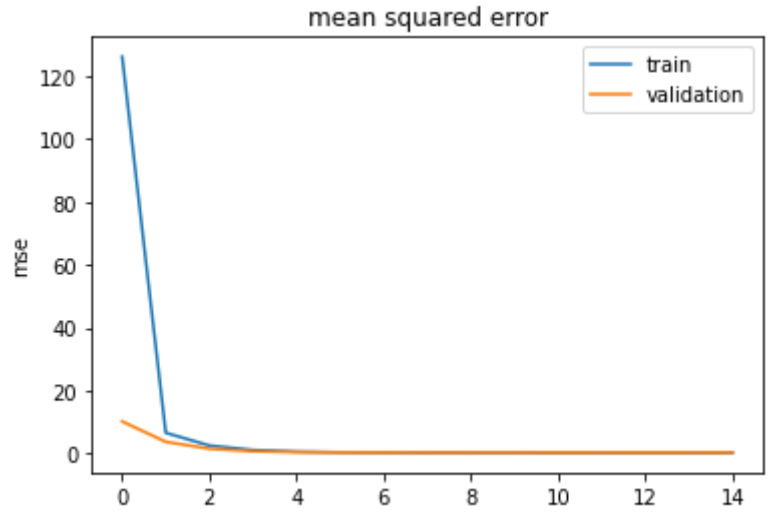


Mean squared error (MSE)

Plot the mean squared error on both training and validation datasets.

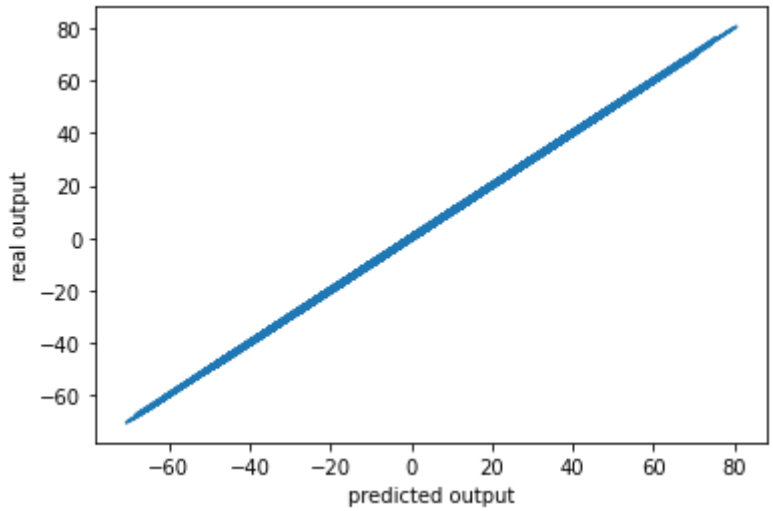
```
plt.plot(history.history['mse'])
plt.plot(history.history['val_mse'])
plt.title('mean squared error')
plt.ylabel('mse')
plt.xlabel('epochs')
```

```
plt.legend(['train' , 'validation'] , loc = 'upper right')
plt.show()
```



Predict and visualize prediction along with the "ground truth" values

```
plt.plot(np.squeeze(model.predict_on_batch(input)), np.squeeze(z))
plt.xlabel('predicted output')
plt.ylabel('real output')
plt.show()
```



Predict any value and ...

```
#print("Predicted z for x=2, y=3 ---> ", model.predict([[2,3]]).round(2))
print("Predicted z for x=2, y=3, a=4 ---> ", model.predict([[2,3,4]]).round(2))
```

```
Predicted z for x=2, y=3, a=4 --->  [[49.]]
```

... compare with the **ground truth** value

```
#checking from equation
#z = 7*x + 6*y + 5
#print("Expected output: ", 7*2 + 6*3 + 5)

#z = 7*x + 6*y + 3*4 + 5
print("Expected output: ", 7*2 + 6*3 + 3*4 + 5)
```

```
Expected output:  49
```

DL Workflow - Example at Colab - **GPU** version

Let's consider the components and workflow of DL

Setup Hardware Environment:

selectby menu command **Runtime -> Change Runtime Type** above:

- None (CPU),
- GPU,
- TPU

Setup Software Environment

Import required libraries

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Set up (get) dataset

Generate data here

```
number_of_datapoints = 1000

# for reproducibility of results
#  every time you call the numpy's other random function, the result will be the s
np.random.seed(1941)

# generate random x values in the range -5 to +5
x = np.random.uniform(low = -5 , high = 5 , size = (number_of_datapoints, 1))
# generate random y values in the range -5 to +5
y = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
# generate some random error in the range -1 to +1
noise = np.random.uniform(low =-1 , high =1, size = (number_of_datapoints, 1))
#z = 7 * x + 6 * y + 5 + noise

a = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
z = 7 * x + 6 * y + 3 * a + 5 + noise
```

Print x, y and z sample values for manual verification

```
x[:5,:].round(2)
```

```
array([[-1.51],
       [ 0.06],
       [-0.87],
       [-1.67],
       [-2.13]])
```

```
y[:2,:].round(2)
```

```
array([[-3.76],
       [ 2.25]])
```

```
z[:2,:]
```

```
array([[-21.85820882],
       [ 18.21713423]])
```

```
z[:2,:].round(2)
```

```
array([[-21.86],
       [ 18.22]])
```

```
z.shape
```

```
(1000, 1)
```

Stack x and y arrays for inputting to neural network

```
#input = np.column_stack((x,y))
input = np.column_stack((x,y,a))
```

Print few values of input array for demonstration purpose.

```
input[:2,:].round(2)
```

```
array([[-1.51, -3.76,  1.89],
       [ 0.06,  2.25, -0.27]])
```

Get (define or load) model

Define by Keras tools here

Create a sequential model consisting of single layer with a single neuron.

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1)])
```

Compile (configure hyperparameters) model

Compile the model with the spcified optimizier, loss function and error metrics.

```
model.compile(optimizer = 'sgd' , loss = 'mean_squared_error' , metrics = ['mse']
```

Train and validate the model

Create callbacks (for various aims - to save training history here)

Import History module to record loss and accuracy on each epoch during training

```
from tensorflow.keras.callbacks import History
history = History()
```

Start training/validation ...

The first parameter specifies the input, the second specifies the corresponding target values. The epochs defines the number of iterations in training. The validation split specifies what percentage of input data would be used for validating the model. The callbacks parameter specifies the variable were the collected statistics would be accumulated.

```
%%time
model.fit(input, z , epochs = 15 , verbose = 1, validation_split=0.2, callbacks=[h
```

```
Epoch 1/15
25/25 [==============================] - 4s 17ms/step - loss: 101.9569 - mse:
Epoch 2/15
25/25 [==============================] - 0s 5ms/step - loss: 6.4287 - mse: 6.
Epoch 3/15
25/25 [==============================] - 0s 5ms/step - loss: 2.5595 - mse: 2.
Epoch 4/15
25/25 [==============================] - 0s 6ms/step - loss: 1.1270 - mse: 1.
Epoch 5/15
25/25 [==============================] - 0s 9ms/step - loss: 0.6111 - mse: 0.
```

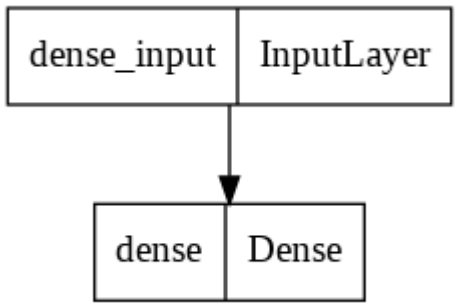
```
Epoch 6/15
25/25 [==============================] - 0s 9ms/step - loss: 0.4193 - mse: 0.
Epoch 7/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3480 - mse: 0.
Epoch 8/15
25/25 [==============================] - 0s 7ms/step - loss: 0.3245 - mse: 0.
Epoch 9/15
25/25 [==============================] - 0s 11ms/step - loss: 0.3154 - mse: 0
Epoch 10/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3133 - mse: 0.
Epoch 11/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3109 - mse: 0.
Epoch 12/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3114 - mse: 0.
Epoch 13/15
25/25 [==============================] - 0s 6ms/step - loss: 0.3093 - mse: 0.
Epoch 14/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3100 - mse: 0.
Epoch 15/15
25/25 [==============================] - 0s 8ms/step - loss: 0.3095 - mse: 0.
CPU times: user 3.14 s, sys: 401 ms, total: 3.54 s
Wall time: 6.32 s
<keras.callbacks.History at 0x7f7500095090>
```

Visualize the model structure

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 1)                 4

=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

```
tf.keras.utils.plot_model(model, 'simple_model.png')
```



Check the available metrics

Print keys in the history just to know their names. These will be used for plotting the metrics.

```
print(history.history.keys())
```

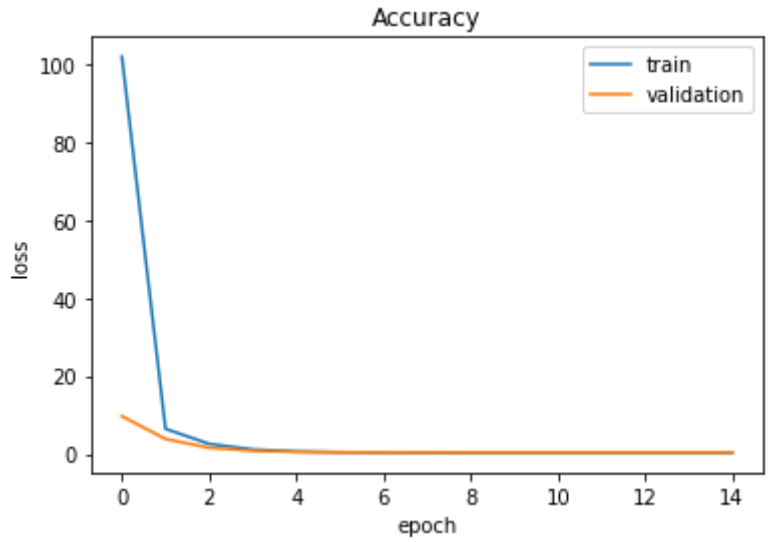
```
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

Visualize the training/validation history

Loss

Plot the loss metric on both training and validation datasets.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Accuracy')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

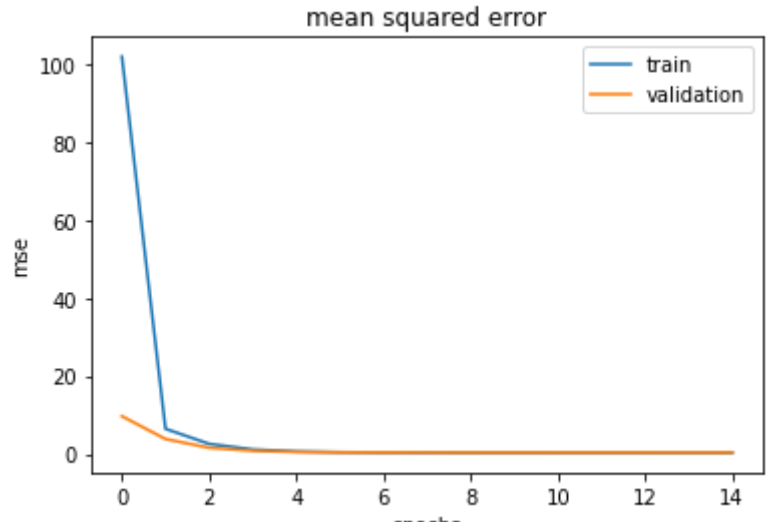


Mean squared error (MSE)

Plot the mean squared error on both training and validation datasets.

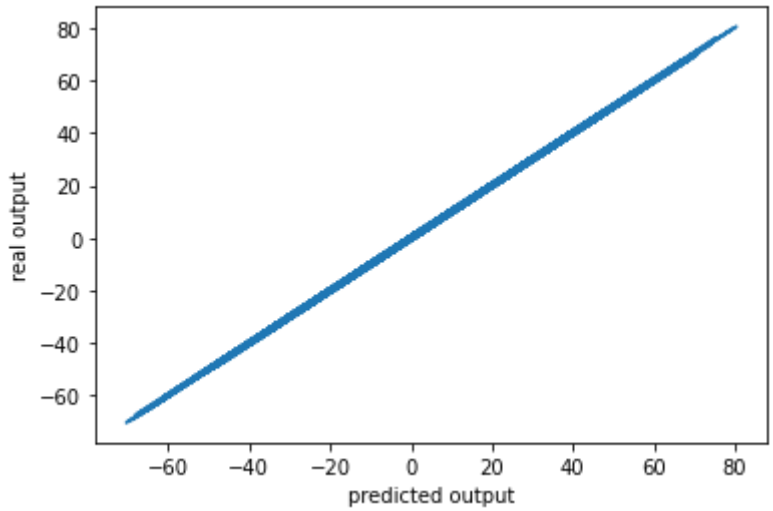
```
plt.plot(history.history['mse'])
plt.plot(history.history['val_mse'])
plt.title('mean squared error')
plt.ylabel('mse')
plt.xlabel('epochs')
```

```
plt.legend(['train' , 'validation'] , loc = 'upper right')
plt.show()
```



Predict and visualize prediction along with the "ground truth" values

```
plt.plot(np.squeeze(model.predict_on_batch(input)), np.squeeze(z))
plt.xlabel('predicted output')
plt.ylabel('real output')
plt.show()
```



Predict any value and ...

```
#print("Predicted z for x=2, y=3 ---> ", model.predict([[2,3]]).round(2))
print("Predicted z for x=2, y=3, a=4 ---> ", model.predict([[2,3,4]]).round(2))
```

```
Predicted z for x=2, y=3, a=4 --->  [[48.88]]
```

... compare with the **ground truth** value

```
#checking from equation
#z = 7*x + 6*y + 5
#print("Expected output: ", 7*2 + 6*3 + 5)

#z = 7*x + 6*y + 3*4 + 5
print("Expected output: ", 7*2 + 6*3 + 3*4 + 5)
```

```
Expected output:  49
```

DL Workflow - Example at Colab - TPU version

Let's consider the components and workflow of DL

Setup Hardware Environment:

selectby menu command **Runtime -> Change Runtime Type** above:

- None (CPU),
- GPU,
- TPU

Setup Software Environment

Import required libraries

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Set up (get) dataset

Generate data here

```
number_of_datapoints = 1000

# for reproducibility of results
#  every time you call the numpy's other random function, the result will be the s
np.random.seed(1941)

# generate random x values in the range -5 to +5
x = np.random.uniform(low = -5 , high = 5 , size = (number_of_datapoints, 1))
# generate random y values in the range -5 to +5
y = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
# generate some random error in the range -1 to +1
noise = np.random.uniform(low =-1 , high =1, size = (number_of_datapoints, 1))
#z = 7 * x + 6 * y + 5 + noise

a = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
z = 7 * x + 6 * y + 3 * a + 5 + noise
```

Print x, y and z sample values for manual verification

```
x[:5,:].round(2)
```

```
array([[-1.51],
       [ 0.06],
       [-0.87],
       [-1.67],
       [-2.13]])
```

```
y[:2,:].round(2)
```

```
array([[-3.76],
       [ 2.25]])
```

```
z[:2,:]
```

```
array([[-21.85820882],
       [ 18.21713423]])
```

```
z[:2,:].round(2)
```

```
array([[-21.86],
       [ 18.22]])
```

```
z.shape
```

```
(1000, 1)
```

Stack x and y arrays for inputting to neural network

```
#input = np.column_stack((x,y))
input = np.column_stack((x,y,a))
```

Print few values of input array for demonstration purpose.

```
input[:2,:].round(2)
```

```
array([[-1.51, -3.76,  1.89],
       [ 0.06,  2.25, -0.27]])
```

Get (define or load) model

Define by Keras tools here

Create a sequential model consisting of single layer with a single neuron.

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1)])
```

Compile (configure hyperparameters) model

Compile the model with the spcified optimizier, loss function and error metrics.

```
model.compile(optimizer = 'sgd' , loss = 'mean_squared_error' , metrics = ['mse']
```

Train and validate the model

Create callbacks (for various aims - to save training history here)

Import History module to record loss and accuracy on each epoch during training

```
from tensorflow.keras.callbacks import History
history = History()
```

Start training/validation ...

The first parameter specifies the input, the second specifies the corresponding target values. The epochs defines the number of iterations in training. The validation split specifies what percentage of input data would be used for validating the model. The callbacks parameter specifies the variable were the collected statistics would be accumulated.

```
%%time
model.fit(input, z , epochs = 15 , verbose = 1, validation_split=0.2, callbacks=[h
```

```
Epoch 1/15
25/25 [==============================] - 1s 19ms/step - loss: 134.1552 - mse:
Epoch 2/15
25/25 [==============================] - 0s 6ms/step - loss: 6.6240 - mse: 6.
Epoch 3/15
25/25 [==============================] - 0s 6ms/step - loss: 2.5988 - mse: 2.
Epoch 4/15
25/25 [==============================] - 0s 6ms/step - loss: 1.1371 - mse: 1.
Epoch 5/15
25/25 [==============================] - 0s 6ms/step - loss: 0.6147 - mse: 0.
```

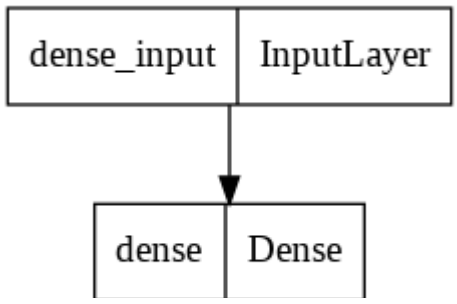
```
Epoch 6/15
25/25 [==============================] - 0s 6ms/step - loss: 0.4233 - mse: 0.
Epoch 7/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3498 - mse: 0.
Epoch 8/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3248 - mse: 0.
Epoch 9/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3157 - mse: 0.
Epoch 10/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3125 - mse: 0.
Epoch 11/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3110 - mse: 0.
Epoch 12/15
25/25 [==============================] - 0s 7ms/step - loss: 0.3111 - mse: 0.
Epoch 13/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3110 - mse: 0.
Epoch 14/15
25/25 [==============================] - 0s 5ms/step - loss: 0.3095 - mse: 0.
Epoch 15/15
25/25 [==============================] - 0s 4ms/step - loss: 0.3112 - mse: 0.
CPU times: user 2.3 s, sys: 114 ms, total: 2.41 s
Wall time: 6.05 s
<keras.callbacks.History at 0x7f55739f3990>
```

Visualize the model structure

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 1)                 4

=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

```
tf.keras.utils.plot_model(model, 'simple_model.png')
```



Check the available metrics

Print keys in the history just to know their names. These will be used for plotting the metrics.

```
print(history.history.keys())
```

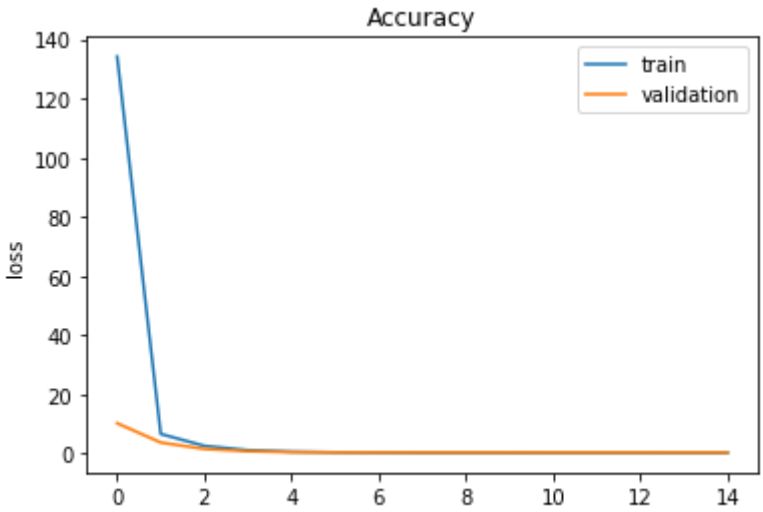
```
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

Visualize the training/validation history

Loss

Plot the loss metric on both training and validation datasets.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Accuracy')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

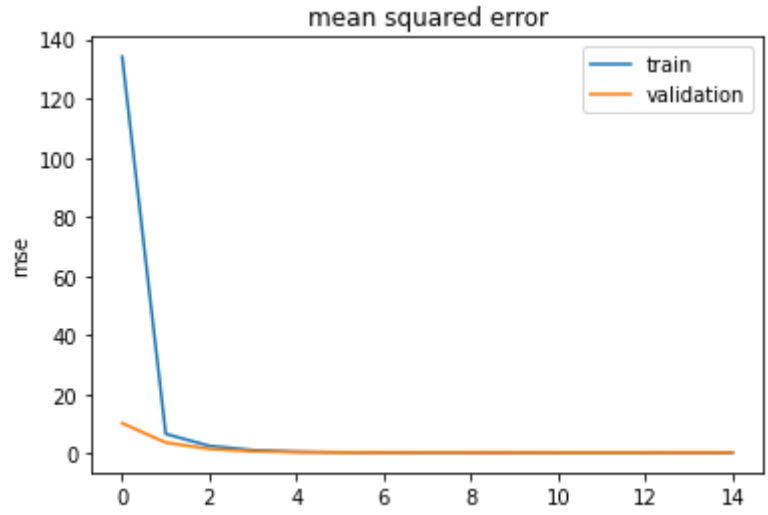


Mean squared error (MSE)

Plot the mean squared error on both training and validation datasets.

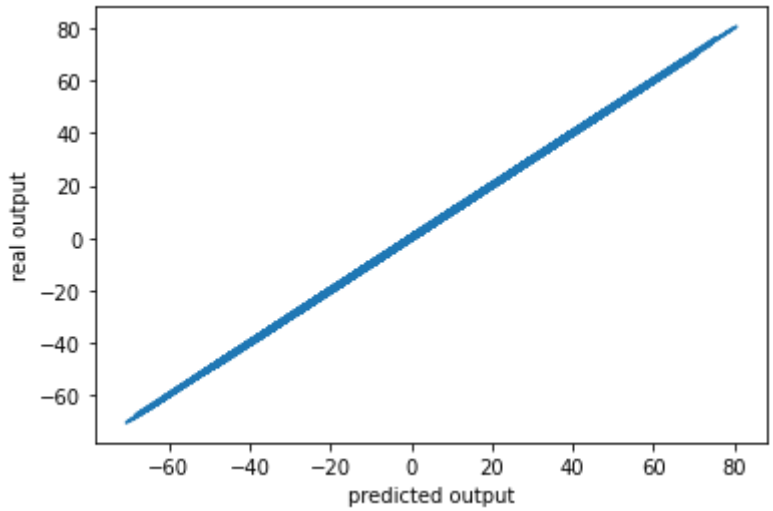
```
plt.plot(history.history['mse'])
plt.plot(history.history['val_mse'])
plt.title('mean squared error')
plt.ylabel('mse')
plt.xlabel('epochs')
```

```
plt.legend(['train' , 'validation'] , loc = 'upper right')
plt.show()
```



Predict and visualize prediction along with the "ground truth" values

```
plt.plot(np.squeeze(model.predict_on_batch(input)), np.squeeze(z))
plt.xlabel('predicted output')
plt.ylabel('real output')
plt.show()
```



Predict any value and ...

```
#print("Predicted z for x=2, y=3 ---> ", model.predict([[2,3]]).round(2))
print("Predicted z for x=2, y=3, a=4 ---> ", model.predict([[2,3,4]]).round(2))
```

```
Predicted z for x=2, y=3, a=4 --->  [[48.98]]
```

... compare with the **ground truth** value

```
#checking from equation
#z = 7*x + 6*y + 5
#print("Expected output: ", 7*2 + 6*3 + 5)

#z = 7*x + 6*y + 3*4 + 5
print("Expected output: ", 7*2 + 6*3 + 3*4 + 5)
```

```
Expected output:  49
```

DEMO 2 - Workflow Example with External Data, Tensorboard, Binary Classification

```
import tensorflow as tf
from tensorflow import keras
import pandas as pd
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Loading Data

```
# Load data from local source
DATASET_PATH = '/content/drive/MyDrive/2022_COLAB_NN/Lecture_04_DL_in_TF/NN_Lectur
data=pd.read_csv(DATASET_PATH)
```

Preprocessing Data

```
# Shuffle data for taking care of patterns in data collection
from sklearn.utils import shuffle
data=shuffle(data) #shuffling the data
```

```
# Examine loaded data
data
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Ten
617	618	15766575	Larionova	612	Germany	Female	62	
229	230	15605461	Lucas	594	Germany	Female	29	
3856	3857	15653306	Ermakova	679	Germany	Female	32	

```
# Check for null values
data.isnull().sum()
```

```
RowNumber          0
CustomerId         0
Surname            0
CreditScore        0
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

```
# Drop irrelevant columns to set up features vector
X = data.drop(labels=['CustomerId', 'Surname', 'RowNumber', 'Exited'], axis = 1)
# Set up labels vector
y = data['Exited']
```

```
# Check data types for finding categorical columns
X.dtypes
```

```
CreditScore          int64
Geography           object
Gender              object
Age                  int64
Tenure               int64
Balance            float64
NumOfProducts        int64
HasCrCard            int64
IsActiveMember       int64
EstimatedSalary    float64
dtype: object
```

```
# Examine few records for finding values in categorical columns
X.head()
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfPrc
617	612	Germany	Female	62	8	140745.33	
229	594	Germany	Female	29	3	130830.22	

```
# Encode categorical columns
from sklearn.preprocessing import LabelEncoder
label = LabelEncoder()
X['Geography'] = label.fit_transform(X['Geography'])
X['Gender'] = label.fit_transform(X['Gender'])
```

```
# Drop the Geography column to reduce the number of features
X = pd.get_dummies(X, drop_first=True, columns=['Geography'])
X.head()
```

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasC
617	612	0	62	8	140745.33	1	
229	594	0	29	3	130830.22	1	
3856	679	0	32	0	88335.05	1	
6087	561	0	27	9	135637.00	1	

```
# Scale all data points to -1 to + 1
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
# Split dataset into training and validation
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

```
# Determine number of features
X_train.shape[1]
```

```
11
```

Defining DNN

```
# Create a stacked layers sequential network
model = keras.models.Sequential() # Create linear stack of layers
model.add(keras.layers.Dense(128, activation = 'relu', input_dim = X_train.shape[1
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(keras.layers.Dense(32, activation = 'relu'))
model.add(keras.layers.Dense(1, activation = 'sigmoid')) # activation sigmoid for
```

```
# Print model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               1536

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 1)                 33

=================================================================
Total params: 11,905
Trainable params: 11,905
Non-trainable params: 0
_________________________________________________________________
```

Compiling Model

```
# Compile model with desired loss function, optimizer and evaluation metrics
model.compile(loss = 'binary_crossentropy', optimizer='adam', metrics=['accuracy']
```

```
tf.keras.utils.plot_model(model, 'simple_DNN_model.png')
```

Preparing TensorBoard

```
#to clear any other logs if present so that graphs won't overlap with previous sav
!rm -rf ./log/
```

```
#tensorboard visualization
import datetime, os
logdir = os.path.join("log", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq = 1)
```

Perform Training

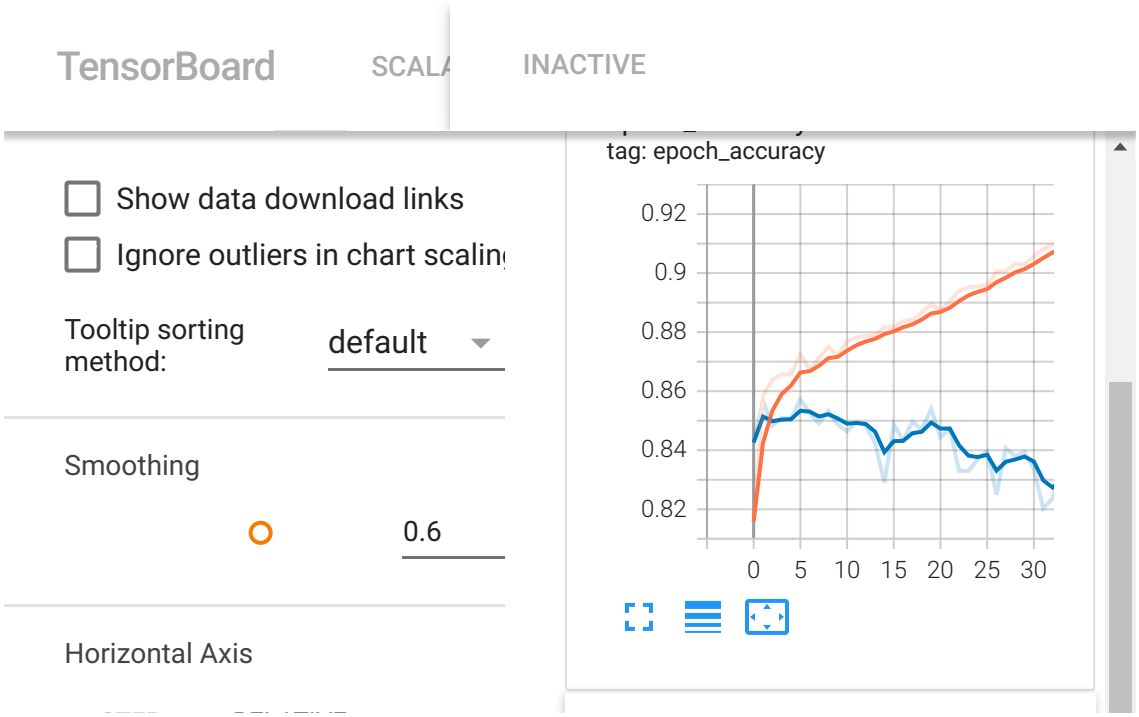
```
# Perform training
r = model.fit(X_train, y_train, batch_size = 32, epochs = 50, validation_data = (X
```

```
Epoch 20/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2654 - acc
Epoch 21/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2647 - acc
Epoch 22/50
219/219 [==============================] - 1s 5ms/step - loss: 0.2585 - acc
Epoch 23/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2560 - acc
Epoch 24/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2485 - acc
Epoch 25/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2468 - acc
Epoch 26/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2464 - acc
Epoch 27/50
219/219 [==============================] - 1s 5ms/step - loss: 0.2393 - acc
Epoch 28/50
219/219 [==============================] - 1s 5ms/step - loss: 0.2358 - acc
Epoch 29/50
219/219 [==============================] - 1s 5ms/step - loss: 0.2315 - acc
Epoch 30/50
219/219 [==============================] - 1s 5ms/step - loss: 0.2310 - acc
Epoch 31/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2248 - acc
Epoch 32/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2206 - acc
Epoch 33/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2156 - acc
Epoch 34/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2152 - acc
Epoch 35/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2079 - acc
Epoch 36/50
219/219 [==============================] - 1s 4ms/step - loss: 0.2047 - acc
Epoch 37/50
```

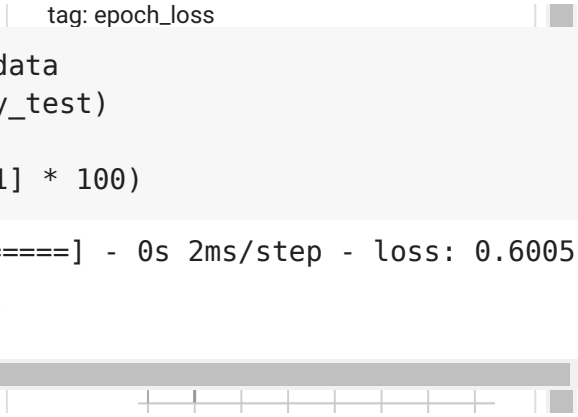
```
219/219 [==============================] - 1s 4ms/step - loss: 0.2065 - acc
Epoch 38/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1996 - acc
Epoch 39/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1972 - acc
Epoch 40/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1924 - acc
Epoch 41/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1897 - acc
Epoch 42/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1851 - acc
Epoch 43/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1810 - acc
Epoch 44/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1783 - acc
Epoch 45/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1767 - acc
Epoch 46/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1748 - acc
Epoch 47/50
219/219 [==============================] - 1s 4ms/step - loss: 0.1706 - acc
Epoch 48/50
```

TensorBoard Visualization

```
# Load tensorboard in Colab
%load_ext tensorboard
%tensorboard --logdir log #command to launch tensorboard on colab
```



Evaluting Model Performance

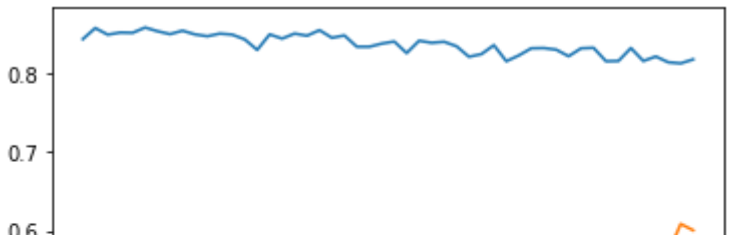



```
# evaluate model performance on test data
test_scores = model.evaluate(X_test, y_test)
print('Test Loss: ', test_scores[0])
print('Test accuracy: ', test_scores[1] * 100)
```

```
94/94 [==============================] - 0s 2ms/step - loss: 0.6005 - accurac
Test Loss:  0.6005458831787109
Test accuracy:  81.69999718666077
```

Plotting Metrics in matplotlib

```
# Plot metrics in matplotlib
%matplotlib inline
import matplotlib.pyplot as plt #for plotting curves

plt.plot(r.history['val_accuracy'], label='val_acc')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
plt.show()
```



Predicting on test data


```
import numpy as np
# Predict on test data
y_pred = model.predict(X_test)
print(y_pred)
pred_classes = np.round(y_pred)
print(pred_classes)
```

```
[[0.00120634]
 [0.00346696]
 [0.16901328]
 ...
 [0.31429288]
 [0.13142611]
 [0.01158743]]
[[0.]
 [0.]
 [0.]
 ...
 [0.]
 [0.]
 [0.]]
```

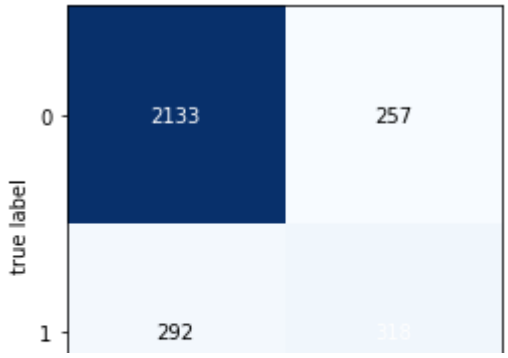
Confusion Matrix

```
# Create confusion matrix
from sklearn.metrics import confusion_matrix
cf = confusion_matrix(y_test, pred_classes)
cf
```

```
array([[2133,  257],
       [ 292,  318]])
```

```
# Plot confusion matrix
from mlxtend.plotting import plot_confusion_matrix
plot_confusion_matrix(conf_mat = cf) #, cmap = plt.cm.hsv)
```

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7efe46a33410>)
```



Accuracy Score

```
# Compute accuracy score
from sklearn.metrics import accuracy_score
accuracy_score(y_test, pred_classes)
```

```
0.817
```

Predicting on unseen data

```
# Predict on unseen customer data
customer = model.predict([[615, 1, 22, 5, 20000, 5, 1, 1, 60000, 0, 0]])
customer
```

```
array([[1.]], dtype=float32)
```

```
if customer[0] == 1:
  print ("Customer is likely to leave")
else:
  print ("Customer will stay")
```

```
Customer is likely to leave
```

```

```

✓ 0s completed at 6:58 PM

DEMO 3A. Model from TF2 Keras library

Learning type: from scratch

Model: custom CNN with 2, 4, 6 layers

Dataset: CIFAR10

Task: image classification

Import libraries

```
import tensorflow as tf
```

Look at datasets available in Keras

```
# See what datasets are available?
import tensorflow_datasets as tfds
print ("Number of datasets: ", len(tfds.list_builders()))
tfds.list_builders()
```

```
      'huggingface:quac',
      'huggingface:quail',
      'huggingface:quarel',
      'huggingface:quartz',
      'huggingface:quickdraw',
      'huggingface:quora',
      'huggingface:quoref',
      'huggingface:race',
      'huggingface:re_dial',
      'huggingface:reasoning_bg',
      'huggingface:recipe_nlg',
      'huggingface:reclor',
      'huggingface:red_caps',
      'huggingface:reddit',
      'huggingface:reddit_tifu',
      'huggingface:refresd',
      'huggingface:reuters21578',
      'huggingface:riddle_sense',
      'huggingface:ro_sent',
      'huggingface:ro_sts',
      'huggingface:ro_sts_parallel',
      'huggingface:roman_urdu',
      'huggingface:roman_urdu_hate_speech',
      'huggingface:ronec',
      'huggingface:ropes',
      'huggingface:rotten_tomatoes',
      'huggingface:russian_super_glue',
      'huggingface:rvl_cdip',
      'huggingface:s2orc'
```

```
'huggingface:samsum',
'huggingface:sanskrit_classic',
'huggingface:saudinewsnet',
'huggingface:sberquad',
'huggingface:sbu_captions',
'huggingface:scan',
'huggingface:scb_mt_enth_2020',
'huggingface:scene_parse_150',
'huggingface:schema_guided_dstc8',
'huggingface:scicite',
'huggingface:scielo',
'huggingface:scientific_papers',
'huggingface:scifact',
'huggingface:sciq',
'huggingface:scitail',
'huggingface:scitldr',
'huggingface:search_qa',
'huggingface:sede',
'huggingface:selqa',
'huggingface:sem_eval_2010_task_8',
'huggingface:sem_eval_2014_task_1',
'huggingface:sem_eval_2018_task_1',
'huggingface:sem_eval_2020_task_11',
'huggingface:sent_comp',
'huggingface:senti_lex',
'huggingface:senti_ws',
'huggingface:sentiment140',
'huggingface:sepedi_ner',
'huggingface:sesotho_ner_corpus',
```

Load CIFAR10 dataset

```
# Load the training/testing datasets
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 14s 0us/step
170508288/170498071 [==============================] - 14s 0us/step
```

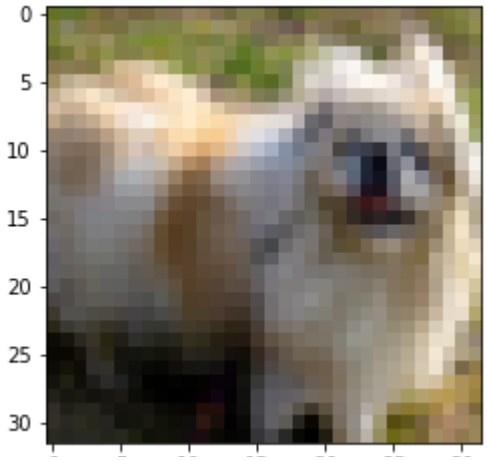
```
# Check how many data points are loaded?
print("x_train dimensions : ",x_train.shape)
print("x_test dimensions  : ",x_test.shape)
print("y_train dimensions : ",y_train.shape)
print("y_test dimensions  : ",y_test.shape)
```

```
x_train dimensions :  (50000, 32, 32, 3)
x_test dimensions  :  (10000, 32, 32, 3)
y_train dimensions :  (50000, 1)
y_test dimensions  :  (10000, 1)
```

```
# print one sample data point
import matplotlib.pyplot as plt
```

```
plt.imshow(x_train[40])
#plt.imshow(x_test[10])
```

```
<matplotlib.image.AxesImage at 0x7f4127c11c50>
```



Split dataset to train/validation parts

```
# Splitting training data into train and validation sets
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size = 0.
```

Apply data augmentation

```
# Image augumentatiion with Keras image generator class
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
  rotation_range = 15,
  width_shift_range = 0.1,
  height_shift_range = 0.1,
  horizontal_flip = True, )
```

Apply data normalization

```
# Function for scaling data
def normalize(data):
    data = data.astype("float32")
    data = data/255.0
    return data
```

```
# Scale and augment train/test datasets
x_train = normalize(x_train)
datagen.fit(x_train)
```

```
x_val = normalize(x_val)
datagen.fit(x_val)
x_test = normalize(x_test)
```

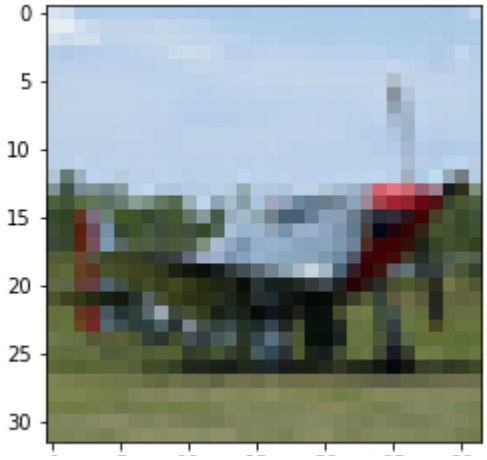
Pre-process labels

```
# Treat categorical columns
y_train = tf.keras.utils.to_categorical(y_train , 10)
y_test  = tf.keras.utils.to_categorical(y_test , 10)
y_val = tf.keras.utils.to_categorical(y_val , 10)
```

Look at some input data

```
# Display the modified image for fun
plt.imshow(x_train[40])
```

```
<matplotlib.image.AxesImage at 0x7f4126db7e10>
```




```
# print the dimensions after the data preprocessing
print("x_train dimensions : ",x_train.shape)
print("y_train dimensions : ",y_train.shape)
print("x_test dimensions  : ",x_test.shape)
print("y_test dimensions  : ",y_test.shape)
print("x_val dimensions   : ",x_val.shape)
print("y_val dimensions   : ",y_val.shape)
```

```
x_train dimensions :  (47500, 32, 32, 3)
y_train dimensions :  (47500, 10)
x_test dimensions  :  (10000, 32, 32, 3)
y_test dimensions  :  (10000, 10)
x_val dimensions   :  (2500, 32, 32, 3)
y_val dimensions   :  (2500, 10)
```

Configure training, evlauation, and estimation stages

```
# Train, evaluate and print metrics
def results(model):
  epoch = 20
  r = model.fit(datagen.flow(x_train, y_train, batch_size = 32), epochs = epoch, s
  acc = model.evaluate(x_test , y_test)
  print("test set loss : " , acc[0])
  print("test set accuracy :", acc[1]*100)

  # Plot training and validation accuracy
  epoch_range = range(1, epoch+1)
  plt.plot(epoch_range, r.history['accuracy'])
  plt.plot(epoch_range, r.history['val_accuracy'])
  plt.title('Classification Accuracy')
  plt.ylabel('Accuracy')
  plt.xlabel('Epoch')
  plt.legend(['Train', 'Val'], loc='lower right')
  plt.show()

  # Plot training & validation loss values
  plt.plot(epoch_range,r.history['loss'])
  plt.plot(epoch_range, r.history['val_loss'])
  plt.title('Model loss')
  plt.ylabel('Loss')
  plt.xlabel('Epoch')
  plt.legend(['Train', 'Val'], loc='lower right')
  plt.show()
```

```
# Predict the class in a given image
from tensorflow.keras.preprocessing.image import load_img , img_to_array

# setup class names array
classes = ['airplane' ,'automobile', 'bird' , 'cat' , 'deer' ,'dog' ,'frog', 'hors
def predict_class(filename , model):

  # load and display image
  img = load_img(filename, target_size=(32, 32))
  plt.imshow(img)

  # convert to array
  # reshape into a single sample with 3 channels
  img = img_to_array(img)
  img = img.reshape(1,32,32,3)

  # prepare pixel data
  img = img.astype('float32')
  img = img/255.0

  # predicting the results
  result = model.predict(img)

  # copy predictions to dictionary
  dict2 = {}
```

```
  for i in range(10):
      dict2[result[0][i]] = classes[i]

  # sort on predictions
  res = result[0]
  res.sort()

  # pick up top 3 predictions
  res = res[::-1]
  results = res[:3]

  print("Top predictions of these images are")
  for i in range(3):
      print("{} : {}".format(dict2[results[i]] , (results[i]*100).round(2)))

  print('The image given as input is')
```

First Model : Simple model with 2 convolution layers

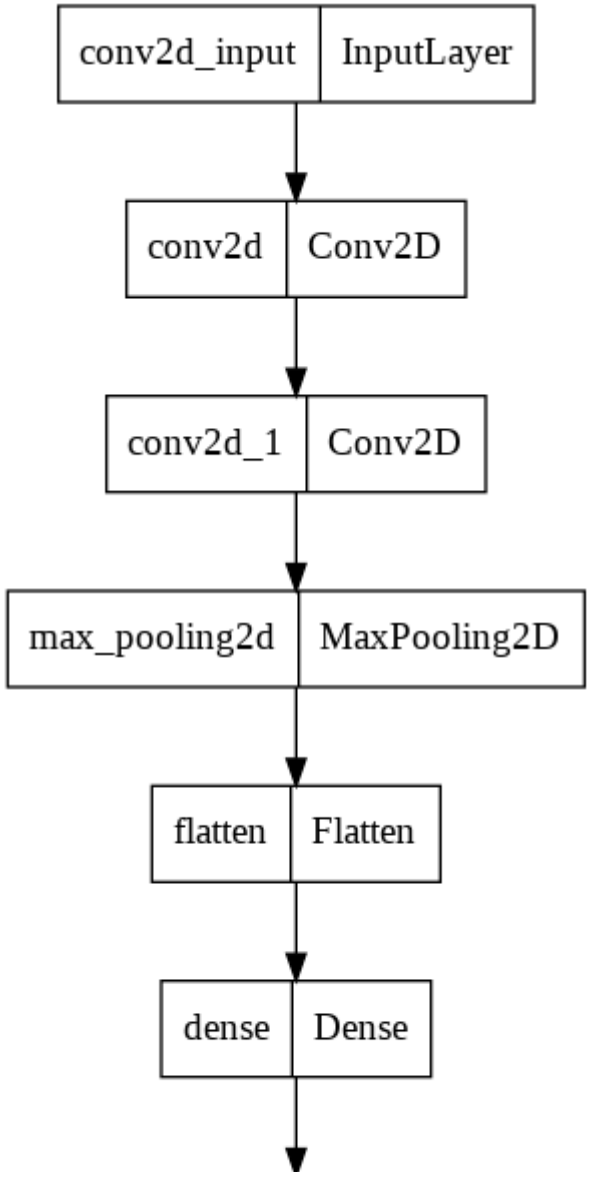
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten,

# define model
model_1 = Sequential([
      Conv2D(32, (3, 3), activation = 'relu', padding = 'same',
             input_shape = (32, 32, 3)),
      Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
      MaxPooling2D((2, 2)),
      Flatten(),
      Dense(128, activation = 'relu'),
      Dense(10, activation = 'softmax')
])
```

```
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 128)               1048704

 dense_1 (Dense)             (None, 10)                1290
```

```
=================================================================
Total params: 1,060,138
Trainable params: 1,060,138
Non-trainable params: 0
_________________________________________________________________
```

```
# network plot
from tensorflow.keras.utils import plot_model
plot_model(model_1, to_file = 'model1.png')
```




```
# compile
opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_1.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics = ['ac
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/gradient_descent.py
  super(SGD, self).__init__(name, **kwargs)
```

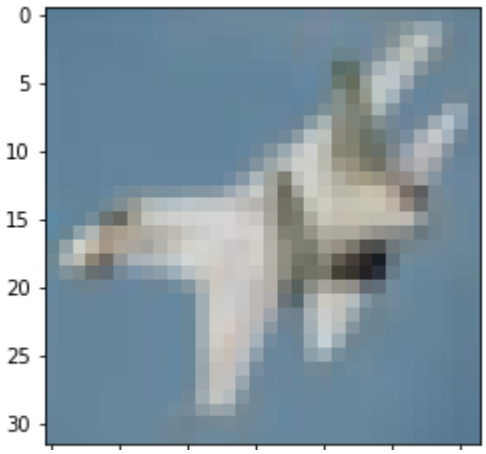
```
# train, evaluate, display metrics
```

```
results(model_1)
```

```
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 12/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 13/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 14/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 15/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 16/20
1484/1484 [==============================] - 23s 16ms/step - loss
Epoch 17/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 18/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 19/20
1484/1484 [==============================] - 23s 16ms/step - loss
Epoch 20/20
1484/1484 [==============================] - 23s 16ms/step - loss
313/313 [==============================] - 1s 3ms/step - loss: 0.9
test set loss :  0.9246014356613159
```

```
# predict on unseen image
import urllib
resource = urllib.request.urlopen("https://raw.githubusercontent.com/Apress/artifi
output = open("file01.jpg", "wb")
output.write(resource.read())
output.close()
predict_class("file01.jpg", model_1)
```

```
Top predictions of these images are
airplane : 88.28
deer : 6.35
bird : 1.84
The image given as input is
```



Second Model : with 4 Convolution layers

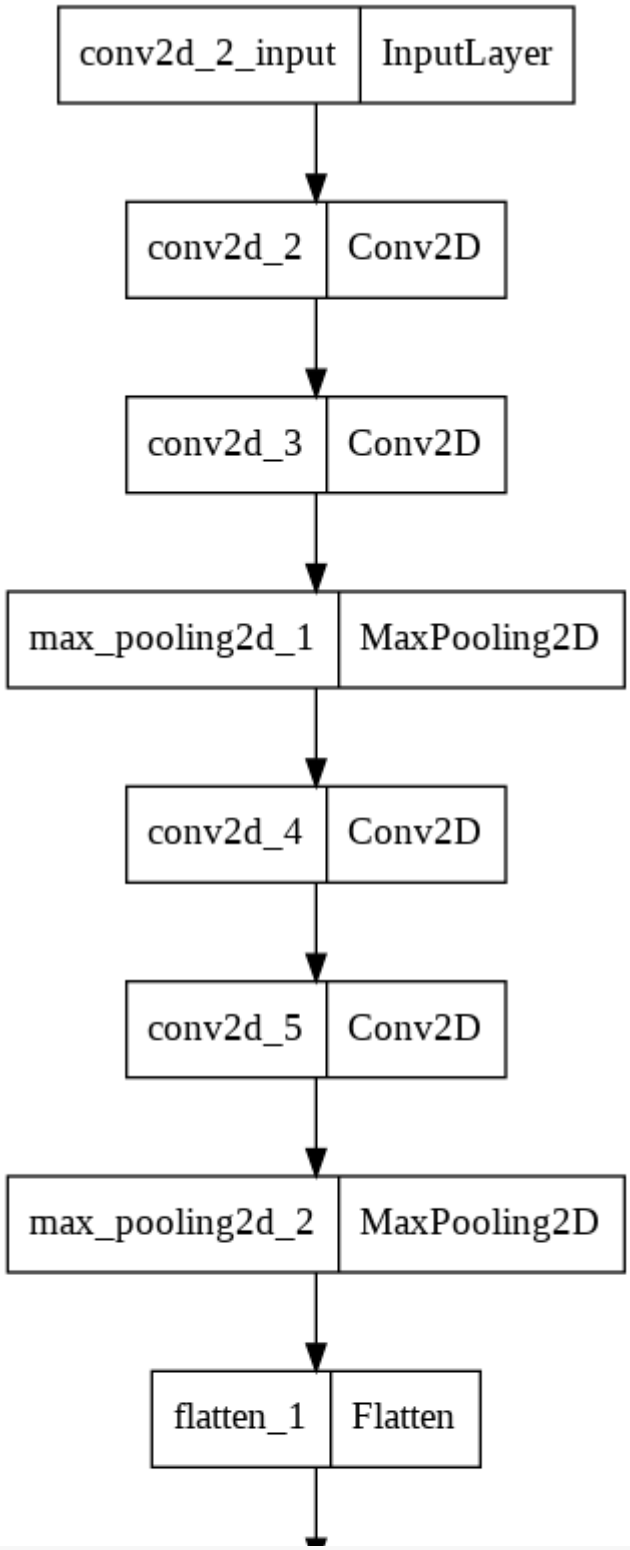
```
# define model
model_2 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same', input_shape = (32, 32,
```

```
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_2.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics = ['ac
```

```
model_2.summary()
```

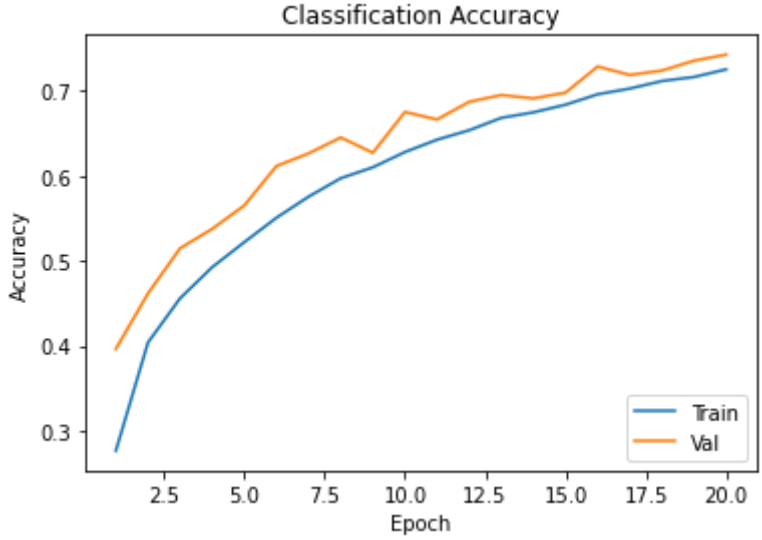
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #   
=================================================================
 conv2d_2 (Conv2D)            (None, 32, 32, 32)        896       

 conv2d_3 (Conv2D)            (None, 32, 32, 32)        9248      

 max_pooling2d_1 (MaxPooling  (None, 16, 16, 32)       0         
 2D)                                                             

 conv2d_4 (Conv2D)            (None, 16, 16, 64)        18496     

 conv2d_5 (Conv2D)            (None, 16, 16, 64)        36928     

 max_pooling2d_2 (MaxPooling  (None, 8, 8, 64)         0         
 2D)                                                             

 flatten_1 (Flatten)          (None, 4096)              0         

 dense_2 (Dense)              (None, 128)               524416    

 dense_3 (Dense)              (None, 10)                1290      

=================================================================
Total params: 591,274
Trainable params: 591,274
Non-trainable params: 0
_________________________________________________________________
```

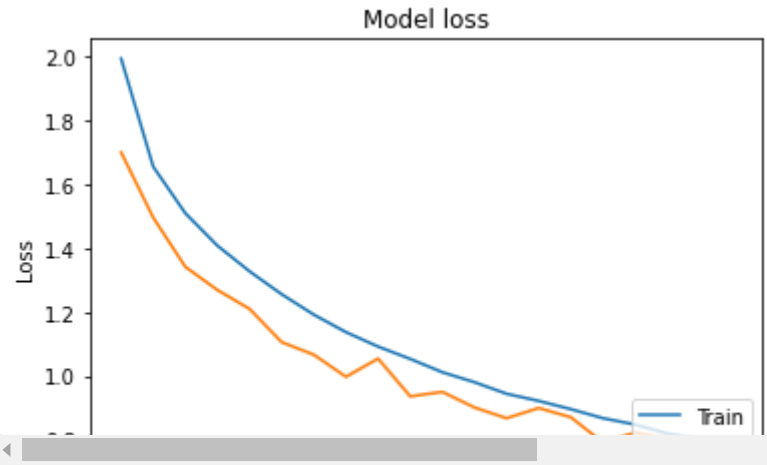
```
plot_model(model_2 , to_file='model2.png')
```




```
# train, evaluate, display metrics
results(model_2)
```

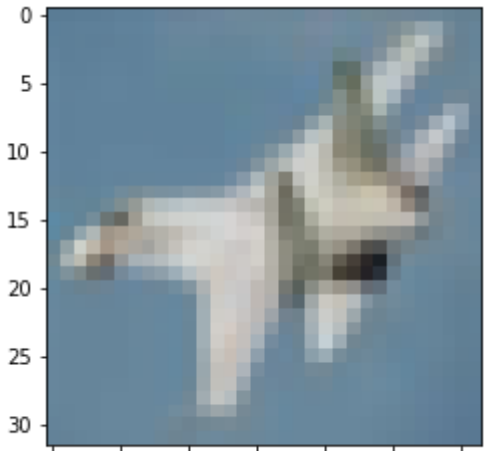
```
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 12/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 13/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 14/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 15/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 16/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 17/20
1484/1484 [==============================] - 24s 16ms/step - loss
Epoch 18/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 19/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 20/20
1484/1484 [==============================] - 25s 17ms/step - loss
313/313 [==============================] - 1s 4ms/step - loss: 0.
test set loss :  0.7524604201316833
test set accuracy : 74.37000274658203
```





```
# predict on earlier loaded image
predict_class("file01.jpg" ,model_2)
```

```
Top predictions of these images are
airplane : 80.69
deer : 15.46
cat : 1.17
The image given as input is
```



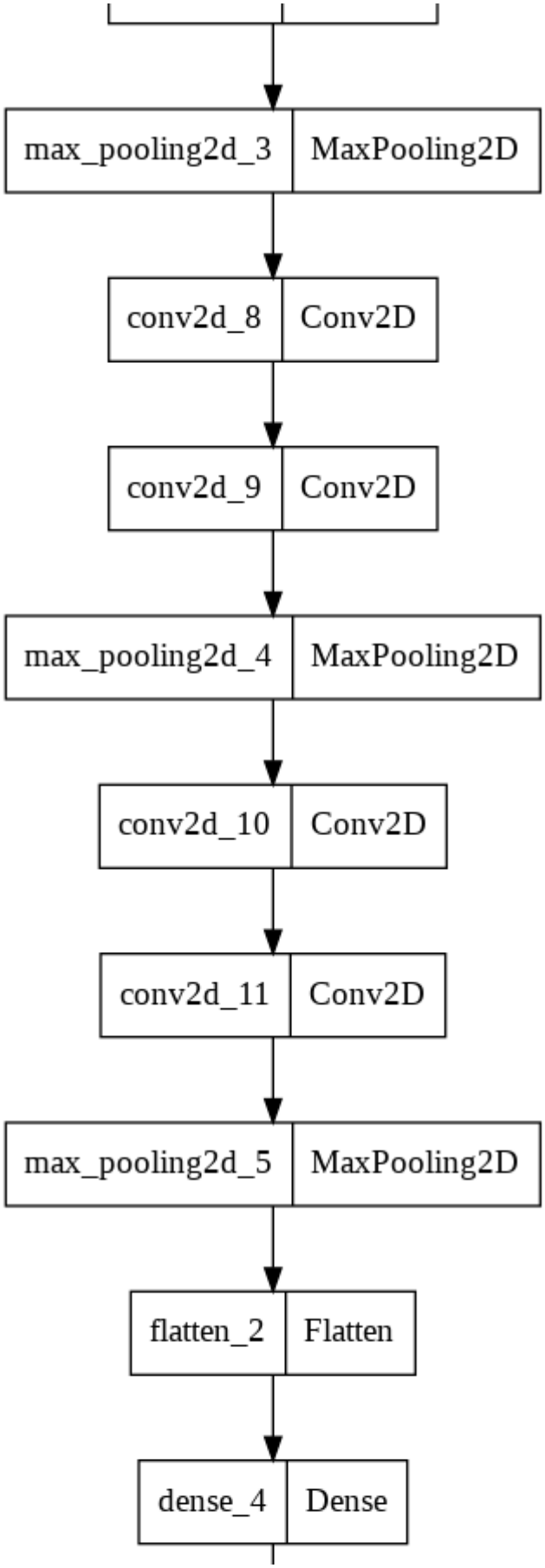
Third Model : 6 convolution layers with 32 , 64 and 128 filters respectively

```
# define model
model_3 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same', input_shape = (32, 32,
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Conv2D(128, (3, 3), activation = 'relu', padding = 'same'),
  Conv2D(128, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_3.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics = ['ac
```

```
model_3.summary()
```

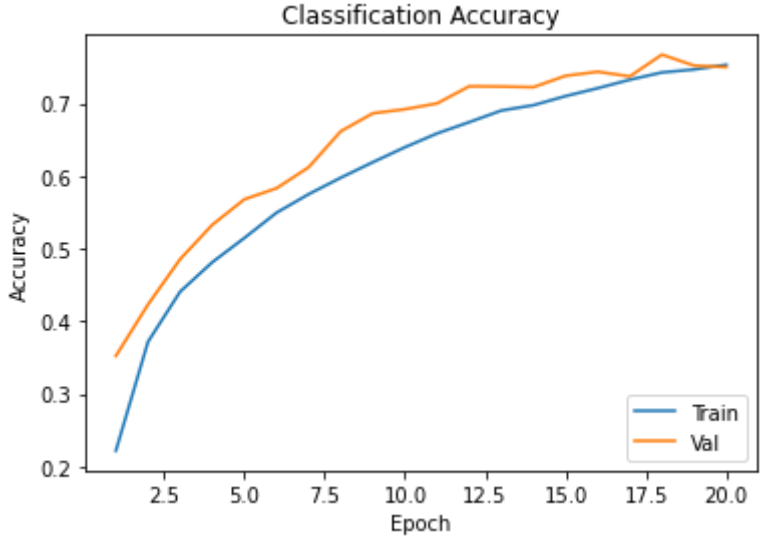
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)            (None, 32, 32, 32)        896

 conv2d_7 (Conv2D)            (None, 32, 32, 32)        9248

 max_pooling2d_3 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 conv2d_8 (Conv2D)            (None, 16, 16, 64)        18496

 conv2d_9 (Conv2D)            (None, 16, 16, 64)        36928

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 conv2d_10 (Conv2D)           (None, 8, 8, 128)         73856

 conv2d_11 (Conv2D)           (None, 8, 8, 128)         147584

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 flatten_2 (Flatten)          (None, 2048)              0

 dense_4 (Dense)              (None, 128)               262272

 dense_5 (Dense)              (None, 10)                1290

=================================================================
Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0
_________________________________________________________________
```

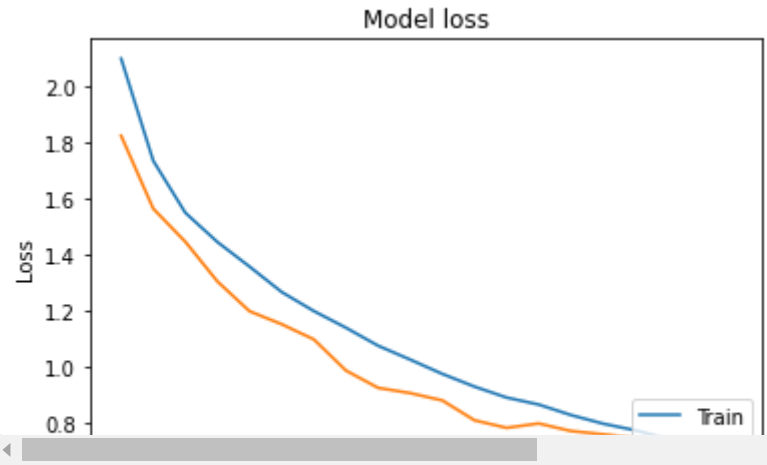
```
plot_model(model_3 , to_file='model3.png')
```




```
results(model_3)
```

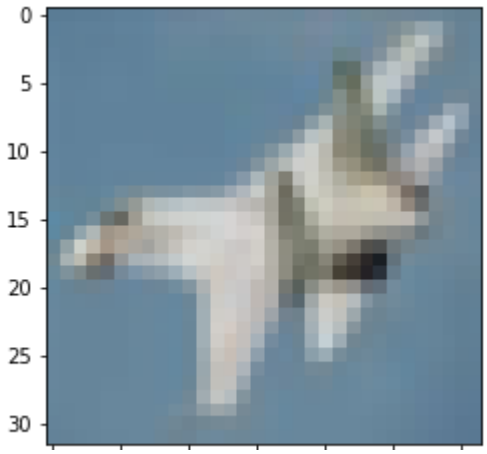
```
1484/1484 [==============================] - 27s 18ms/step - loss
Epoch 12/20
1484/1484 [==============================] - 26s 18ms/step - loss
Epoch 13/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 14/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 15/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 16/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 17/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 18/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 19/20
1484/1484 [==============================] - 26s 18ms/step - loss
Epoch 20/20
1484/1484 [==============================] - 25s 17ms/step - loss
313/313 [==============================] - 2s 5ms/step - loss: 0.
test set loss :  0.7536427974700928
test set accuracy : 74.5199978351593
```





```
predict_class("file01.jpg" ,model_3)
```

```
Top predictions of these images are
airplane : 58.95
deer : 36.06
bird : 2.24
The image given as input is
```



Model 4 : Adding Dropouts

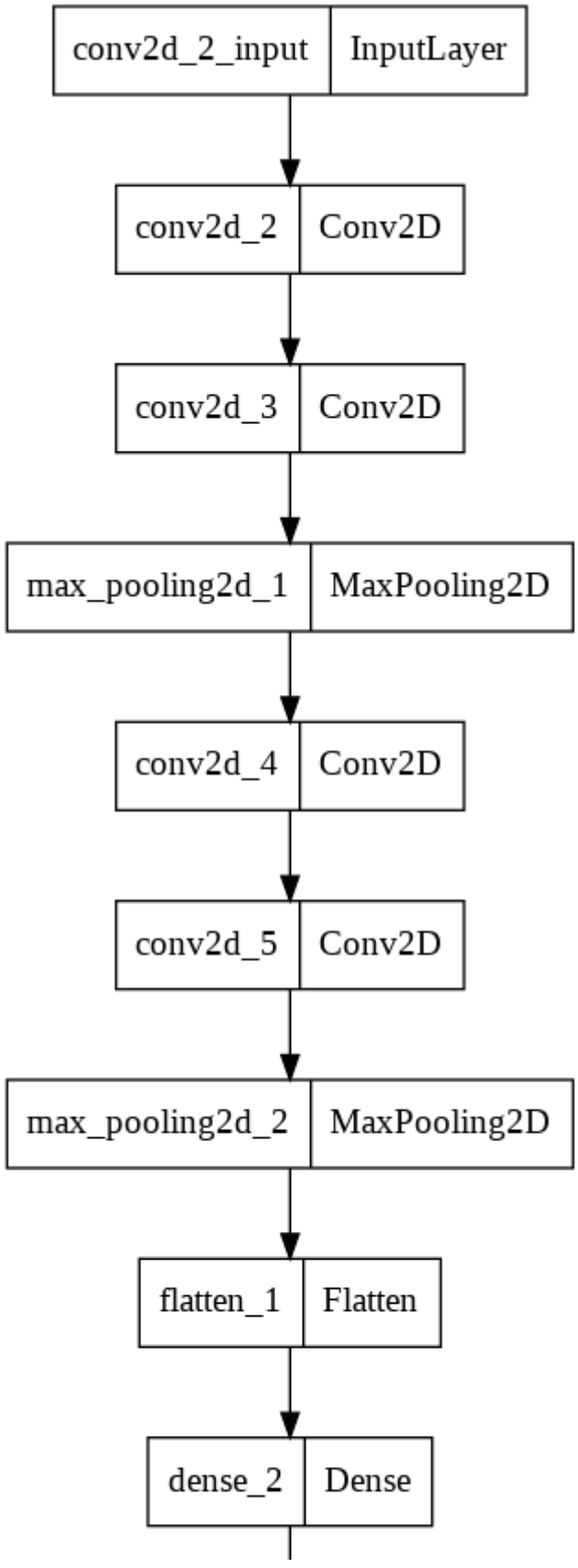
```
model_4 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', kernel_initializer = 'he_uniform', paddi
  Conv2D(32, (3, 3), activation = 'relu', kernel_initializer = 'he_uniform', paddi
  MaxPooling2D((2, 2)),
  Dropout(0.2),
  Conv2D(64, (3, 3), activation = 'relu', kernel_initializer = 'he_uniform', paddi
  Conv2D(64, (3, 3), activation = 'relu', kernel_initializer = 'he_uniform', paddi
  MaxPooling2D((2, 2)),
  Dropout(0.2),
  Conv2D(128, (3, 3), activation = 'relu', kernel_initializer = 'he_uniform', padd
  Conv2D(128, (3, 3), activation = 'relu', kernel_initializer = 'he_uniform', padd
  MaxPooling2D((2, 2)),
  Dropout(0.3),
```

```
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_4.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics = ['ac
```

```
model_4.summary()
```

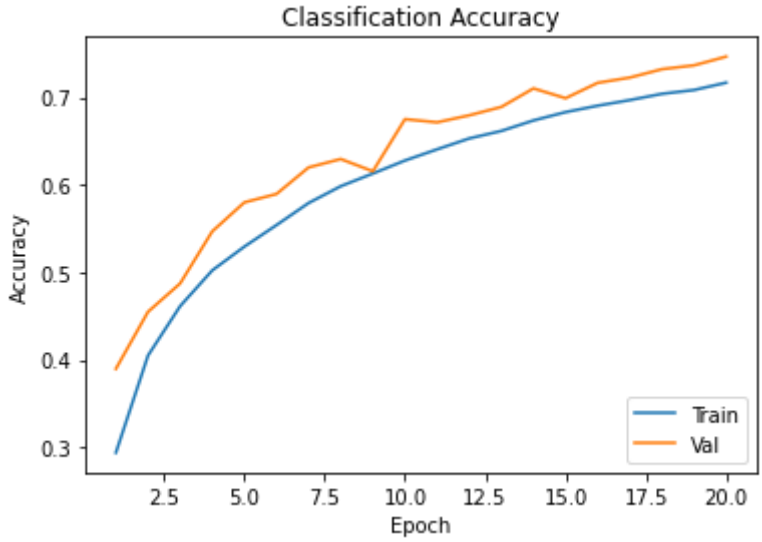
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #   
=================================================================
 conv2d_12 (Conv2D)           (None, 32, 32, 32)        896       

 conv2d_13 (Conv2D)           (None, 32, 32, 32)        9248      

 max_pooling2d_6 (MaxPooling  (None, 16, 16, 32)       0         
 2D)                                                             

 dropout (Dropout)            (None, 16, 16, 32)        0         

 conv2d_14 (Conv2D)           (None, 16, 16, 64)        18496     

 conv2d_15 (Conv2D)           (None, 16, 16, 64)        36928     

 max_pooling2d_7 (MaxPooling  (None, 8, 8, 64)         0         
 2D)                                                             

 dropout_1 (Dropout)          (None, 8, 8, 64)          0         

 conv2d_16 (Conv2D)           (None, 8, 8, 128)         73856     

 conv2d_17 (Conv2D)           (None, 8, 8, 128)         147584    

 max_pooling2d_8 (MaxPooling  (None, 4, 4, 128)        0         
 2D)                                                             

 dropout_2 (Dropout)          (None, 4, 4, 128)         0         

 flatten_3 (Flatten)          (None, 2048)              0         

 dense_6 (Dense)              (None, 128)               262272    

 dense_7 (Dense)              (None, 10)                1290      

=================================================================
Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(model_2 , to_file='model4.png')
```



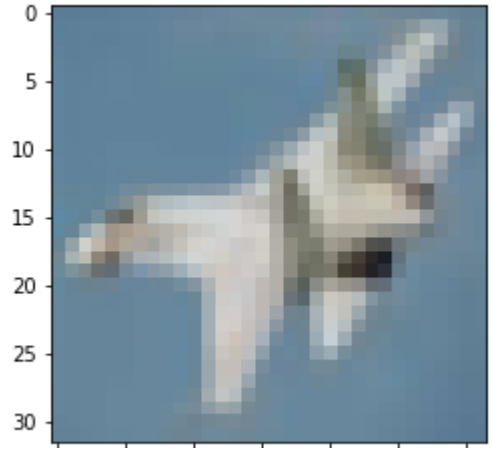

```
results(model_4)
```

```
1484/1484 [==============================] - 26s 17ms/step - loss
Epoch 12/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 13/20
1484/1484 [==============================] - 26s 17ms/step - loss
Epoch 14/20
1484/1484 [==============================] - 26s 17ms/step - loss
Epoch 15/20
1484/1484 [==============================] - 25s 17ms/step - loss
Epoch 16/20
1484/1484 [==============================] - 27s 18ms/step - loss
Epoch 17/20
1484/1484 [==============================] - 26s 17ms/step - loss
Epoch 18/20
1484/1484 [==============================] - 26s 17ms/step - loss
Epoch 19/20
1484/1484 [==============================] - 26s 17ms/step - loss
Epoch 20/20
1484/1484 [==============================] - 26s 17ms/step - loss
313/313 [==============================] - 1s 4ms/step - loss: 0.
test set loss :  0.7141719460487366
test set accuracy : 75.0
```



```
predict_class("file01.jpg" ,model_4)
```

```
Top predictions of these images are
airplane : 85.03
dog : 4.77
ship : 3.27
The image given as input is
```



Model 5 : Adding Batch Normalization and Regularization

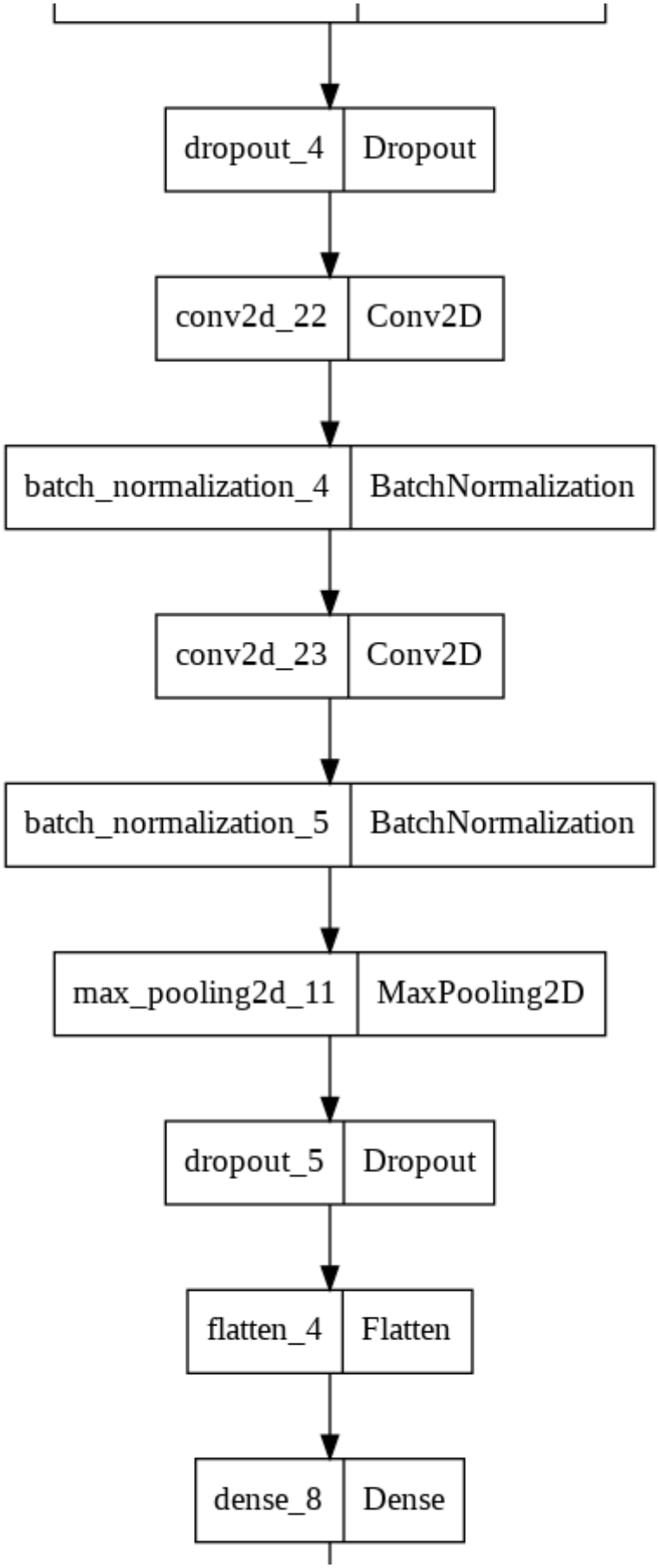
```
weight_decay = 1e-4
model_5 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same', kernel_regularizer = t
  BatchNormalization(),
  Conv2D(32, (3, 3), activation = 'relu', kernel_regularizer = tf.keras.regularize
  BatchNormalization(),
  MaxPooling2D((2, 2)),
  Dropout(0.2),
  Conv2D(64, (3, 3), activation = 'relu', kernel_regularizer = tf.keras.regularize
  BatchNormalization(),
  Conv2D(64, (3, 3), activation = 'relu', kernel_regularizer = tf.keras.regularize
  BatchNormalization(),
  MaxPooling2D((2, 2)),
  Dropout(0.3),
```

```
  Conv2D(128, (3, 3), activation = 'relu', kernel_regularizer = tf.keras.regulariz
  BatchNormalization(),
  Conv2D(128, (3, 3), activation = 'relu', kernel_regularizer = tf.keras.regulariz
  BatchNormalization(),
  MaxPooling2D((2, 2)),
  Dropout(0.3),
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_5.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics = ['ac

model_5.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_18 (Conv2D)           (None, 32, 32, 32)        896

 batch_normalization (BatchN  (None, 32, 32, 32)       128
 ormalization)

 conv2d_19 (Conv2D)           (None, 32, 32, 32)        9248

 batch_normalization_1 (Batc  (None, 32, 32, 32)       128
 hNormalization)

 max_pooling2d_9 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 dropout_3 (Dropout)          (None, 16, 16, 32)        0

 conv2d_20 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization_2 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 conv2d_21 (Conv2D)           (None, 16, 16, 64)        36928

 batch_normalization_3 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 max_pooling2d_10 (MaxPoolin  (None, 8, 8, 64)         0
 g2D)

 dropout_4 (Dropout)          (None, 8, 8, 64)          0

 conv2d_22 (Conv2D)           (None, 8, 8, 128)         73856

 batch_normalization_4 (Batc  (None, 8, 8, 128)        512
 hNormalization)

 conv2d_23 (Conv2D)           (None, 8, 8, 128)         147584

 batch_normalization_5 (Batc  (None, 8, 8, 128)        512
```

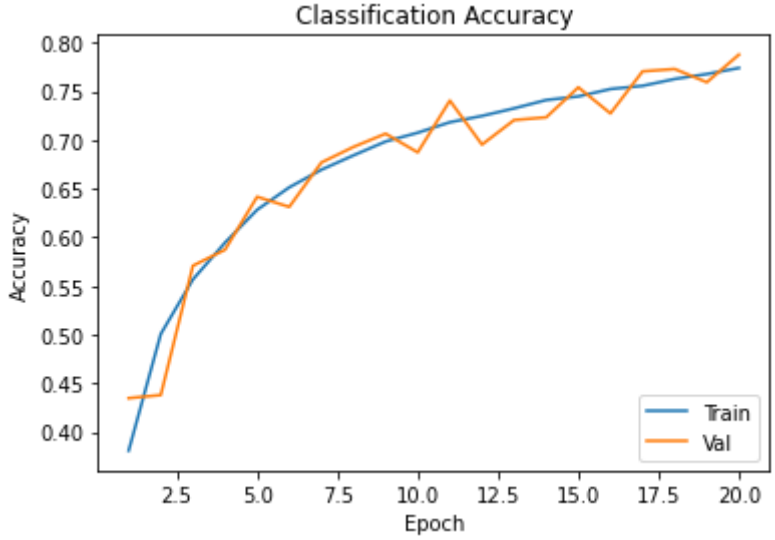
```
hNormalization)

max_pooling2d_11 (MaxPoolin  (None, 4, 4, 128)        0
g2D)

dropout_5 (Dropout)          (None, 4, 4, 128)        0

flatten_4 (Flatten)          (None, 2048)             0

dense_8 (Dense)              (None, 128)              262272

dense_9 (Dense)              (None, 10)               1290

=================================================================
Total params: 552,362
```

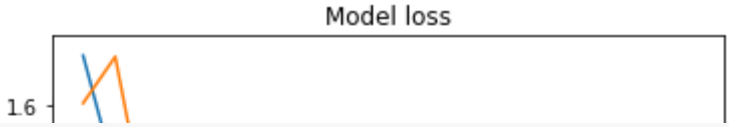
```
plot_model(model_5 , to_file = 'model5.png')
```


dropout_4
Dropout
conv2d_22
Conv2D
batch_normalization_4
BatchNormalization
conv2d_23
Conv2D
batch_normalization_5
BatchNormalization
max_pooling2d_11
MaxPooling2D
dropout_5
Dropout
flatten_4
Flatten
dense_8
Dense

```
results(model_5)
```

```
1484/1484 [==============================] - 27s 18ms/step - loss
Epoch 12/20
1484/1484 [==============================] - 34s 23ms/step - loss
Epoch 13/20
1484/1484 [==============================] - 30s 20ms/step - loss
Epoch 14/20
1484/1484 [==============================] - 32s 22ms/step - loss
Epoch 15/20
1484/1484 [==============================] - 28s 19ms/step - loss
Epoch 16/20
1484/1484 [==============================] - 27s 18ms/step - loss
Epoch 17/20
1484/1484 [==============================] - 28s 19ms/step - loss
Epoch 18/20
1484/1484 [==============================] - 27s 18ms/step - loss
Epoch 19/20
1484/1484 [==============================] - 27s 18ms/step - loss
Epoch 20/20
1484/1484 [==============================] - 27s 18ms/step - loss
313/313 [==============================] - 1s 4ms/step - loss: 0.6
test set loss :  0.6758444905281067
test set accuracy : 78.21999788284302
```






```
predict_class("file01.jpg" ,model_5)
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function Model.mak
Top predictions of these images are
airplane : 95.81
deer : 2.39
dog : 0.47
The image given as input is
```

Inferencing on "External" Data

Model Saving


```
model_1.save("model_1.h5")
model_2.save("model_2.h5")
model_3.save("model_3.h5")
model_4.save("model_4.h5")
model_5.save("model_5.h5")
```

Model Loading

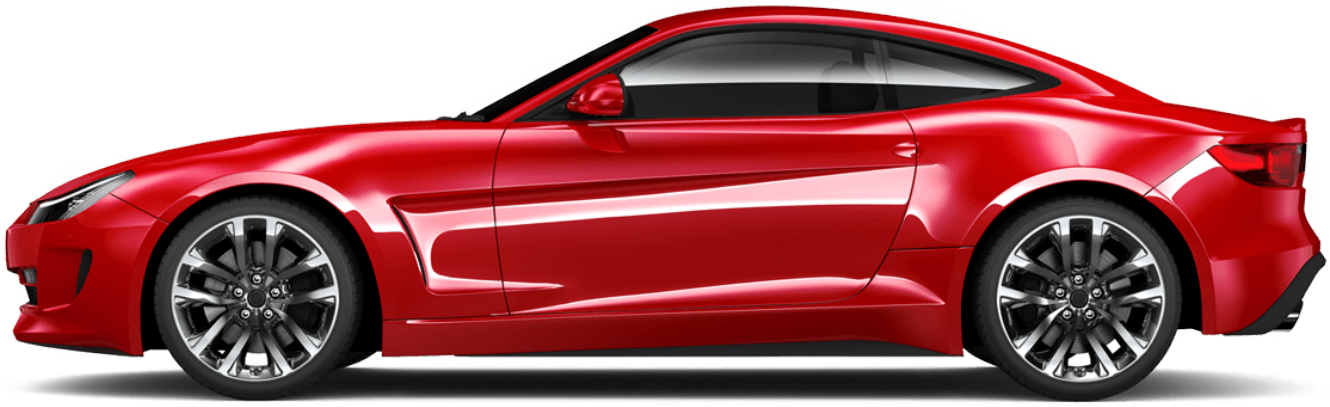
```
from tensorflow.keras.models import load_model

m1 = load_model("model_1.h5")
m2 = load_model("model_2.h5")
m3 = load_model("model_3.h5")
m4 = load_model("model_4.h5")
m5 = load_model("model_5.h5")
```

Example: airplane - SUCCESS

```
# unseen image 1
#resource = urllib.request.urlopen("https://en.wikipedia.org/wiki/File:MiG-21_Lanc
resource = urllib.request.urlopen("https://freepngimg.com/thumb/airplane/29553-8-a
output = open("aircraft.jpg","wb")
output.write(resource.read())
output.close()

predict_class("aircraft.jpg" , m1)
predict_class("aircraft.jpg" , m2)
predict_class("aircraft.jpg" , m3)
predict_class("aircraft.jpg" , m4)
predict_class("aircraft.jpg" , m5)
```

Top predictions of these images are

Example: car - SUCCESS



airplane : 0.56

```
# unseen image 2
resource = urllib.request.urlopen("https://avtokluch.com.ua/wp-content/uploads/rev
output = open("car.jpg","wb")
output.write(resource.read())
output.close()

predict_class("car.jpg" , m1)
predict_class("car.jpg" , m2)
predict_class("car.jpg" , m3)
predict_class("car.jpg" , m4)
predict_class("car.jpg" , m5)
```

```
/usr/local/lib/python3.7/dist-packages/PIL/Image.py:960: UserWarnin
  "Palette images with Transparency expressed in bytes should be "
Top predictions of these images are
truck : 89.05
frog : 6.88
automobile : 4.05
The image given as input is
Top predictions of these images are
truck : 81.43
automobile : 18.04
frog : 0.47
The image given as input is
Top predictions of these images are
automobile : 83.97
truck : 16.03
ship : 0.0
```

Example: bird - SUCCESS

```
# unseen image 3
resource = urllib.request.urlopen("https://upload.wikimedia.org/wikipedia/commons/
output = open("bird.jpg","wb")
output.write(resource.read())
output.close()

predict_class("bird.jpg" , m1)
predict_class("bird.jpg" , m2)
predict_class("bird.jpg" , m3)
predict_class("bird.jpg" , m4)
predict_class("bird.jpg" , m5)
```

DEMO 3B. Model from TF2 Hub

Learning type: transfer learning

Model: MobileNetV2

Dataset: ImageNet

Task: image classification

Import libraries

```
import tensorflow as tf
```

```
# other imports
import tensorflow_hub as hub
import numpy as np
import matplotlib.pyplot as plt
```

Get DNN Model from Tensorflow Hub

```
# Classsifier URL from TensorFlow Hub
classifier_url = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification
IMAGE_SHAPE = (224, 224)
```

```
# create a model
classifier = tf.keras.Sequential([
    hub.KerasLayer(classifier_url, input_shape = IMAGE_SHAPE+(3,))
])
```

Collect example images

```
# set up URLs for image downloads
image_url_1 = "https://upload.wikimedia.org/wikipedia/commons/9/91/Cessna.206h.sta
image_url_2 = "https://static.wikia.nocookie.net/elias-the-little-rescue-boat/imag
image_url_3 = "https://images.theconversation.com/files/250946/original/file-20181
image_url_4 = "https://www.birdspot.co.uk/wp-content/uploads/2019/12/cuckoo.jpg"
```

```
# install utility for download

!pip install wget
```

```
!pip install wget
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Requirement already satisfied: wget in /usr/local/lib/python3.7/dist-packages
```

```
# download images
import wget
wget.download(image_url_1,'image1.jpg')
wget.download(image_url_2,'image2.jpg')
wget.download(image_url_3,'image3.jpg')
wget.download(image_url_4,'image4.jpg')
```

```
'image5 (3).jpg'
```

Load and pre-process example images

```
# load images and reshape to 224x224 required by the model
import PIL.Image as Image

image1 = tf.keras.utils.get_file("/content/image1.jpg", image_url_1)
image1 = Image.open(image1).resize(IMAGE_SHAPE)
# scale the array
image1 = np.array(image1)/255.0

image2 = tf.keras.utils.get_file("/content/image2.jpg", image_url_2)
image2 = Image.open(image2).resize(IMAGE_SHAPE)
image2 = np.array(image2)/255.0

image3 = tf.keras.utils.get_file("/content/image3.jpg", image_url_3)
image3 = Image.open(image3).resize(IMAGE_SHAPE)
image3 = np.array(image3)/255.0

image4 = tf.keras.utils.get_file("/content/image4.jpg", image_url_3)
image4 = Image.open(image4).resize(IMAGE_SHAPE)
image4 = np.array(image4)/255.0

image5 = tf.keras.utils.get_file("/content/image5.jpg", image_url_4)
image5 = Image.open(image5).resize(IMAGE_SHAPE)
image5 = np.array(image5)/255.0
```

```
/usr/local/lib/python3.7/dist-packages/PIL/TiffImagePlugin.py:788: UserWarnir
  warnings.warn(str(msg))
```

Prediction on example images

Load labels (class names) from Imagenet

```
# Load labels
labels_path = tf.keras.utils.get_file('ImageNetLabels.txt','https://storage.google
imagenet_labels = np.array(open(labels_path).read().splitlines())
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/
16384/10484 [==============================================] - 0s 0us/step
24576/10484 [==============================================================
```

```
print (imagenet_labels)
print ("Number of labels: " , len(imagenet_labels))
```

```
['background' 'tench' 'goldfish' ... 'bolete' 'ear' 'toilet tissue']
Number of labels:  1001
```

```
# print the entire list of categories
for i in range (len(imagenet_labels)):
  print (imagenet_labels[i])
```

```
swing
switch
syringe
table lamp
tank
tape player
teapot
teddy
television
tennis ball
thatch
theater curtain
thimble
thresher
throne
tile roof
toaster
tobacco shop
toilet seat
torch
totem pole
tow truck
toyshop
tractor
trailer truck
tray
trench coat
tricycle
trimaran
tripod
triumphal arch
trolleybus
trombone
tub
turnstile
typewriter keyboard
umbrella
unicycle
```

```
upright
vacuum
vase
vault
velvet
vending machine
vestment
viaduct
violin
volleyball
waffle iron
wall clock
wallet
wardrobe
warplane
washbasin
washer
water bottle
water jug
water tower
```

Start prediction on the selected examples

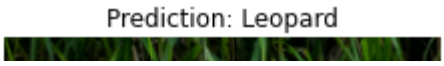
```
# infer first image
result = classifier.predict(image3[np.newaxis, ...])
result.shape
```

```
(1, 1001)
```

```
# pick up the bottom most prediction
predicted_class = np.argmax(result[0], axis=-1)
predicted_class
```

```
289
```

```
# Show image and prediction
plt.imshow(image3)
plt.axis('off')
predicted_class_name = imagenet_labels[predicted_class]
_ = plt.title("Prediction: " + predicted_class_name.title())
```



Create Predict-and-Display Function

```
def predict_display_image(imagex):
  result = classifier.predict(imagex[np.newaxis, ...])
  predicted_class = np.argmax(result[0], axis=-1)
  plt.imshow(imagex)
  plt.axis('off')
  predicted_class_name = imagenet_labels[predicted_class]
  _ = plt.title("Prediction: " + predicted_class_name.title())
```

Again ... start prediction by the function

```
# predict and print results for image1
predict_display_image(image1)
```


```
# predict and print results for image2
predict_display_image(image2)
```

```
# predict and print results for image3
predict_display_image(image3)
```


```
# predict and print results for image4
predict_display_image(image4)
```


```

```

✓ 0s completed at 7:59 PM

Нейронні мережі

Лекція_05

********** **********

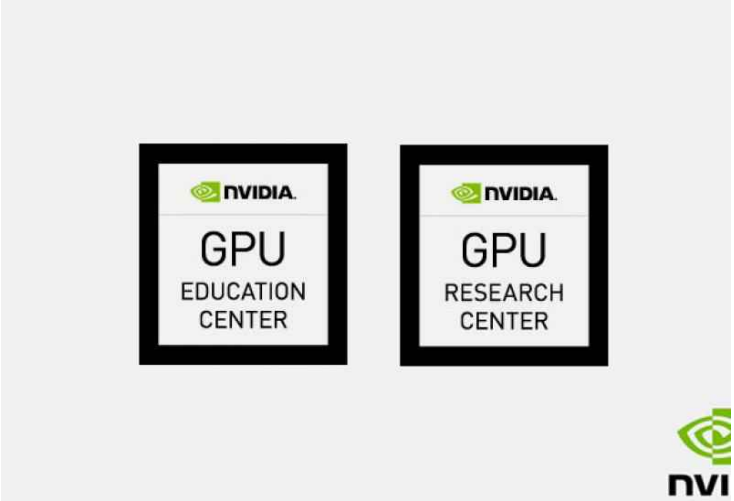
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 05 - Нейронні мережі з середини - Симулятори

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Посилання на DEMO для самостійного навчання:

Що робить TPU налаштованими для глибокого навчання?
https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning

Анімована презентація:
https://storage.googleapis.com/nexttpu/index.html

Відео-версія - 7:14 хв
https://youtu.be/BV1HMEA5f2o

Інтерактивні демонстрації
********** **********
NN - візуальна ДЕМО
Майданчик для нейронної мережі - TensorFlow
Це техніка створення комп’ютерної програми, яка вивчає дані.
https://playground.tensorflow.org

Deep playground — це інтерактивна візуалізація нейронних мереж, написана на TypeScript з використанням d3.js
Коди:
https://github.com/tensorflow/playground

Додаткові вправи:
https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/playground-exercises

Лекс Фрідман на ігровому майданчику Tensorflow – 5 хв
https://www.youtube.com/watch?v=i3ZnDRrmFjg

********** **********
CNN - візуальна ДЕМО
https://poloclub.github.io/cnn-explainer/

Демонстраційне відео "CNN Explainer: вивчення згорткових нейронних мереж за допомогою інтерактивної візуалізації" - 3 хв.
https://www.youtube.com/watch?v=HnWIHWFbuUQ

Стаття
CNN Explainer: Вивчення згорткових нейронних мереж за допомогою інтерактивної візуалізації
https://arxiv.org/abs/2004.15004

Нейронні мережі
-

Лекція 05. NN/DNN/CNN симуляторами в TensorFlow

(на основі (C) Джея Ванга, Роберта Турко, Омара Шейха, Хекю Парка, Нілакша Даса, Фреда Хомана, Мінсука Канга)

Зміст

Рекомендовані джерела — Книги

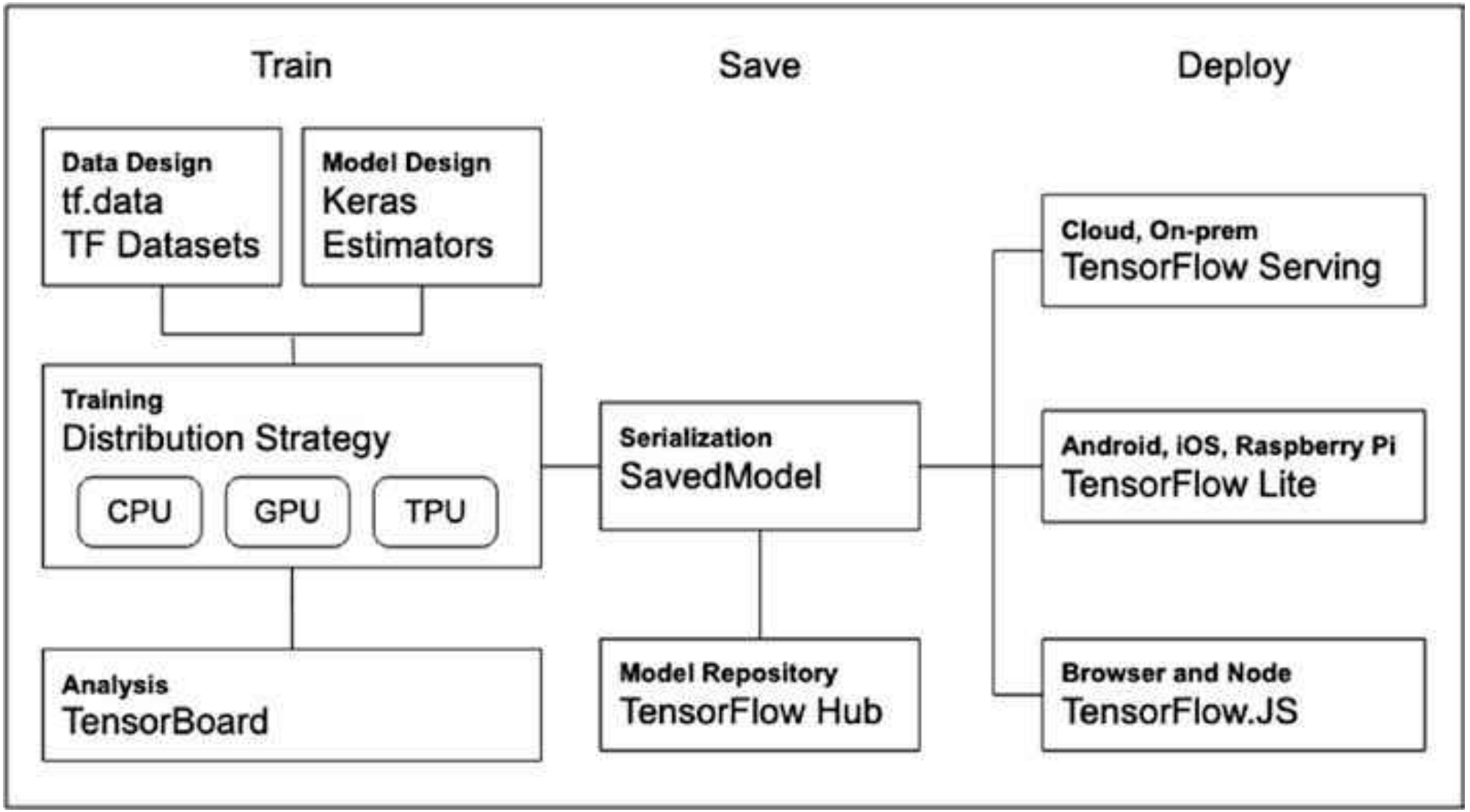
Книги (наукові):

Гудфелло, І., Бенгіо, Ю., Курвіль, А. (2016). Глибоке навчання. Кембридж: MIT Press

Цитовано в 23692 джерелах.

Зміст

Нагадаємо...DL Framework —Екосистема (Приклад Tensorflow)



Знайомство з CPU/GPU/TPU від Google

Що робить TPU налаштованими для глибокого навчання?
https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning

Анімована презентація:
https://storage.googleapis.com/nexttpu/index.html

https://youtu.be/BV1HMEA5f2o

Зміст

Симулятори DNN —основи

Ігровий майданчик ТФ

https://playground.tensorflow.org

Пояснювач CNN

https://poloclub.github.io/cnn-explainer/

https://www.youtube.com/watch?v=HnWIHWFbuUQ

Зміст

Симулятори DNN —Ігровий майданчик ТФ

Ігровий майданчик ТФ

https://playground.tensorflow.org

Майданчик для нейронної мережі - TensorFlow

Це техніка для**будівля**комп'ютерна програма що**вчиться**з даних. https://playground.tensorflow.org

Глибокий майданчик є**інтерактивна візуалізація**нейронних мережах, написано в**TypeScript**використовуючи**d3.js**

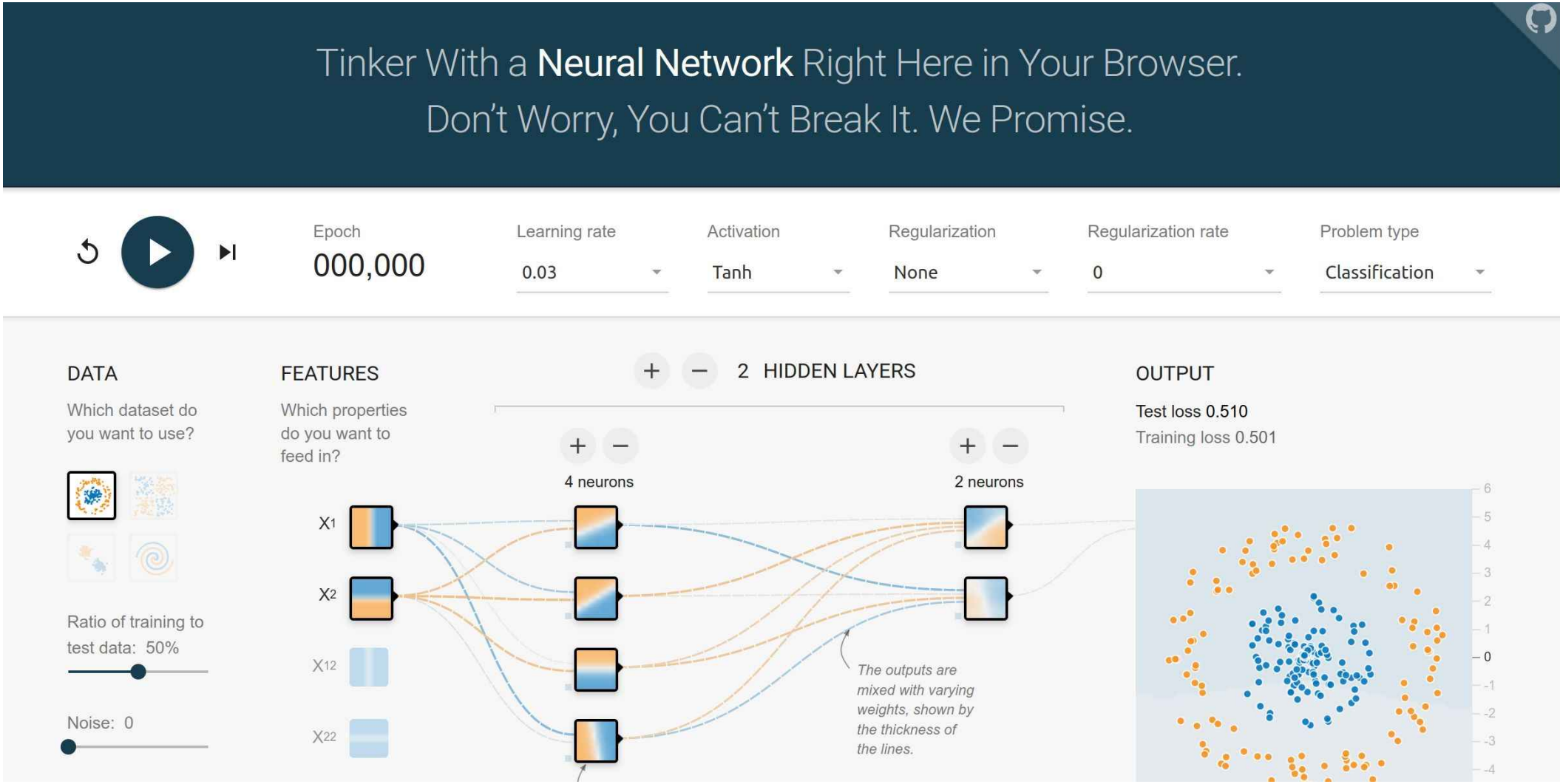
Додаткові завдання:

https://developers.google.com/machine-learning/crash-course/вступ до нейронних мереж/вправи на ігровому майданчику

Коди:

https://github.com/tensorflow/playground

Симулятори DNN —Ігровий майданчик ТФ



Ігровий майданчик ТФ

https://playground.tensorflow.org

Зміст

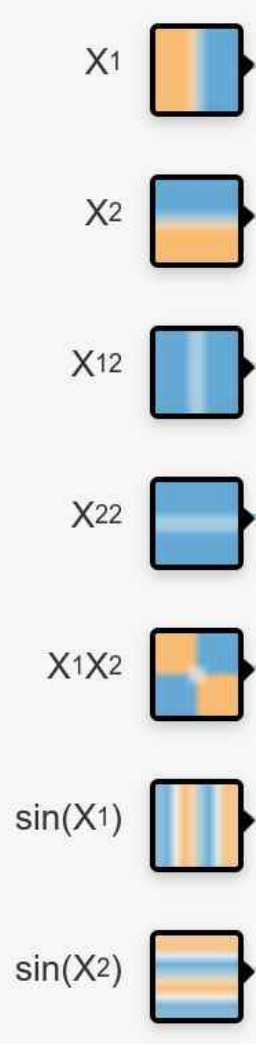
Ігровий майданчик TF —

Завдання 1: Набір даних кола

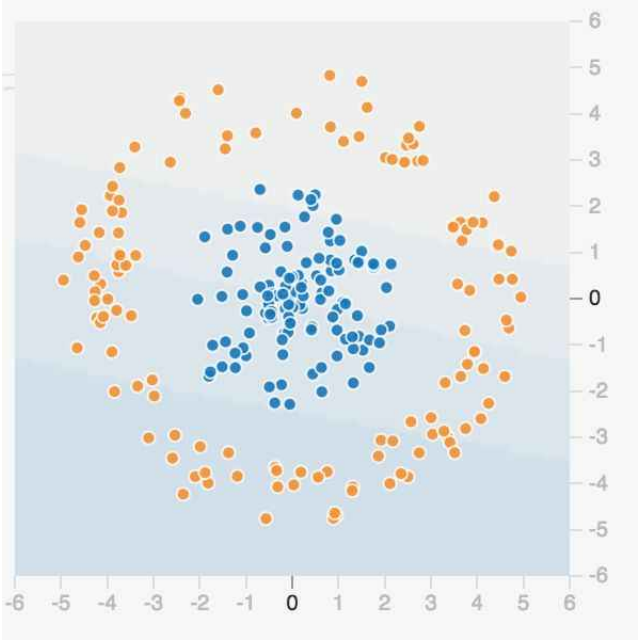
можливо

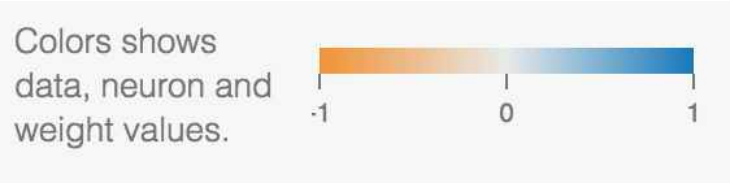
особливості:

Проблема класифікації:
знайдіть межу між 2 наборами
об’єктів з їх координатами X1, X2
і синьо-жовті етикетки



Завдання 1: Набір даних кола

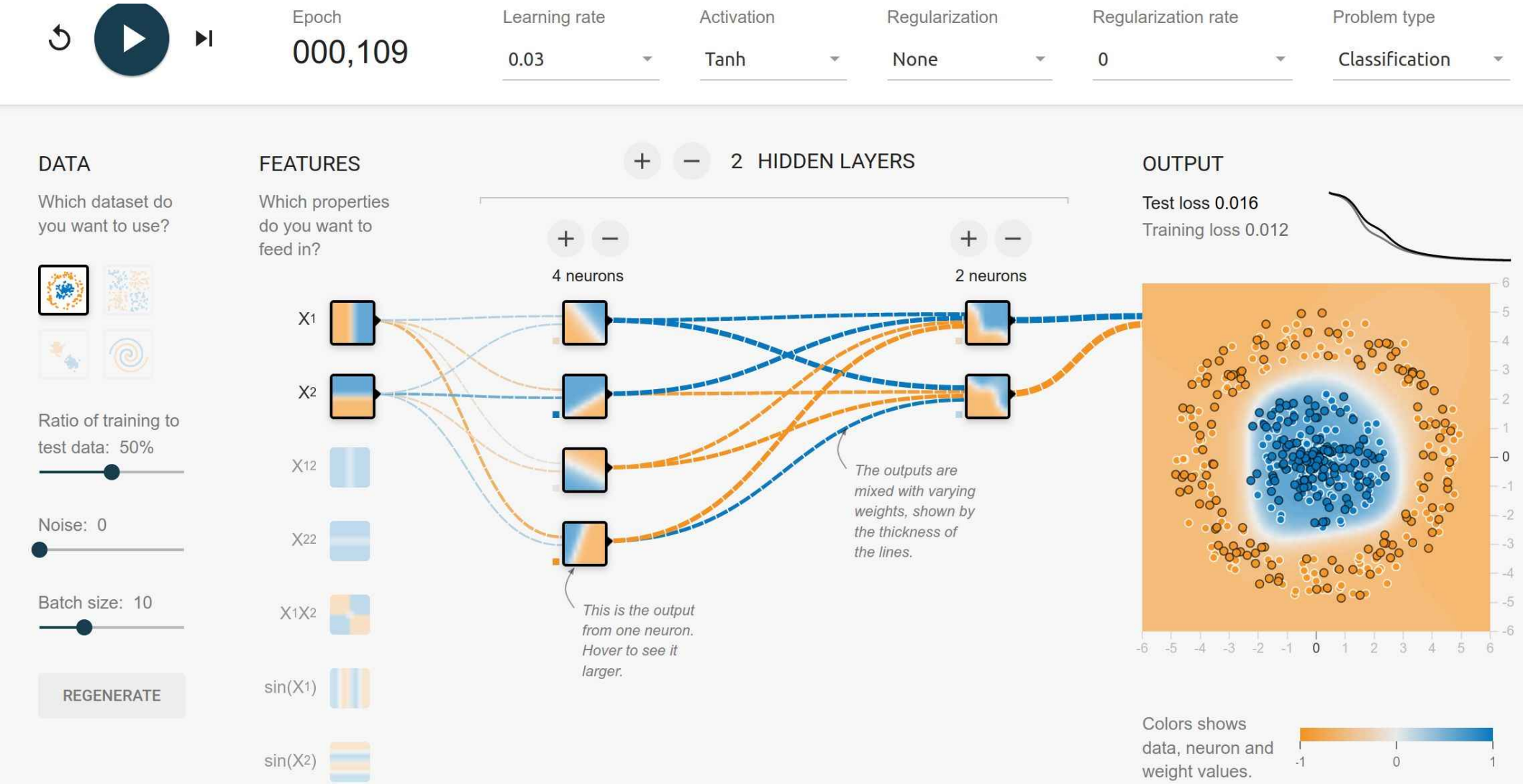




Синій означає +1,
помаранчевий означає -1, і
білий означає 0.

Ігровий майданчик TF —

Завдання 1: Circle Dataset — Рішення

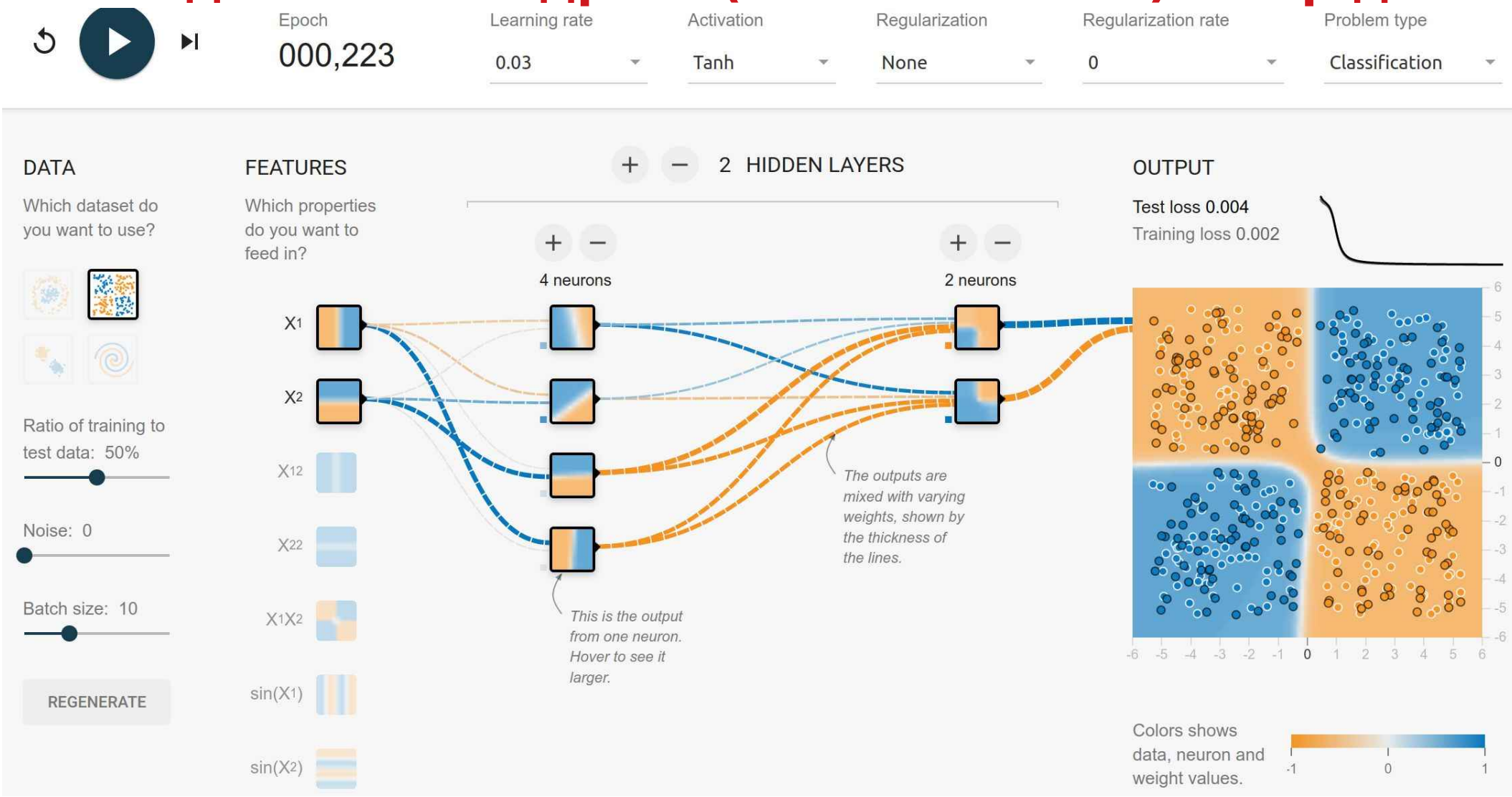


Дуже просто для гіперпараметрів за замовчуванням ... навіть: ~100 епох.

Зміст

Ігровий майданчик TF —

Завдання 2: Квадрант (виключне АБО) набір даних

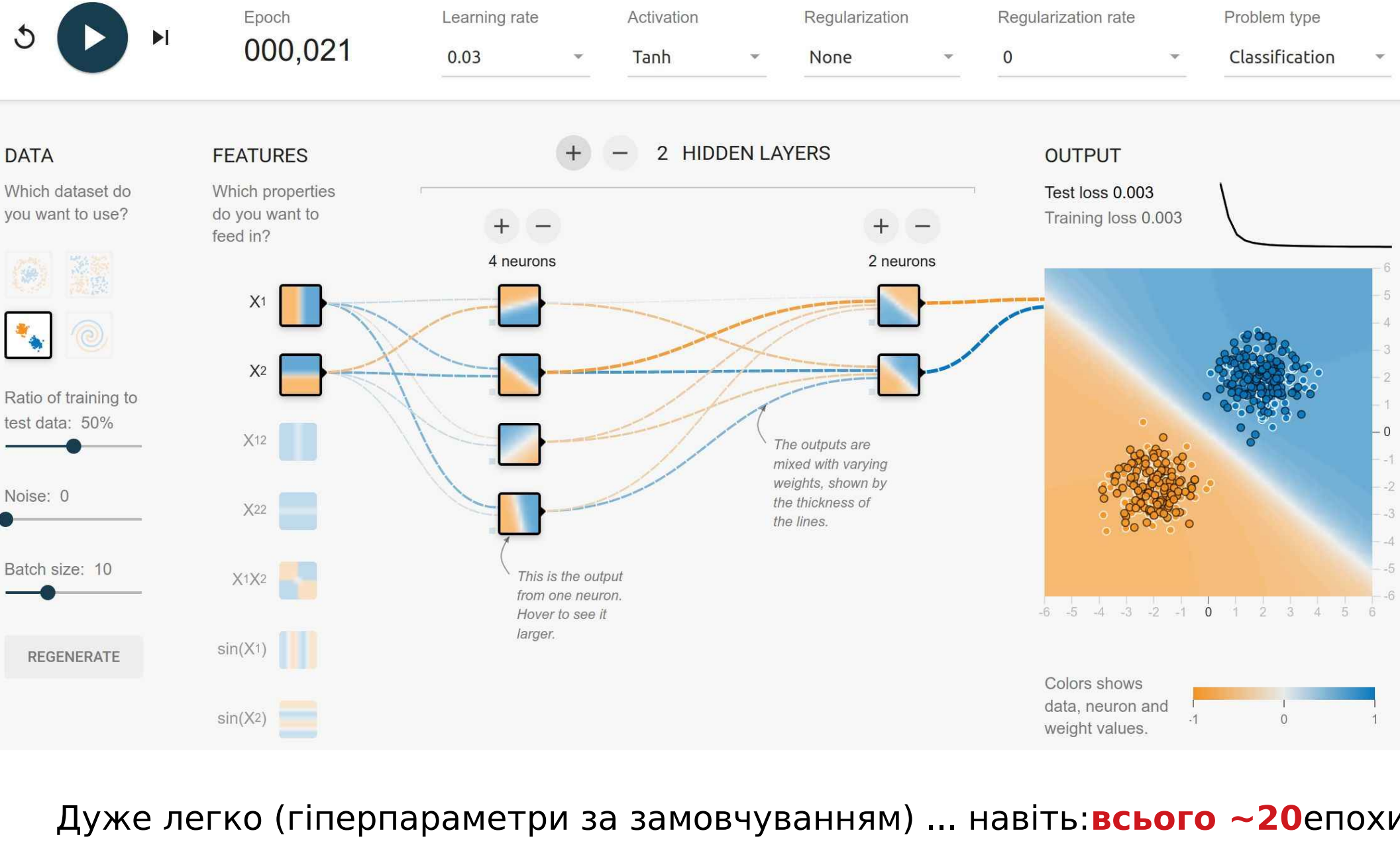


Легко для гіперпараметрів за замовчуванням ... навіть: ~**200**епохи.

Зміст

Ігровий майданчик TF —

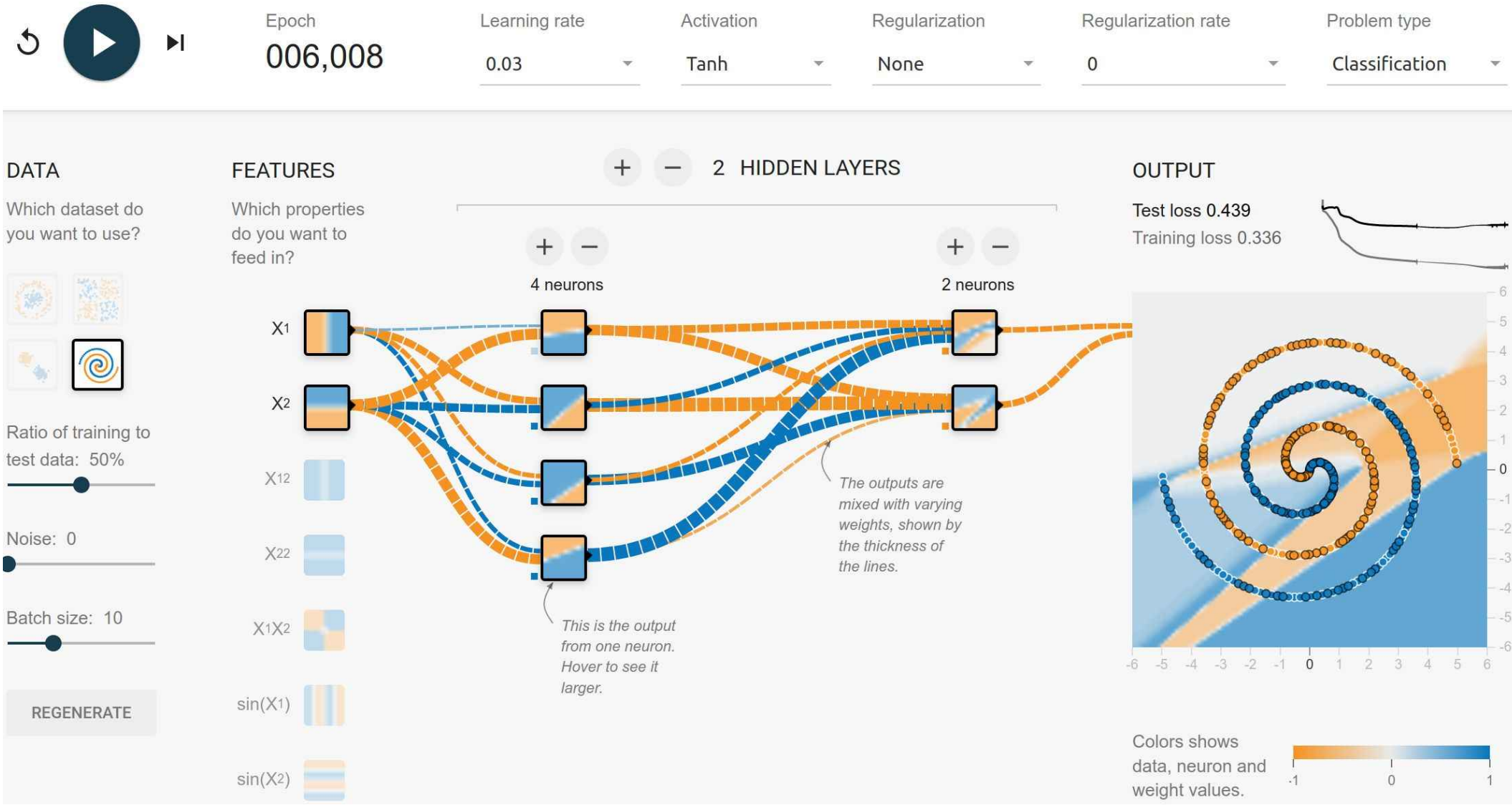
Завдання 3: Набір даних кластерів Гауса



Дуже легко (гіперпараметри за замовчуванням) ... навіть: **всього ~20** епохи

Зміст

Ігровий майданчик TF —



дуже**важко**(Гіперпараметри за замовчуванням):**NO розчин >6000**епохи

Ігровий майданчик TF —

Завдання 4: Спіральний набір даних

дуже **важко**(Гіперпараметри за замовчуванням):

БЕЗ рішення >6000 епох.

Спробуємо знайти рішення двома способами:

а)**Розробка функцій** як в **класичний ML**:

Додати новий**нелінійний**функції введення (**8 всього**): X_1, X_2, $X_1 * X_2$, X_1^2, $\sin(X_1)$ і $\sin(X_2)$ і введіть їх в а**неглибокий**(1-прихований шар) нейронна мережа.

Б)**Додайте шари** як в **класичний DL**:

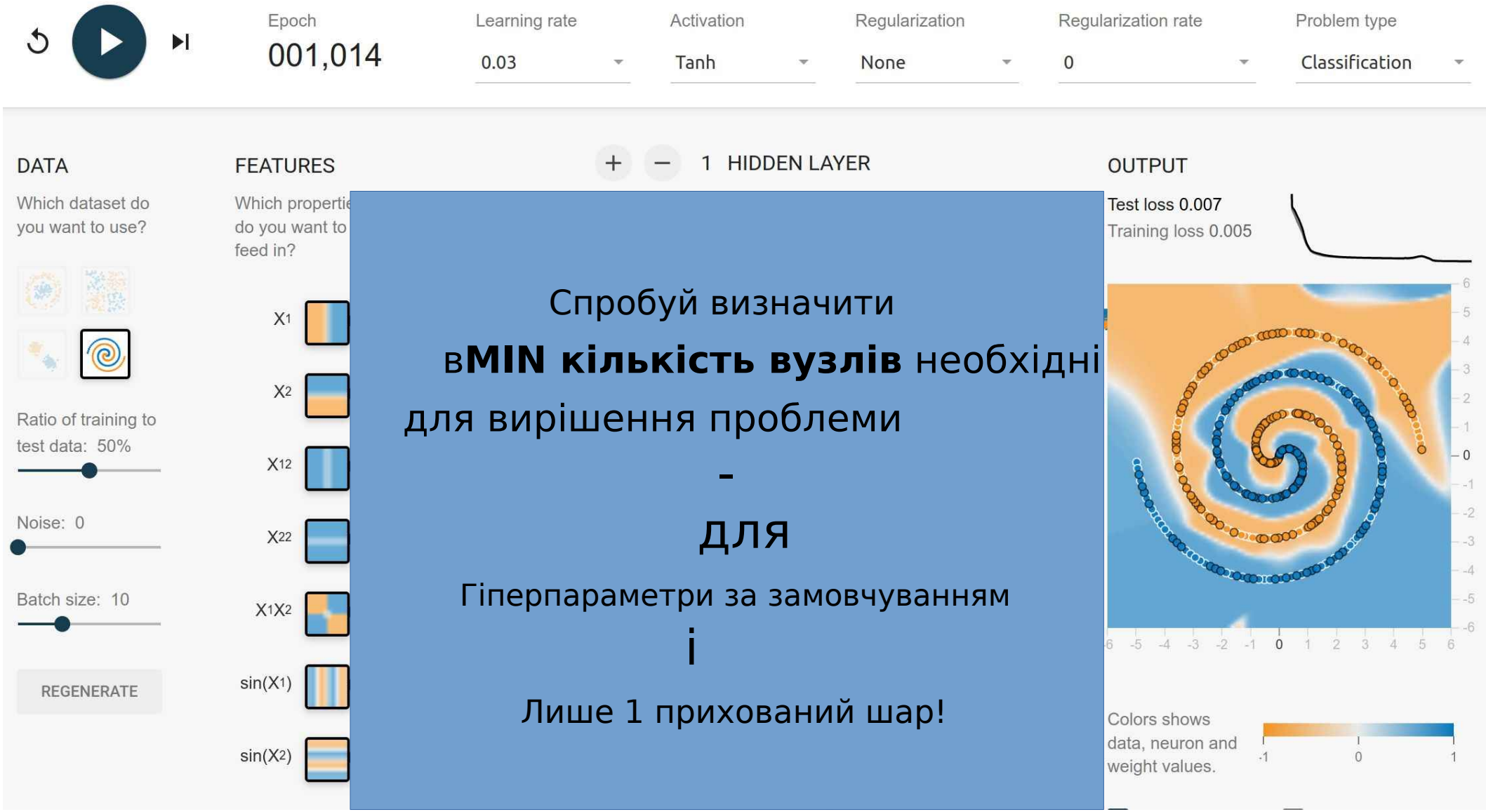
Ті самі лише 2 функції в 6-прихований шар DNN.

Спробуйте відтворити ці два рішення!

Якщо хочете — перейдіть до відповідей пізніше...

Ігровий майданчик TF —

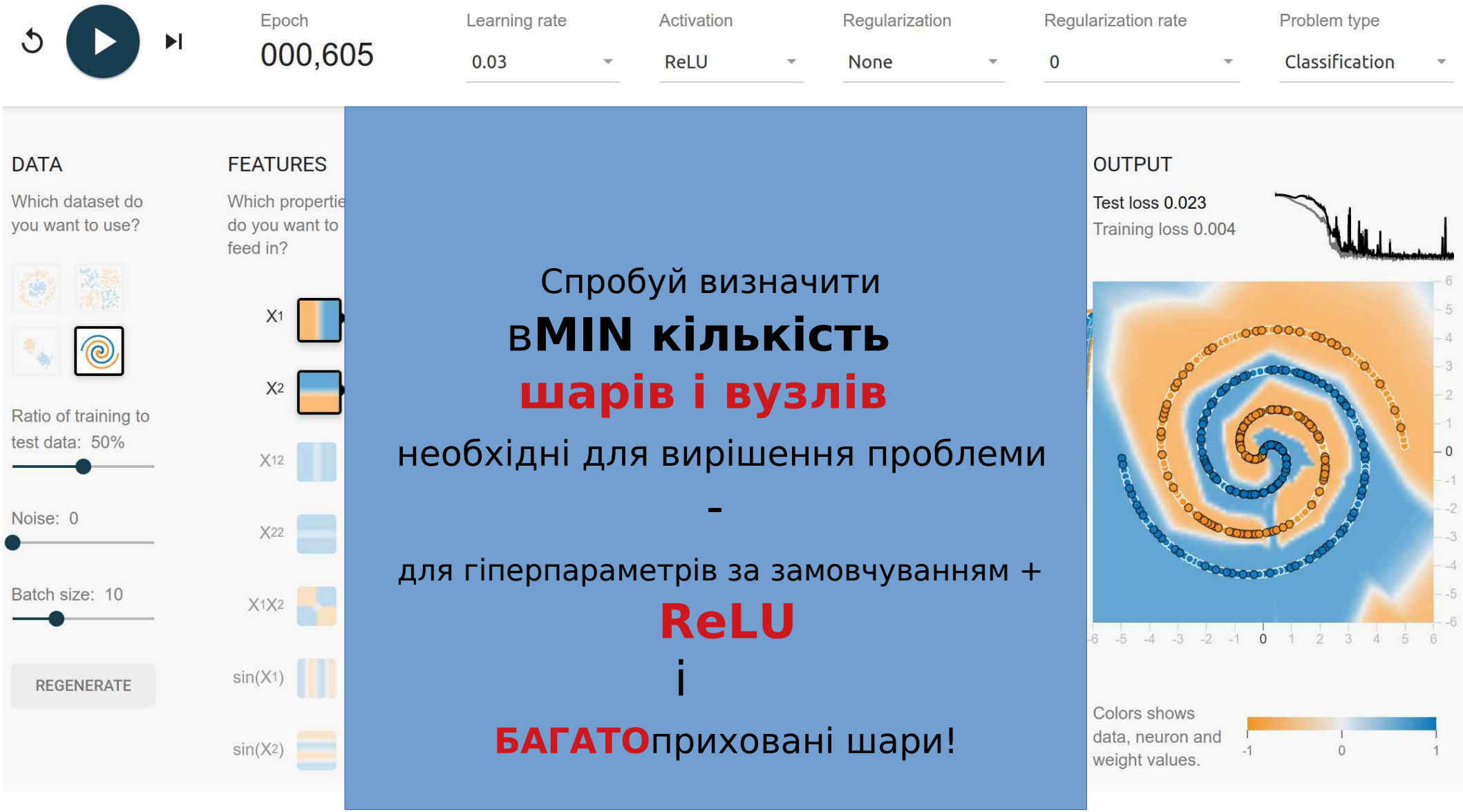
Завдання 4: Спіральний набір даних — розробка функці



Легко (гіперпараметри за замовчуванням):**розчин ~1000**епохи.

Ігровий майданчик TF —

Завдання 4: Спіральний набір даних — DNN



Легко (гіперпараметри за замовчуванням +**ReLU**):**розчин ~600**епох

Зміст

Симулятори DNN —Пояснювач CNN

Пояснювач CNN

https://poloclub.github.io/cnn-explainer/

CNN Explainer використовує TensorFlow.js,

у браузері**GPU-прискорення**Бібліотека DL для завантаження

попередньо підготовлена модель(**TinyVGG**) для візуалізації.

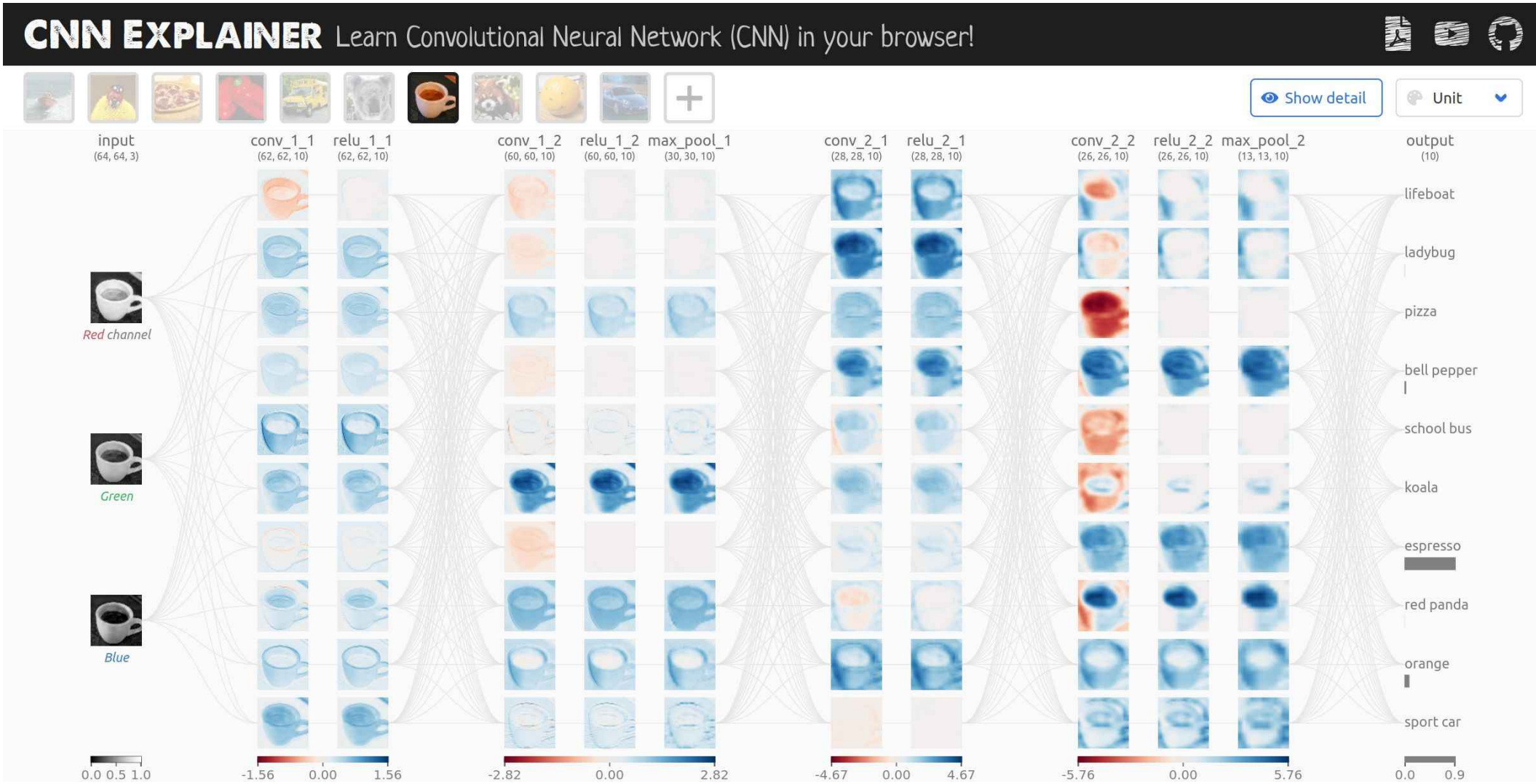
Вся інтерактивна система написана**Javascript**використовуючи

Svelte як фреймворк і D3.js для візуалізації.

потрібен лише веб-браузер щоб

почати вивчати CNN вже сьогодні!

Симулятори DNN —Пояснювач CNN



Пояснювач CNN

https://poloclub.github.io/cnn-explainer/

завдання:наведіть/клацніть мишкою на всі елементи та прочитайте їх! :)

Зміст

Симулятори DNN

—

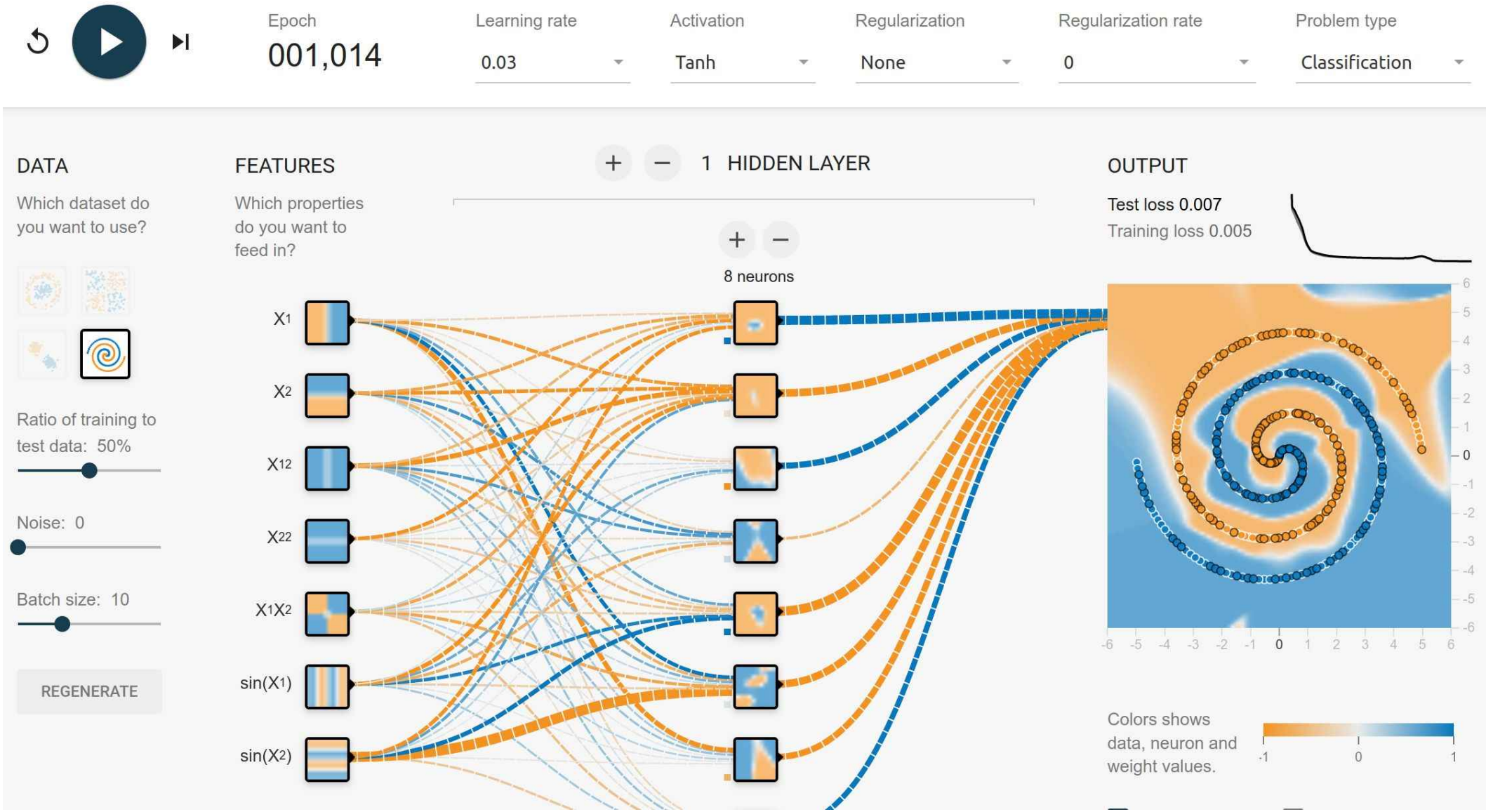
Ігровий майданчик ТФ

—

Рішення

Ігровий майданчик TF —

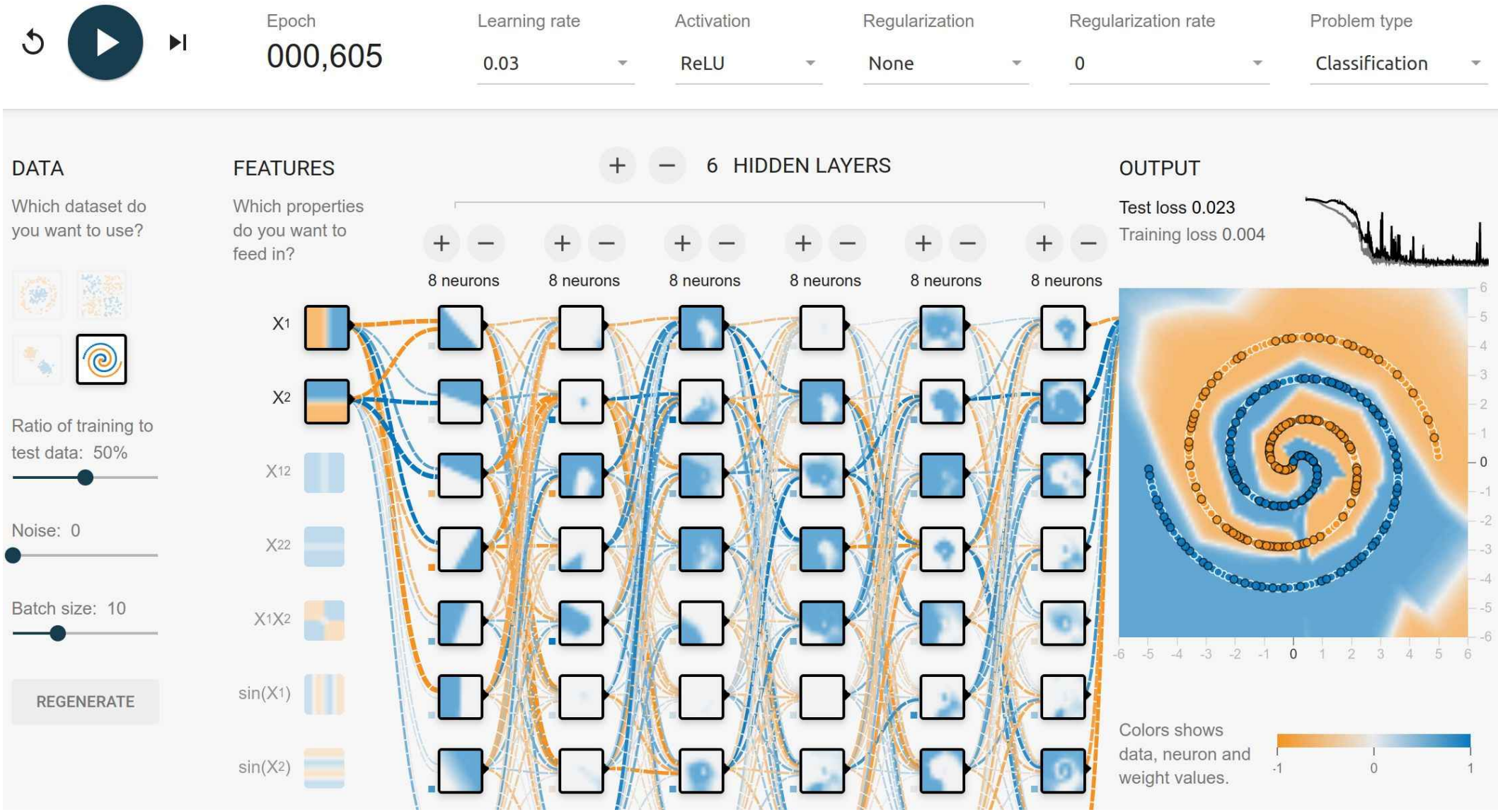
Завдання 4: Спіральний набір даних — розробка функці



8 вузлів(Гіперпараметри за замовчуванням):**~1000**епохи.

Ігровий майданчик TF —

Завдання 4: Спіральний набір даних — DNN



6 шарів з 8 вузлами(Типовий Hypers +**ReLU**):**~600**епох

Симулятори DNN

—

Ігровий майданчик ТФ

+

Не забудьте:

Додаткові завдання:

https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/playground-exercises

CNN EXPLAINER: Learning Convolutional Neural Networks with Interactive Visualization

Zijie J. Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das,
Fred Hohman, Minsuk Kahng, and Duen Horng (Polo) Chau

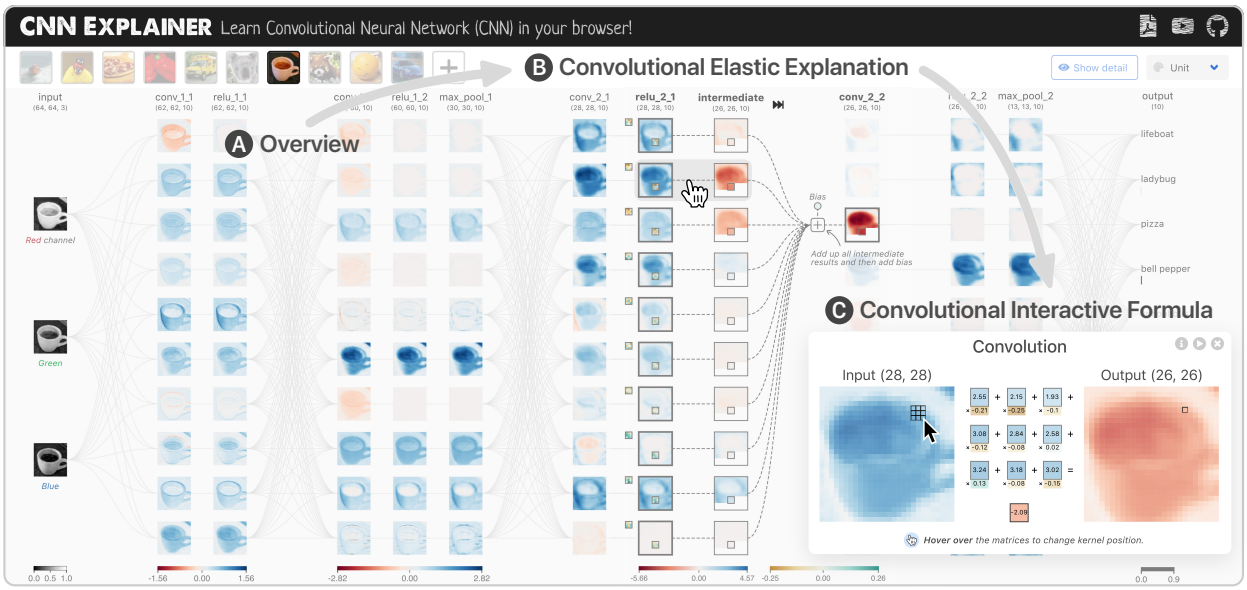


Fig. 1. With CNN EXPLAINER, learners can visually examine how *Convolutional Neural Networks* (CNNs) transform input images into classification predictions (e.g., predicting *espresso* for an image of a coffee cup), and interactively learn about their underlying mathematical operations. In this example, a learner uses CNN EXPLAINER to understand how convolutional layers work through three tightly integrated views, each explaining the convolutional process in increasing levels of detail. **(A)** The *Overview* visualizes a CNN architecture where each neuron is encoded as a square with a heatmap representing the neuron's output. **(B)** Clicking a neuron reveals how its activations are computed by the previous layer's neurons, displaying the often-overlooked intermediate computation through animations of sliding kernels. **(C)** *Convolutional Interactive Formula View* for inspecting underlying mathematics of the dot-product operation core to convolution. For clarity, some annotations are removed and views are re-positioned.

Abstract— Deep learning's great success motivates many practitioners and students to learn about this exciting technology. However, it is often challenging for beginners to take their first step due to the complexity of understanding and applying deep learning. We present CNN EXPLAINER, an interactive visualization tool designed for non-experts to learn and examine convolutional neural networks (CNNs), a foundational deep learning model architecture. Our tool addresses key challenges that novices face while learning about CNNs, which we identify from interviews with instructors and a survey with past students. CNN EXPLAINER tightly integrates a model overview that summarizes a CNN's structure, and on-demand, dynamic visual explanation views that help users understand the underlying components of CNNs. Through smooth transitions across levels of abstraction, our tool enables users to inspect the interplay between low-level mathematical operations and high-level model structures. A qualitative user study shows that CNN EXPLAINER helps users more easily understand the inner workings of CNNs, and is engaging and enjoyable to use. We also derive design lessons from our study. Developed using modern web technologies, CNN EXPLAINER runs locally in users' web browsers without the need for installation or specialized hardware, broadening the public's education access to modern deep learning techniques.

Index Terms—Deep learning, machine learning, convolutional neural networks, visual analytics

1 INTRODUCTION

Deep learning now enables many of our everyday technologies. Its continued success and potential application in diverse domains has attracted immense interest from students and practitioners who wish to learn and apply this technology. However, many beginners find it challenging to take the first step in studying and understanding deep learning concepts. For example, convolutional neural networks (CNNs), a foundational deep learning model architecture, is often one of the first and most widely used models that students learn. CNNs are often used in image classification, achieving state-of-the-art performance [33]. However, through interviews with deep learning instructors and a survey of past students, we found that even for this "introductory" model, it can be challenging for beginners to understand how inputs (e.g., image data) are transformed into class predictions. This steep learning curve stems from CNN's complexity, which typically leverages many computational layers to reach a final decision. Within a CNN, there are many types of

- *Zijie J. Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, and Duen Horng Chau are with Georgia Tech. E-mail: {jayw|rturko3|oshaikh|haekyu|nilakshdas|fredhohman|polo}@gatech.edu.*
- *Minsuk Kahng is with Oregon State University. E-mail: minsuk.kahng@oregonstate.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

network layers (e.g., fully-connected, convolutional, activation), each with a different structure and underlying mathematical operations. Thus, a student needs to develop a mental model of not only how each layer operates, but also how to choose different layers that work together to transform data. Therefore, a key challenge in learning about CNNs is the intricate interplay between *low-level mathematical operations* and *high-level integration* of such operations within the network.

Key challenges in designing learning tools for CNNs. There is a growing body of research that uses interactive visualization to explain the complex mechanisms of modern machine learning algorithms, such as TensorFlow Playground [50] and GAN Lab [29], which help students learn about dense neural networks and generative adversarial networks (GANs) respectively. Regarding CNNs, some existing visualization tools focus on demonstrating the high-level model structure and connections between layers (e.g., Harley's Node-Link Visualization [20]), while others focus on explaining the low-level mathematical operations (e.g., Karpathy's interactive CNN demo [30]). There is no visual learning tool that explains and connects CNN concepts from both levels of abstraction. This interplay between global model structure and local layer operations has been identified as one of the main obstacles to learning deep learning models, as discussed in [50] and corroborated from our interviews with instructors and student survey. CNN EXPLAINER aims to bridge this critical gap.

Contributions. In this work, we contribute:

- **CNN EXPLAINER, an interactive visualization tool designed for non-experts** to learn about both CNN's high-level model structure and low-level mathematical operations, addressing learners' key challenge in connecting unfamiliar layer mechanisms with complex model structures. Our tool advances over prior work [20, 30], overcoming unique design challenges identified from a literature review, instructor interviews and a survey with past students (Sect. 4).
- **Novel interactive system design** of CNN EXPLAINER (Fig. 1), which adapts familiar techniques such as *overview + detail* and *animation* to simultaneously summarize intricate model structure, while providing context for users to inspect detailed mathematical operations. CNN EXPLAINER's visualization techniques work together through fluid transitions between different abstraction levels (Fig. 2), helping users gain a more comprehensive understanding of complex concepts within CNNs (Sect. 6).
- **Design lessons distilled from user studies** on an interactive visualization tool for machine learning education. While visual and interactive approaches have been gaining popularity in explaining machine learning concepts to non-experts, little work has been done to evaluate such tools [28, 43]. We interviewed four instructors who have taught CNNs and conducted a survey with 19 students who have previously learned about CNNs to identify the needs and challenges for a deep learning educational tool (Sect. 4). In addition, we conducted an observational study with 16 students to evaluate the usability of CNN EXPLAINER, and investigated how our tool could help students better understand CNN concepts (Sect. 8). Based on these studies, we discuss the advantages and limitations of interactive visual educational tools for machine learning.
- **An open-source, web-based implementation** that broadens the public's education access to modern deep learning techniques without the need for advanced computational resources. Deploying deep learning models conventionally requires significant computing resources, e.g., servers with powerful hardware. In addition, even with a dedicated backend server, it is challenging to support a large number of concurrent users. Instead, CNN EXPLAINER is developed using modern web technologies, where all results are directly and efficiently computed in users' web browsers (Sect. 6.7). Therefore, anyone can access CNN EXPLAINER using their web browser without the need for installation or a specialized backend. Our code is open-sourced[1] and

[1]Code: https://github.com/poloclub/cnn-explainer

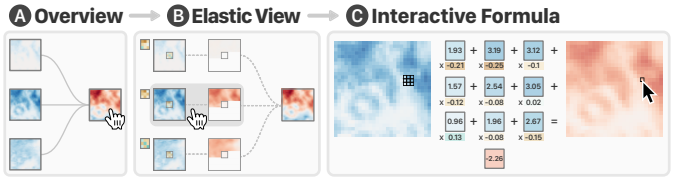


Fig. 2. In CNN EXPLAINER, tightly integrated views with different levels of abstractions work together to help users more easily learn about the intricate interplay between a CNN's high-level structure and low-level mathematical operations. **(A)** the *Overview* summarizes connections of all neurons; **(B)** the *Elastic View* animates the intermediate convolutional computation of the user-selected neuron in the *Overview*; and **(C)** *Interactive Formula* interactively demonstrates the detailed calculation on the selected input in the *Elastic View*.

CNN EXPLAINER is available at the following public demo link: https://poloclub.github.io/cnn-explainer.

Broadening impact of visualization for AI. In recent years, many visualization systems have been developed for deep learning, but very few are designed for non-experts [20, 29, 44, 50], as surveyed in [23]. CNN EXPLAINER joins visualization research that introduces beginners to modern machine learning concepts. Applying visualization techniques to explain the inner workings of complex models has great potential. We hope our work will inspire further research and development of visual learning tools that help democratize and lower the barrier to understanding and applying artificial intelligent technologies.

2 BACKGROUND FOR CONVOLUTIONAL NEURAL NETWORKS

This section provides a high-level overview of convolutional neural networks (CNNs) in the context of image classification, which will help ground our work throughout this paper.

Image classification has a long history in the machine learning research community. The objective of supervised image classification is to map an input image, X, to an output class, Y. For example, given a cat image, a sophisticated image classifier would output a class label of "cat". CNNs have demonstrated state-of-the-art performance on this task, in part because of their multiple layers of computation that aim to learn a better representation of image data.

CNNs are composed of several different layers (e.g., convolutional layers, downsampling layers, and activation layers)—each layer performs some predetermined function on its input data. Convolutional layers "extract features" to be used for image classification, with early convolutional layers in the network extracting low-level features (e.g., edges) and later layers extracting more-complex semantic features (e.g., car headlights). Through a process called backpropagation, a CNN learns kernel weights and biases from a collection of input images. These values also known as parameters, which summarize important features within the images, regardless of their location. These kernel weights slide across an input image performing an element-wise dot-product, yielding intermediate results that are later summed together with the learned bias value. Then, each neuron gets an output based on the input image. These outputs are also called activation maps. To decrease the number of parameters and help avoid overfitting, CNNs downsample inputs using another type of layer called pooling. Activation functions are used in a CNN to introduce non-linearity, which allows the model to learn more complex patterns in data. For example, a Rectified Linear Unit (ReLU) is defined as $\max(0, x)$, which outputs the positive part of its argument. These functions are also often used prior to the output layer to normalize classification scores, for example, the activation function called Softmax performs a normalization on unscaled scalar values, known as logits, to yield output class scores that sum to one. To summarize, compared to classic image classification models that can be over-parameterized and fail to take advantage of inherent properties in image data, CNNs create spatially-aware representations through multiple stacked layers of computation.

3 Related Work

3.1 Visualization for Deep Learning Education

Researchers and practitioners have been developing visualization systems that aim to help beginners learn about deep learning concepts. Teachable Machine [9] teaches the basic concept of machine learning classification, such as overfitting and underfitting, by allowing users to train a deep neural network classifier with data collected from their own webcam or microphone. The Deep Visualization Toolbox [58] also uses live webcam images to interactively help users to understand what each neuron has learned. These deep learning educational tools feature direct model manipulation as core to their experience. For example, users learn about CNNs, dense neural networks, and GANs through experimenting with model training in ConvNetJS MNIST demo [30], TensorFlow Playground [50], and GAN Lab [29], respectively. Beyond 2D visualizations, Node-Link Visualization [20] and TensorSpace [3] demonstrate deep learning models in 3D space. Inspired by Chris Olah's interactive blog posts [44], interactive articles explaining deep learning models with interactive visualization are gaining popularity as an alternative medium for education [10, 39].

Most existing educational resources focus on explaining either the high-level model structures and training process or the low-level mathematics, but not both. However, we found that one key challenge for beginners learning about deep learning models is the difficulty connecting unfamiliar layer mechanisms with complex model structures (discussed in Sect. 4). For example, TensorFlow Playground [50], one of the few yet popular deep learning educational tools, focuses on helping users develop intuition about the effects of different *dense neural network* architectures, but does not explain the underlying mathematical operations. TensorFlow Playground also operates on synthetic 2D data, which can be challenging for users to transfer newly learned concepts to more realistic data and models. In comparison, our work explains both model structure and mathematics of *CNNs*, a more complex architecture, with real image data.

3.2 Algorithm Visualization

Before deep learning started to attract interest from students and practitioners, visualization researchers have been studying how to design algorithm visualizations (AV) to help people learn about dynamic behavior of various algorithms [7, 26, 48]. These tools often graphically represent data structures and algorithms using interactive visualization and animations [7, 14, 18]. While researchers have found mixed results on AV's effectiveness in computer science education [8, 13, 16], growing evidence has shown that student engagement is the key factor for successfully applying AV in education settings [25, 42]. Naps, et al. defined a taxonomy of six levels of engagement[2] at which learners can interact with AVs [42], and studies have shown higher engagement level leads to better learning outcomes [13, 19, 32, 47].

Deep learning models can be viewed as specialized algorithms comprised of complex and stochastic interactions between multiple different computational layers. However, there has been little work in designing and evaluating visual educational tools for deep learning in the context of AV. CNN Explainer's design draws inspiration from the guidelines proposed in AV literature (discussed in Sect. 5); our user study results also corroborate some of the key findings in prior AV research (discussed in Sect. 8.3). Our work advances AV's landscape in covering modern and pervasive machine learning algorithms.

3.3 Visual Analytics for Neural Networks & Predictions

Many visual analytics tools have been developed to help deep learning experts analyze their models and predictions [5, 15, 23, 27, 36, 37]. These tools support many different tasks. For example, recent work such as Summit [24] uses interactive visualization to summarize what features a CNN model has learned and how those features interact and attribute to model predictions. LSTMVis [54] makes long short-term memory (LSTM) networks more interpretable by visualizing the model's hidden states. Similarly, GANVis [56] helps experts to interpret what a trained generative adversarial network (GAN) model has learned. People also use visual analytics tools to diagnose and monitor the training process of deep learning models. Two examples, DGMTracker [35] and DeepEyes [46], help developers better understand the training process of CNNs and GANs, respectively. Also, visual analytics tools recently have been developed to help experts detect and interpret the vulnerability in their deep learning models [12, 34]. These existing analytics tools are designed to assist experts in analyzing their model and predictions, however, we focus on non-experts and learners, helping them more easily learn about deep learning concepts.

[2] Six engagement categories: *No Viewing*, *Viewing*, *Responding*, *Changing*, *Constructing*, *Presenting*.

4 Formative Research & Design Challenges

Our goal is to build an interactive visual learning tool to help students gain understanding of key CNN concepts to design their own models. To identify the learning challenges faced by the students, we conducted interviews with deep learning instructors and surveyed past students.

Instructor interviews. To inform our tool's design, we recruited 4 instructors (2 female, 2 male) who have taught CNNs in a large university. We refer to them as T1-T4 throughout our discussion. One instructor teaches computer vision, and the others teach deep learning. We interviewed them one-on-one in a conference room (3/4) and via a video-conferencing software (1/4); each interview lasted around 30 minutes. Through these semi-structured interviews, we learned that (1) instructors currently rely on simple illustrations with toy examples to explain CNN concepts, and an interactive tool like TensorFlow Playground with real image inputs would be highly appreciated; and (2) key challenges exist for instructors teaching and students learning about CNNs, which informed us to design a student survey.

Student survey. After the interviews, we recruited students from a large university who have previously studied CNNs to fill out an online survey. We received 43 responses, and 19 of them (4 female, 15 male) met the criteria. Among these 19 participants, 10 were Ph.D. students, 3 were M.S. students, 5 were undergraduates, and 1 was a faculty member. We asked participants what were "the biggest challenges in studying CNNs" and "the most helpful features if there was a visualization tool for explaining CNNs to beginners". We provided pre-selected options based on the prior instructor interviews, but participants could write down their own response if it was not included in the options. The aggregated results of this survey are shown in Fig. 3.

Together with a literature review, we synthesized our findings from these two studies into the following five design challenges (C1-C5).

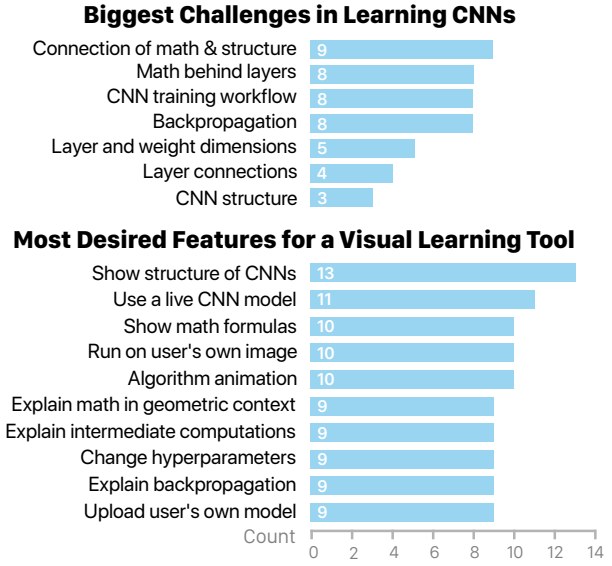


Fig. 3. Survey results from 19 participants who have previously learned about CNNs. **Top:** Biggest challenges encountered during learning. **Bottom:** Desired features for an interactive visual learning tool for CNNs.

C1. Intricate model structure. CNN models consist of many layers, each having a different structure and underlying mathematical functions [33]. Fewer past students listed CNN structure as their biggest challenge, but most of them believe a visual learning tool should explain the structure (Fig. 3), as the complex construction of CNNs can be overwhelming, especially for beginners who just started learning. T2 said "*It can be very hard for them [students with less knowledge of neural networks] to understand the structure of CNNs, you know, the connections between layers.*"

C2. Complex layer operations. Different layers serve different purposes in CNNs [17]. For example, convolutional layers exploit the spatially local correlations in inputs—each convolutional neuron connects to only a small region of its input; whereas max pooling layers introduce regularization to prevent overfitting. T1 said, "*The most challenging part is learning the math behind it [CNN model].*" Many students also reported that CNN layer computations are the most challenging learning objective (Fig. 3). To make CNNs perform better than other models in tasks like image classification, these models have complex and unique mathematical operations that many beginners may not have seen elsewhere.

C3. Connection between model structure and layer operation. Based on instructor interviews and the survey results from past students (Fig. 3), one of the cruxes to understand CNNs is understanding the interplay between low-level mathematical operations (**C2**) and the high-level model structure (**C1**). Smilkov et al., creators of the popular dense neural network learning tool Tensorflow Playground [50], also found this challenge key to learning about deep learning models: "*It's not trivial to translate the equations defining a deep network into a mental model of the underlying geometric transformations [change of feature representations].*" In other words, in addition to comprehending the mathematical formulas behind different layers, students are also required to understand how each operation works within the complex, layered model structure.

C4. Effective algorithm visualization (AV). The success of applying visualization to explain machine learning algorithms to beginners [9,29,50] suggests that an AV tool is a promising approach to help people more easily learn about CNNs. However, AV tools need to be carefully designed to be effective in helping learners gain an understanding of algorithms [13]. In particular, AV systems need to clearly explain the mapping between the algorithm and its visual encoding [40], and actively engage learners [32].

C5. Challenge in deploying interactive learning tools. Most neural networks are written in deep learning frameworks, such as TensorFlow [4] and PyTorch [45]. Although these libraries have made it much easier to create AI models, they require users to understand key concepts of deep learning in the first place [53]. Can we make understanding CNNs more accessible without installation and coding, so that everyone has the opportunity to learn and interact with deep learning models?

The above design challenges cover most of the desired features (Fig. 3). We assessed the feasibility to also support explaining backpropagation in the same tool, and we concluded that its effective explanation will necessitate designs that are hard to be unified (e.g., backpropagation Algorithm [2]) . Indeed, T1 commented that "*Deriving backpropagation is applying a series chain rules [...] It doesn't really make sense to visualize the gradients [in our tool].*" Supporting the training process would require client-side in-browser computation on many data examples, which incur both high amount of data download and slow convergence ([29,30]). Therefore, as the first prototype, we decided for CNN EXPLAINER to focus on explaining inference after a model has been trained. We plan to support the explanation for backpropagation and training process as future work (Sect. 9).

5 DESIGN GOALS

Based on the identified design challenges (Sect. 4), we distill the following key design goals (**G1G5**) for CNN EXPLAINER, an interactive visualization tool to help students more easily learn about CNNs.

G1. Visual summary of CNN models and data flow. Based on the survey results, showing the structure of CNNs is the most desired feature for a visual learning tool (Fig. 3). Therefore, to give users an overview of the structure of CNNs, we aim to create a visual summary of a CNN model by visualizing all layer outputs and connections in one view. This could help users to visually track how input image data are transformed to final class predictions through a series of layer operations (**C1**). (Sect. 6.1)

G2. Interactive interface for mathematical formulas. Since CNNs employ various complex mathematical functions to achieve high classification performance, it is important for users to understand each mathematical operation in detail (**C2**). In response, we would like to design an interactive interface for each mathematical formula, enabling users to examine and better understand the inner-workings of layers. (Sect. 6.3)

G3. Fluid transition between different levels of abstraction. To help users connect low-level layer mathematical mechanisms to high-level model structure (**C3**), we would like to design a focus + context display of different views, and provide smooth transitions between them. By easily navigating through different levels of CNN model abstraction, users can get a holistic picture of how CNN works. (Sect. 6.4)

G4. Clear communication and engagement. Our goal is to design and develop an interactive system that is easy to understand and engaging to use so that it can help people to more easily learn about CNNs (**C4**). We aim to accompany our visualizations with explanations to help users to interpret the graphical representation of the CNN model (Sect. 6.5), and we wish to actively engage learners through visualization customizations. (Sect. 6.6)

G5. Web-based implementation. To develop an interactive visual learning tool that is accessible for users without installation and coding (**C5**), we would like to use modern web browsers as the platform to explain the inner-workings of a CNN model, where users can access directly on their laptops or tablets. We also open-source our code to support future research and development of deep learning educational tools. (Sect. 6.7)

6 VISUALIZATION INTERFACE OF CNN EXPLAINER

CNN EXPLAINER's interface is built on our prior prototype [57]. We visualize the forward propagation, i.e., transforming an input image into a class prediction, of a trained model (Fig. 4). Users can explore a CNN at different levels of abstraction through the tightly integrated *Overview* (Sect. 6.1), *Elastic Explanation View* (Sect. 6.2), and the *Interactive Formula View* (Sect. 6.3). Our tool allows users to smoothly transition between these views (Sect. 6.4), provides text annotations and a tutorial article to help users interpret the visualizations (Sect. 6.5), and engages them to test hypotheses through visualization customizations (Sect. 6.6). The system is targeted towards beginners and describes all mathematical operations necessary for a CNN to classify an image.

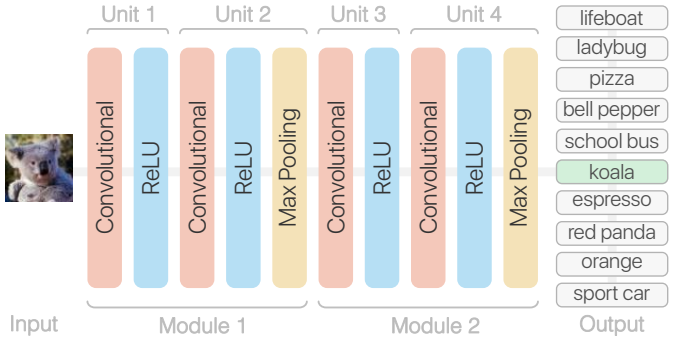


Fig. 4. Illustration of *Tiny VGG* model used in CNN EXPLAINER: this model uses the same, but fewer, convolutional layers as the original VGGNet model [49]. We trained it to classify 10 classes of images.

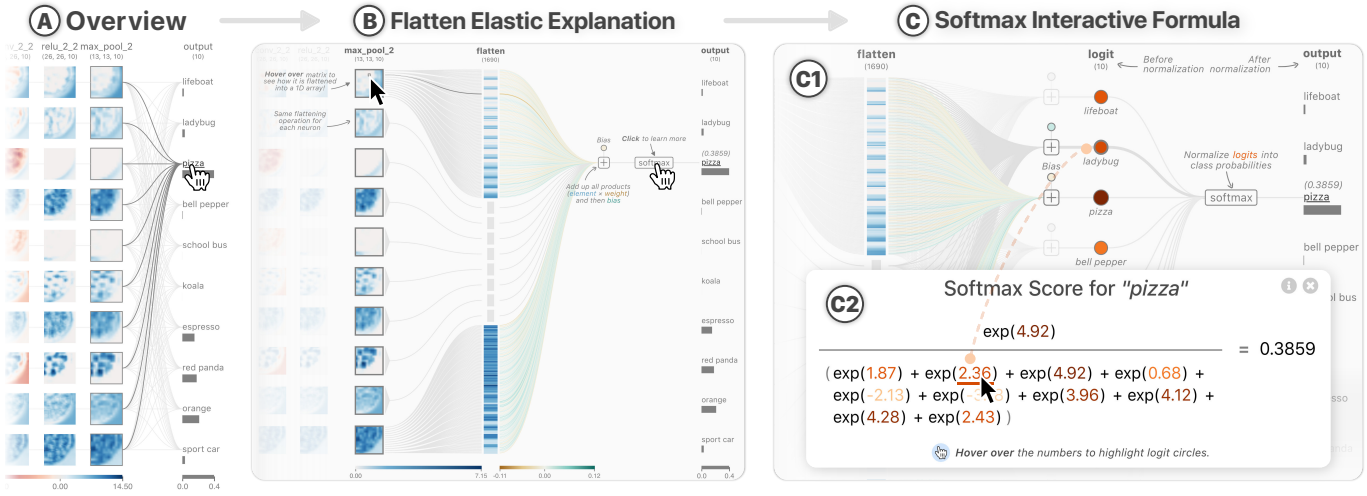


Fig. 5. CNN EXPLAINER helps users learn about the connection between the `output` layer and its previous layer via three tightly integrated views. Users can smoothly transition between these views to gain a more holistic understanding of the `output` layer's lifeboat prediction computation. **(A)** The *Overview* summarizes neurons and their connections. **(B)** The *Flatten Elastic Explanation View* visualizes the often-overlooked flatten layer, helping users more easily understand how a high-dimensional `max_pool_2` layer is connected to the 1-dimensional `output` layer. **(C)** The *Softmax Interactive Formula View* further explains how the softmax function that precedes the `output` layer normalizes the penultimate computation results (i.e., logits) into class probabilities through linking the **(C1)** numbers from the formula to **(C2)** their visual representations within the model structure.

Color scales are used throughout the visualization to show the impact of weight, bias, and activation map values. Consistently in the interface, a red to blue color scale is used to visualize neuron activation maps as heatmaps, and a yellow to green color scale represents weights and biases. A persistent color scale legend is present across all views, so the user always has context for the displayed colors. We chose these distinct, diverging color scales with white representing zero, so that a user can easily differentiate positive and negative values. We group layers in the *Tiny VGG* model, our CNN architecture, into four units and two modules (Fig. 4). Each unit starts with one convolutional layer. Both modules are identical and contain the same sequence of operations and hyperparameters. To analyze neuron activations throughout the network with varying contexts, users can alter the range of the heatmap color scale (Sect. 6.6).

6.1 Overview

The *Overview* (Fig. 1A, Fig. 5A) is the opening view of CNN EXPLAINER. This view represents the high-level structure of a CNN: neurons grouped into layers with distinct, sequential operations. It shows neuron activation maps for all layers represented as heatmaps with a diverging red to blue color scale. Neurons in consecutive layers are connected with edges, which connect each neuron to its inputs; to see these edges, users simply can hover over any activation map. In the model, neurons in convolutional layers and the `output` layer are fully connected to the previous layer, while all other neurons are only connected to one neuron in the previous layer.

6.2 Elastic Explanation View

The *Elastic Explanation Views* visualize the computations that leads to an intermediate result without overwhelming users with low-level mathematical operations. CNN EXPLAINER enters two elastic views after a user clicks a convolutional or an output neuron from the *Overview*. After the transition, far-away heatmaps and edges fade out to help users focus on the selected layers while providing CNN structural context in the background (Fig. 1A).

Explaining the Convolutional Layer (Fig. 1B). The *Convolutional Elastic Explanation View* applies a convolution on each input node of the selected neuron, visualized by a kernel sliding across the input neurons, which yields an intermediate result for each input neuron. This sliding kernel forms the output heatmap during the animation, which imitates the internal process during a convolution operation. While the sliding kernel animation is in progress, the edges in this view are represented as flowing-dashed lines; upon the animations completion, the edges transition to solid lines.

Explaining the Flatten Layer (Fig. 5B). The *Flatten Elastic Explanation View* visualizes the operation of transforming an n-dimensional tensor into a 1-dimensional tensor by traversing pixels in row-major order. This flattening operation is often necessary in a CNN prior to classification so that the fully-connected `output` layer can make classification decisions. The view represents each neuron in the `flatten` layer as a short line whose color is the same as its source pixel in the previous layer. Then, edges connect these neurons with their source components and intermediate results. These edges are colored based on the model's weight value. Users can hover over any component of this connection to highlight the associated edges as well as the `flatten` layer's neuron and the pixel value from the previous layer.

6.3 Interactive Formula View

The *Interactive Formula View* consists of four variations designed for convolutional layers, ReLU activation layers, pooling layers, and the softmax activation function. After users have built up a mental model of the CNN model structure from the previous *Overview* and *Elastic Explanation Views*, these four views demonstrate the detailed mathematics occurring in each layer.

Explaining Convolution, ReLU Activation, and Pooling (Fig. 6A, B, C)) Each view animates the window-sliding operation on the input matrix and output matrix over an interval, so that the user can understand how each element in the input is connected to the output, and vice versa. In addition, the user can interact with the these matrices by hovering over the heatmaps to control the position of the sliding window. For example, in the *Convolutional Interactive Formula View* (Sect. 6.3A), as the user controls the window (kernel) position in either the input or the output matrix, this view visualizes the dot-product formula with input numbers and kernel weights directly extracted from the current kernel. This synchronization between the input, the output and the mathematical function enables the user to better understand how the kernel convolves a matrix in convolutional layers.

Explaining the Softmax Activation (Fig. 6D). This view outlines the operations necessary to calculate the classification score. It is accessible from the *Flatten Elastic Explanation View* to explain how the results (logits) from the previous view lead to the final classification. The view consists of logit values encoded as circles and colored with a

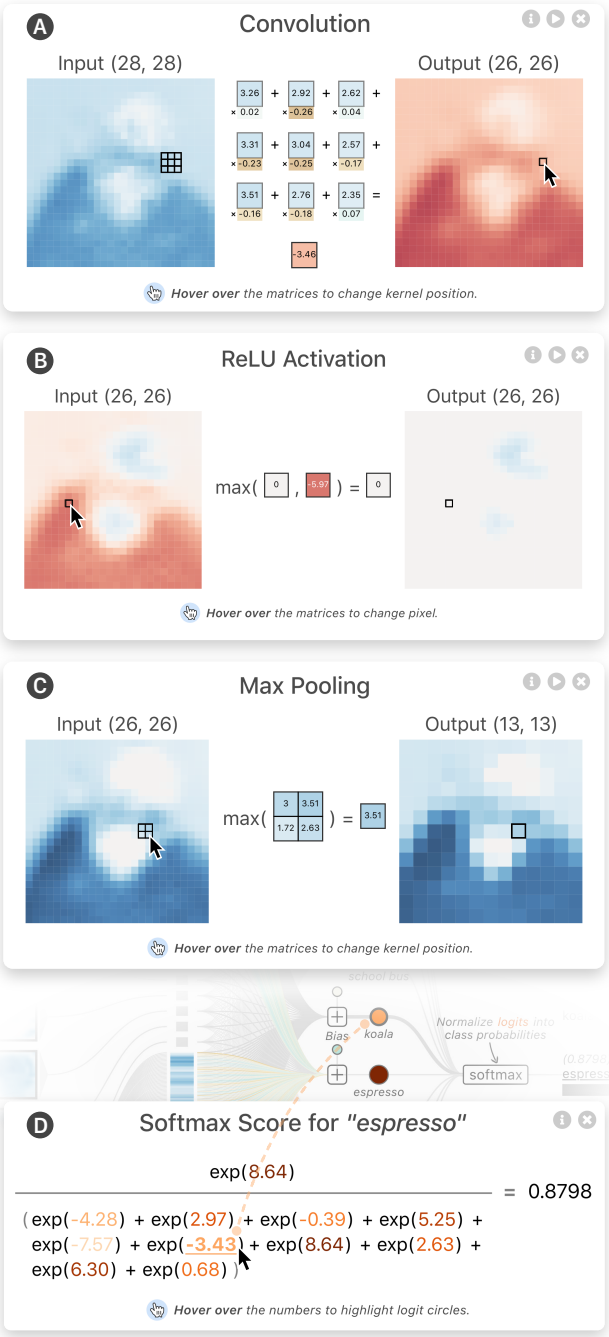


Fig. 6. The *Interactive Formula Views* explain the underlying mathematical operations of a CNN. **(A)** shows the element-wise dot-product occurring in a convolutional neuron, **(B)** visualizes the activation function ReLU, and **(C)** illustrates how max pooling works. Users can hover over heatmaps to display an operation's input-to-output mapping. **(D)** interactively explains the softmax function, helping users connect numbers from the formula to their visual representations. Users can click the info button to scroll to the corresponding section in the tutorial article, and the play button to start the window sliding animation in (A)-(C).

light orange to dark orange color scale, which provides users with a visual cue of the importance of every class. This view also includes a corresponding equation, which explains how the classification score is computed. When users enter this view, pairs of each logit circle and its corresponding value in the equation appear sequentially with animations. As a user hovers over a logit circle, its value will be highlighted in the equation along with the logit circle itself, so the user can understand how each logit contributes to the softmax function. Hovering over numbers in the equation will also highlight the appropriate logit circles. Interacting with logit circles and the mathematical equation in combination allows a user to discern the impact that every logit has on the classification score in the `output` layer.

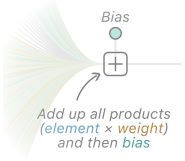
6.4 Transitions Between Views

The *Overview* is the starting state of CNN EXPLAINER and shows the model architecture. From this high-level view, the user can begin inspecting layers, connectivity, classifications, and tracing activations of neurons through the model. When a user is interested in more detail, they can click on neuron activation maps in the visualization. Neurons in a layer that have simple one-to-one connections to a neuron in the previous layer do not require an auxiliary *Elastic Explanation View*, so upon clicking one of these neurons, a user will be able to enter the *Interactive Formula View* to understand the low-level operation that a tensor undergoes at that layer. If a neuron has more complex connectivity, then the user will enter an *Elastic Explanation View* first. In this view, CNN EXPLAINER uses visualizations and annotations before displaying mathematics. Through further interaction, a user can hover and click on parts of the *Elastic Explanation View* to uncover the mathematical operations as well as examine the values of weights and biases. The low-level *Interactive Formula Views* are only shown after transitioning from the previous two views, so that users can learn about the underlying mathemtical operations after hainvg a mental model of the complex and layered CNN model structure.

6.5 Visualizations with Explanations

CNN EXPLAINER is accompanied by an interactive tutorial article beneath the interface that explains CNN layer functions, hyperparameters, and outlines CNN EXPLAINER's interactive features. Learners can read freely, or jump to specific sections by clicking layer names or the info buttons (Fig. 6) from the main visualization. The article provides beginner users detailed information regarding CNNs that can supplement their exploration of the visualization.

Additionally, text annotations are placed throughout the visualization (e.g., explaining the flatten layer operation in the right image), which further guide users and explain concepts that are not easily discernible from the visualization alone. These annotations help users map the underlying algorithm to its visual encoding.



6.6 Customizable Visualizations

The *Control Panel* located across the top of the visualization (Fig. 1) allows the user to alter the CNN input image and edit the overall representation of the network. The *Hyperparameter Widget* (Fig. 7) enables the user to experiment with differnt convolution hyperparameters.

Change input image. Users can choose between (1) preloaded input images for each output class, or (2) upload their own custom image. Preloaded images allow a user to easily access data from the classes the model was originally trained on. User can also freely upload any image for classification into the ten classes the network was trained on. CNN EXPLAINER resizes a user's image while preserving the aspect ratio to fit one dimension of the model input size, and then crop the central region if the other dimensions does not match. The fourth of six AV tool engagement levels is allowing users to change the AV tool's input [42]. Supporting custom image upload engages users, by allowing them to analyze the network's classification decisions and interactively testing their own hypotheses on diverse image inputs.

Show network details. A user can toggle the "Show detail" button, which displays additional network specifications in the *Overview*. When toggled on, the *Overview* will reveal layer dimensions and show color scale legends. Additionally, a user can vary the activation map color scale range. The CNN architecture presented by CNN EXPLAINER is grouped into four units and two modules (Fig. 4). By modifying the drop-down menu in the *Control*

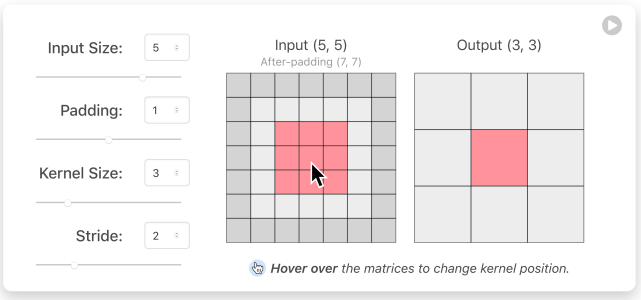


Fig. 7. The *Hyperparameter Widget*, a component of the accompanying interactive article, allows users to adjust hyperparameters and observe in real time how the kernel's sliding pattern changes in convolutional layers.

Panel, a user can adjust the color scale range used by the network to investigate activations with different groupings.

Explore hyperparameter impact. The tutorial article (Sect. 6.5) includes an interactive *Hyperparameter Widget* that allows users to experiment with convolutional hyperparameters (Fig. 7). Users can adjust the input and hyperparameters of the stand-alone visualization to test how different hyperparameters change the sliding convolutional kernel and the output's dimensions. This interactive element emphasizes learning through experimentation by supplementing knowledge gained from reading the article and using the main visualization.

6.7 Web-based, Open-sourced Implementation

CNN EXPLAINER is a web-based, open-sourced visualization tool to teach students the foundations of CNNs. A new user only needs a modern web-broswer to access our tool, no installation required. Additionally, other datasets and linear models can be quickly applied to our visualization system due to our robust implementation.

Model Training. The CNN architecture, Tiny VGG (Fig. 4), presented by CNN EXPLAINER for image classification is inspired by both the popular deep learning architecture, VGGNet [49], and Stanford's CS231n course notes [31]. It is trained on the Tiny ImageNet dataset [1]. The training dataset consists of 200 image classes and contains 100,000 64×64 RGB images, while the validation dataset contains 10,000 images across the 200 image classes. The model is trained using *TensorFlow* [4] on 10 handpicked, everyday classes: lifeboat, ladybug, bell pepper, pizza, school bus, koala, espresso, red panda, orange, and sport car. During the training process, the batch size and learning rate are fine-tuned using a 5-fold-cross-validation scheme. This simple model achieves a 70.8% top-1 accuracy on the validation dataset.

Front-end Visualization. CNN EXPLAINER loads the pre-trained Tiny VGG model and computes forward propagation results in real time in a user's web browser using *TensorFlow.js* [51]. These results are visualized using *D3.js* [6] throughout the multiple interactive views.

7 USAGE SCENARIOS

7.1 Beginner Learning Layer Connectivity

Janis is a virology researcher using CNNs in a current project. Through an online deep learning course she has a general understanding of the goals of applying CNNs, and some basic knowledge of different types of CNN layers, but she needs help filling in some gaps in knowledge. Interested in learning how a 3-dimensional input (RGB image) leads to a 1-dimensional output (vector of class probabilities) in a CNN, Janis begins exploring the architecture from the *Overview* (Fig. 5A).

After clicking the "Show detail" button, Janis notices that the `output` layer is a 1-dimensional tensor of size 10, while `max_pool_2`, the previous layer, is a 3-dimensional (13×13×10) tensor. Confused, she hovers over a neuron in the `output` layer to inspect connections between the final two layers of the architecture: the `max_pool_2` layer has 10 neurons; the `output` layer has 10 neurons each representing a class label, and the `output` layer is fully-connected to the `max_pool_2` layer. She clicks that `output` neuron, which transitions the *Overview* (Fig. 5A) to the *Flatten Elastic Explanation View* (Fig. 5B). She notices that edges between these two layers intersect a 1-dimensional flatten layer and pass through a softmax function. By hovering over pixels from the activation map, Janis understands how the 2-dimensional matrix is "unwrapped" to yield a portion of the 1-dimensional `flatten` layer. As she continues to follow the edge after the `flatten` layer, she clicks the softmax button which leads her to the *Softmax Interactive Formula View* (Fig. 5C). She learns how the outputs of the `flatten` layer are normalized by observing the equation linked with logits through animations. Janis recognizes that her previous coursework has not taught these "hidden" operations prior to the `output` layer, which flatten and normalize the output of the `max_pool_2` layer. Instead of searching through lecture videos and textbooks, CNN EXPLAINER enables Janis to learn these often-overlooked operations through a hierarchy of interactive views in a stand-alone website. She now feels more equipped to apply CNNs to her virology research.

7.2 Teaching Through Interactive Experimentation

A university professor, Damian, is currently teaching a computer vision class which covers CNNs. Damian begins his lecture with standard slides. After describing the theory of convolutions, he opens CNN EXPLAINER to demonstrate the convolution operation working inside a full CNN for image classification. With CNN EXPLAINER projected to the class, Damian transitions from the *Overview* (Fig. 1A) to the *Convolutional Elastic Explanation View* (Fig. 1B). Damian encourages the class to interpret the sliding window animation (Fig. 2B) as it generates several intermediate results. He then asks the class to predict kernel weights in a specific neuron. To test student's hypotheses, Damian enters the *Convolutional Interactive Formula View* (Fig. 1C), to display the convolution operation with the true kernel weights. In this view, he can hover over the input and output matrices to answer questions from the class, and display computations behind the operation.

Recalled from theory, a student asks a question regarding the impact of altering the stride hyperparameter on the animated sliding window in convolutional layers. To illustrate the impact of alternative hyperparameters, Damian scrolls down to the "Convolutional Layer" section of the complementary article, where he experiments by adjusting stride and other hyperparameters with the *Hyperparameter Widget* (Fig. 7) in front of the class. CNN EXPLAINER is the first software that allows Damian to explain convolutional operations and hyperparameters with real image inputs, and quickly answer students' questions in class. Previously, Damian had to draw illustrations with simple matrix inputs on slides or a chalkboard. Finally, to reinforce the concepts and encourage individual experimentation, Damian provides the class with a URL to the web-based CNN EXPLAINER for students to return to in the future.

8 OBSERVATIONAL STUDY

We conducted an observational study to investigate how CNN EXPLAINER's target users (e.g., aspiring deep learning students) would use this tool to learn about CNNs, and also to test the tool's usability.

8.1 Participants

CNN EXPLAINER is designed for deep learning beginners who are interested in learning CNNs. In this study, we aimed to recruit participants who aspire to learn about CNNs and have some knowledge of basic machine learning concepts (e.g., knowing what an image classifier is). We recruited 16 student participants from a large university (4 female, 12 male) through internal mailing lists (e.g., machine learning and computer science Ph.D., M.S., and undergraduate students). Seven participants were Ph.D. students, seven were M.S. students, and the other two were undergraduates. All participants were interested in learning CNNs, and none of them had known CNN EXPLAINER before. Participants self-reported their level of knowledge on non-neural network machine learning techniques, with an average score of 3.26 on a scale of 0 to 5 (0 being "no knowledge" and 5 being "expert"); and an average score of 2.06 on CNNs (on the same scale). No participant self-reported a score of 5 for their knowledge on CNNs, and one participant had a score of 0. To help better organize our discussion, we refer to participants with CNN knowledge score of 0, 1 or 2 as B1-B11, where

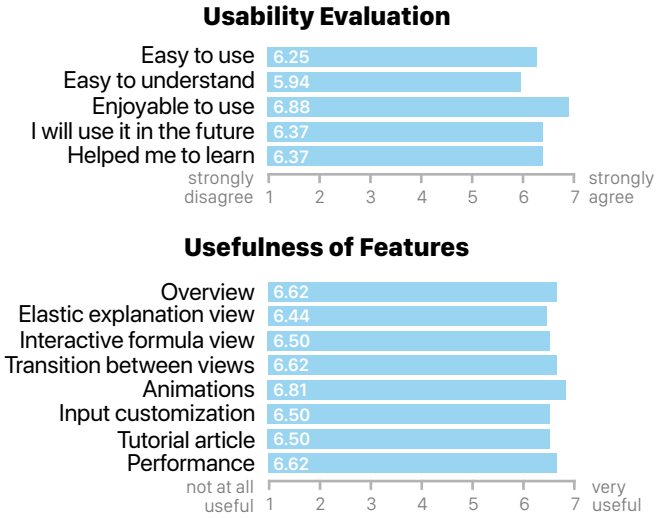


Fig. 8. Average ratings from 16 participants regarding the usability and usefulness of CNN EXPLAINER. **Top:** Participants thought CNN EXPLAINER was easy to use, enjoyable, and helped them learn about CNNs. **Bottom:** All features, especially animations, were rated favorably.

"B" stands for "Beginner"; and those with score of 3 or 4 as K1-K5, where "K" stands for "Knowledgeable."

8.2 Procedure

We conducted this study with participants one-on-one via video-conferencing software. With the permission of all participants, we recorded the participants' audio and computer screen for subsequent analysis. After participants signed consent forms, we provided them a 5-minute overview of CNNs, followed by a 3-minute tutorial of CNN EXPLAINER. Participants then freely explored our tool in their computer's web browser. We also provided a feature checklist, which outlined the main features of our tool and encouraged participants to try as many features as they could. During the study, participants were asked to think aloud and share their computer screen with us; they were encouraged to ask questions when necessary. Each session ended with a usability questionnaire coupled with an exit interview that asked participants about their process of using CNN EXPLAINER, and if this tool could be helpful for them. Each study lasted around 50 minutes, and we compensated each participant with a $10 Amazon Gift card.

8.3 Results and Design Lessons

The exit questionnaire included a series of 7-point Likert-scale questions about the utility and usefulness of different views in CNN EXPLAINER (Fig. 8). All average Likert rating were above 6 except the rating of "easy to understand". From the high ratings and our observations, participants found our tool easy to use and understand, retained a high engagement level during their session, and eventually gained a better understanding of CNN concepts. Our observations also reflect key findings in previous AV research [13, 32]. This section describes design lessons and limitations of our tool distilled from this study.

8.3.1 Transitions between different views

Transitions help users link CNN operations and structures. Several participants (9/16) commented that they liked how our tool transitions between high-level CNN structure views and low-level mathematical explanations. It helps them better understand the interplay between layer computations and the overall CNN data transformation—one of the key challenges for understanding CNN concepts, as we identified from our instructor interviews and our student survey. For example, initially K4 was confused to see the *Convolutional Elastic Explanation View*, but after reading the annotation text, he remarked, "*Oh, I understand what an intermediate layer is now—you run the convolution on the image, then you add all those results to get this.*" After exploring the *Convolutional Interactive Formula View*, he immediately noted, "*Every single aspect of the convolution layer is shown here. [This] is super helpful.*" Similarly, B5 commented, "*Good to see the big picture at once and the transition to different views [...] I like that I can hide details of a unit in a compact way and expand it when [needed].*"

CNN EXPLAINER employs the fisheye view technique for presenting the *Elastic Explanation Views* (Fig. 1B, Fig. 5B): after transitioning from the *Overview* to a specific layer, neighboring layers are still shown while further layers (lower degree-of-interest) have lower opacity. Participants found this transition design helpful for them to learn layer-specific details while having CNN structural context in the background. For instance, K5 said "*I can focus on the current layer but still know the same operation goes on for other layers.*" Our observations from this study suggest that our fluid transition design between different level of abstraction can help users to better connect unfamiliar layer mechanisms to the complex model structure.

8.3.2 Animations for enjoyable learning experience

Another favorite feature of CNN EXPLAINER that participants mentioned was the use of animations, which received the highest rating in the exit questionnaire (Fig. 8). In our tool, animations serve two purposes: to assimilate the relationship between different visual components and to help illustrate the model's underlying operations.

Transition animations help navigating. Layer movement is animated during view transitions. We noticed it helped participants to be aware of different views, and all participants navigated through the views naturally. In addition to assisting with understanding the relationship between distinct views, animation also helped them discover the linking between different visualization elements. For example, B8 quickly found that the logit circle is linked to its corresponding value in the formula, when she saw the circle-number pair appear one-by-one with animation in the *Softmax Interactive Formula View* (Fig. 5C).

Algorithm animations contribute to understanding. Animations that simulate the model's inner-workings helped participants learn underlying operations by validating their hypotheses. In the *Convolutional Elastic Explanation View* (Fig. 2B), we animate a small rectangle sliding through one matrix to mimic the CNN's internal sliding window. We noticed many participants had their attention drawn to this animation when they first transitioned into the *Convolutional Elastic Explanation View*. However, they did not report that they understood the convolution operation until interacting with other features, such as reading the annotation text or transitioning to the *Convolutional Interactive Formula View* (Fig. 2C). Some participants went back to watch the animation multiple times and commented that it made sense, for example, K5 said "*Very helpful to see how the image builds as the window slides through,*" but others, such as B9 remarked, "*It is not easy to understand [convolution] using only animation.*" Therefore, we hypothesize that this animation can indirectly help users to learn about the convolution algorithm by validating their newly formed mental models of how specific operation behave. To test this hypothesis, a rigorous controlled experiment would be needed. Related research work on the effect of animation in computer science education also found that algorithm animation does not automatically improve learning, but it may lead learners to make predictions of the algorithm behavior which in turn helps learning [8].

Animations improve learning engagement and enjoyment. We found animations helped to increase participants' engagement level (e.g., spending more time and effort) and made CNN EXPLAINER more enjoyable to use. In the study, many participants repeatedly played and viewed different animations. For example, K2 replayed the window sliding animation multiple times: "*The is very well-animated [...] I always love smooth animations.*" B7 also attributed animations to his enjoyable experience with our tool: "*[The tool is] enjoyable to use [...] I especially like the lovely animation.*"

8.3.3 Engaging learning through visualization customization

CNN EXPLAINER allows users to modify the visualization. For example, users can change the input image or upload their own image for

classification; CNN EXPLAINER visualizes the new prediction with the new activation maps in every layer. Similarly, users can interactively explore how hyperparameters affect the convolution operation (Fig. 7).

Customization enables hypothesis testing. Many participants used visualization customization to test their predictions of model behaviors. For example, through inspecting the input layer in the *Overview*, B4 learned that the input layer comprised multiple different image channels (e.g., red, green, and blue). He changed the input image to a red bell pepper from Tiny Imagenet (shown on the right) and expected to see high values in the input red channel: "*If I click the red image, I would see...*" After the updated visualization showed what he predicted, he said "*Right, it makes sense.*" We found the *Hyperparameter Widget* also allowed participants to test their hypotheses. While reading the description of convolution hyperparameters in the tutorial article, K3 noted "*Wait, then sometimes they won't work*". He then modified the hyperparatmeters in the *Hyperparameter Widget* and noticed some combinations indeed did not yield a valid operation output: "*It won't be able to slide, because the stride and kernel size don't fit the matrix*".

Customization facilitates engagement. Participants were intrigued to modify the visualization, and their engagement sparked further interest in learning CNNs. In the study, B6 spent a large amount of time on testing the CNN's behavior on edge cases by finding "difficult" images online. He searched with keywords "koala", "koala in a car", "bell pepper pizza", and eventually found a bell pepper pizza photo (shown on the right[3]). Our CNN model predicted the image as bell pepper with a probability of 0.71 and ladybug with a probability of 0.2. He commented, "*The model is not robust [...] oh, the ladybug ['s high softmax score] might come from the red dot.*" Another participant B5 uploaded his own photo as a new input image for the CNN model. After seeing his picture being classified as espresso, B5 started to use our tool to explore the reason of such classification by tracking back activation maps. He also asked how do experts interpret CNNs and said he would be interested in learning more about deep learning interpretability. This observation reflects previous findings that customizable visualization makes learning more engaging [13, 42].

8.3.4 Limitations

While we found CNN EXPLAINER provided participants with an engaging and enjoyable learning experience and helped them to more easily learn about CNNs, we also noticed some potential improvements to our current system design from this study.

Beginners need more guidance. We found that participants with less knowledge of CNNs needed more instructions to begin using CNN EXPLAINER. Some participants reported that the visual representation of the CNN and animation initially were not easy to understand, but the tutorial article and text annotations greatly helped them to interpret the visualizations. B8 skimmed through the tutorial article before interacting with the main visualization. She said, "*After going through the article, I think I will be able to use the tool better [...] I think the article is good, for beginner users especially.*" B2 appreciated the ability to jump to a certain section in the article by clicking the layer name in the visualization, and he suggested us to "*include a step-by-step tutorial for first time users [...] There was too much information, and I didn't know where to click at the beginning*". Therefore, we believe adding more text annotation and having a step-by-step tutorial mode could help beginners better understand the relations between CNN operations and their visual representations.

Limited explanation of *why* CNN works. Some participants, especially those less experienced with CNNs, were interested in learning *why* the CNN architecture works in addition to learning *how* a CNN model makes predictions. For example, B7 asked "*Why do we need ReLU?*" when he was learning the formula of the ReLU function. B5 understood what a Max Pooling layer's operation does but was unclear why it contributes to CNN's performance: "*It is counter-intuitive that Max Pooling reduces the [representation] size but makes the model better.*" Similarly, B6 commented on the Max Pooling layer: "*Why not take the minimum value? [...] I know how to compute them [layers], but I don't know why we compute them.*" Even though it is still an open question why CNNs work so well for various applications [17, 59], there are some commonly accepted "intuitions" of how different layers help this model class succeed. We briefly explain them in the tutorial article: for example, ReLU function is used to introduce non-linearty in the model. However, we believe it is worth designing visualizations that help users to learn about these concepts. For example, allowing users to change the ReLU activation function to a linear function, and then visualizing the new model predictions may help users gain understanding of *why* non-linear activation functions are needed in CNNs.

[3]Photo by Jennifer Laughlin, used with permission.

9 DISCUSSION AND FUTURE WORK

Explaining training process and backpropagation. CNN EXPLAINER helps users to learn how a pre-trained CNN model transforms the input image data into a class prediction. As we identified from two preliminary studies and an observational study, students are also interested in learning about the training process and backpropagation of CNNs. We plan to work with instructors and students to design and develop new visualizations to help beginners gain understanding of the training process and backpropagation in detail.

Generalizing to other layer types and neural network models. Our observational study demonstrated that CNN EXPLAINER helps users more easily understand low-level layer operations, high-level model structure, and their connections. We can adapt the *Interactive Formula Views* to explain other layer types (e.g., Leaky ReLU [38]) or a combination of layers (e.g. Residual Block [21]). Similarly, the transition between different levels of abstraction can be generalized to other neural networks, such as long short-term memory networks [22] and Transformer models [55] that require learners to understand the intricate layer operations in the context of a complex network structure.

Integrating algorithm visualization best practices. Existing work has studied how to design effective visualizations to help students learn algorithms. CNN EXPLAINER applies two key design principles from AV—visualizations with explanations and customizable visualizations (**G4**). However, there are many other AV design practices that future researchers can integrate in educational deep learning tools, such as giving interactive "pop quizzes" during the visualization process [41] and encouraging users to build their own visualizations [52].

Quantitative evaluation of educational effectiveness. We conducted a qualitative observational study to evaluate the usefulness and usability of CNN EXPLAINER. Further quantitative user studies would help us investigate how visualization tools help users gain understanding of deep learning concepts. We will draw inspiration from recent research [11, 28] to assess users' engagement level and content understanding through analysis of interaction logs.

10 CONCLUSION

As deep learning is increasingly used throughout our everyday life, it is important to help learners take the first step toward understanding this promising yet complex technology. In this work, we present CNN EXPLAINER, an interactive visualization system designed for non-experts to more easily learn about CNNs. Our tool runs in modern web browsers and is open-sourced, broadening the public's education access to modern AI techniques. We discussed design lessons learned from our iterative design process and an observational user study. We hope our work will inspire further research and development of visualization tools that help democratize and lower the barrier to understanding and appropriately applying AI technologies.

ACKNOWLEDGMENTS

We thank Anmol Chhabria, Kaan Sancak, Kantwon Rogers, and the Georgia Tech Visualization Lab for their support and constructive feedback. This work was supported in part by NSF grants IIS-1563816, CNS-1704701, NASA NSTRF, DARPA GARD, gifts from Intel, NVIDIA, Google, Amazon. Use, duplication, or disclosure is subject to the restrictions as stated in Agreement number HR00112030001 between the Government and the Performer.

REFERENCES

[1] Tiny ImageNet Visual Recognition Challenge. https://tiny-imagenet.herokuapp.com, 2015.

[2] Backpropagation Algorithm. https://developers-dot-devsite-v2-prod.appspot.com/machine-learning/crash-course/backprop-scroll, 2018.

[3] TensorSpace.js: Neural Network 3D Visualization Framework. https://tensorspace.org, 2018.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283. Savannah, GA, USA, Nov. 2016.

[5] A. Bilal, A. Jourabloo, M. Ye, X. Liu, and L. Ren. Do Convolutional Neural Networks Learn Class Hierarchy? *IEEE Transactions on Visualization and Computer Graphics*, 24(1):152–162, Jan. 2018.

[6] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011.

[7] M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, USA, 1988.

[8] M. D. Byrne, R. Catrambone, and J. T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33(4):253–278, Dec. 1999.

[9] M. Carney, B. Webster, I. Alvarado, K. Phillips, N. Howell, J. Griffith, J. Jongejan, A. Pitaru, and A. Chen. Teachable Machine: Approachable Web-Based Tool for Exploring Machine Learning Classification. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20. ACM, Honolulu, HI, USA, 2020.

[10] S. Carter and M. Nielsen. Using Artificial Intelligence to Augment Human Intelligence. *Distill*, 2(12), Dec. 2017.

[11] M. Conlen, A. Kale, and J. Heer. Capture & Analysis of Active Reading Behaviors for Interactive Articles on the Web. *Computer Graphics Forum*, 38(3):687–698, June 2019.

[12] N. Das, H. Park, Z. J. Wang, F. Hohman, R. Firstman, E. Rogers, and D. H. Chau. Massif: Interactive interpretation of adversarial attacks on deep learning. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20. ACM, Honolulu, HI, USA, 2020.

[13] E. Fouh, M. Akbar, and C. A. Shaffer. The Role of Visualization in Computer Science Education. *Computers in the Schools*, 29(1-2):95–117, Jan. 2012.

[14] D. Galles. Data structure visualizations, 2006.

[15] R. Garcia, A. C. Telea, B. Castro da Silva, J. Tørresen, and J. L. Dihl Comba. A task-and-technique centered survey on visual analytics for deep learning model engineering. *Computers & Graphics*, 77:30–49, Dec. 2018.

[16] S. Grissom, M. F. McNally, and T. Naps. Algorithm visualization in CS education: Comparing levels of student engagement. In *Proceedings of the 2003 ACM Symposium on Software Visualization - SoftVis '03*, pp. 87–94. San Diego, CA, USA, 2003.

[17] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, May 2018.

[18] P. J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*, pp. 579–584. ACM Press, Denver, CO, USA, 2013.

[19] S. Hansen, N. Narayanan, and M. Hegarty. Designing Educationally Effective Algorithm Visualizations. *Journal of Visual Languages & Computing*, 13(3):291–317, June 2002.

[20] A. W. Harley. An Interactive Node-Link Visualization of Convolutional Neural Networks. In *Advances in Visual Computing*, vol. 9474, pp. 867–877. Springer International Publishing, 2015.

[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, Dec. 2015.

[22] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997.

[23] F. Hohman, M. Kahng, R. Pienta, and D. H. Chau. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE Transactions on Visualization and Computer Graphics*, 25(8):2674–2693, Aug. 2019.

[24] F. Hohman, H. Park, C. Robinson, and D. H. Chau. Summit: Scaling Deep Learning Interpretability by Visualizing Activation and Attribution Summarizations. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1096–1106, Jan. 2020.

[25] C. Hundhausen and S. Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pp. 21–28. IEEE Comput. Soc, Seattle, WA, USA, 2000.

[26] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.

[27] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. Chau. ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):88–97, Jan. 2018.

[28] M. Kahng and D. H. Chau. How Does Visualization Help People Learn Deep Learning? Evaluation of GAN Lab. In *IEEE VIS 2019 Workshop on EValuation of Interactive VisuAl Machine Learning Systems*, Oct. 2019.

[29] M. Kahng, N. Thorat, D. H. Chau, F. B. Viegas, and M. Wattenberg. GAN Lab: Understanding Complex Deep Generative Models using Interactive Visual Experimentation. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):310–320, Jan. 2019.

[30] A. Karpathy. ConvNetJS MNIST demo, 2016.

[31] A. Karpathy. CS231n Convolutional Neural Networks for Visual Recognition, 2016.

[32] C. Kehoe, J. Stasko, and A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies*, 54(2):265–284, Feb. 2001.

[33] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[34] M. Liu, S. Liu, H. Su, K. Cao, and J. Zhu. Analyzing the Noise Robustness of Deep Neural Networks. In *2018 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pp. 60–71. IEEE, Berlin, Germany, Oct. 2018.

[35] M. Liu, J. Shi, K. Cao, J. Zhu, and S. Liu. Analyzing the Training Processes of Deep Generative Models. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):77–87, Jan. 2018.

[36] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu. Towards Better Analysis of Deep Convolutional Neural Networks. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):91–100, Jan. 2017.

[37] S. Liu, D. Maljovec, B. Wang, P.-T. Bremer, and V. Pascucci. Visualizing High-Dimensional Data: Advances in the Past Decade. *IEEE Transactions on Visualization and Computer Graphics*, 23(3):1249–1268, Mar. 2017.

[38] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[39] A. Madsen. Visualizing memorization in RNNs. *Distill*, 4(3):10.23915/distill.00016, Mar. 2019.

[40] R. E. Mayer and R. B. Anderson. Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology*, 83(4):484–490, 1991.

[41] T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ—an environment to actively engage students in Web-based algorithm visualizations. *ACM SIGCSE Bulletin*, 32(1):109–113, Mar. 2000.

[42] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bull.*, 35(2):131–152, June 2002.

[43] A. P. Norton and Y. Qi. Adversarial-Playground: A visualization suite showing how adversarial examples fool deep learning. In *2017 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–4. IEEE, Phoenix, AZ, USA, Oct. 2017.

[44] C. Olah. Neural Networks, Manifolds, and Topology, June 2014.

[45] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035. 2019.

[46] N. Pezzotti, T. Hollt, J. Van Gemert, B. P. Lelieveldt, E. Eisemann, and A. Vilanova. DeepEyes: Progressive Visual Analytics for Designing Deep Neural Networks. *IEEE Transactions on Visualization and Computer*

Graphics, 24(1):98–108, Jan. 2018.

[47] D. Schweitzer and W. Brown. Interactive visualization for the active learning classroom. *ACM SIGCSE Bulletin*, 39(1):208, Mar. 2007.

[48] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards. Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education*, 10(3):1–22, Aug. 2010.

[49] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*, Apr. 2015.

[50] D. Smilkov, S. Carter, D. Sculley, F. B. Viégas, and M. Wattenberg. Direct-Manipulation Visualization of Deep Networks. *arXiv:1708.03788*, Aug. 2017.

[51] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viégas, and M. Wattenberg. TensorFlow.js: Machine Learning for the Web and Beyond. *arXiv:1901.05350 [cs]*, Feb. 2019.

[52] J. T. Stasko. Using student-built algorithm animations as learning aids. *ACM SIGCSE Bulletin*, 29(1):25–29, Mar. 1997.

[53] E. Stevens, L. Antiga, and T. Viehmann. *Deep Learning with PyTorch*. O'Reilly Media, 2019.

[54] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):667–676, Jan. 2018.

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 6000–6010, 2017.

[56] J. Wang, L. Gou, H. Yang, and H.-W. Shen. GANViz: A Visual Analytics Approach to Understand the Adversarial Game. *IEEE Transactions on Visualization and Computer Graphics*, 24(6):1905–1917, June 2018.

[57] Z. J. Wang, R. Turko, O. Shaikh, H. Park, N. Das, F. Hohman, M. Kahng, and D. H. Chau. CNN 101: Interactive visual learning for convolutional neural networks. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20. ACM, Honolulu, HI, USA, 2020.

[58] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. Understanding Neural Networks Through Deep Visualization. In *ICML Deep Learning Workshop*, 2015.

[59] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations (ICLR), Toulon, France, Conference Track Proceedings*, 2017.

Нейронні мережі

Лекція_06

********** **********

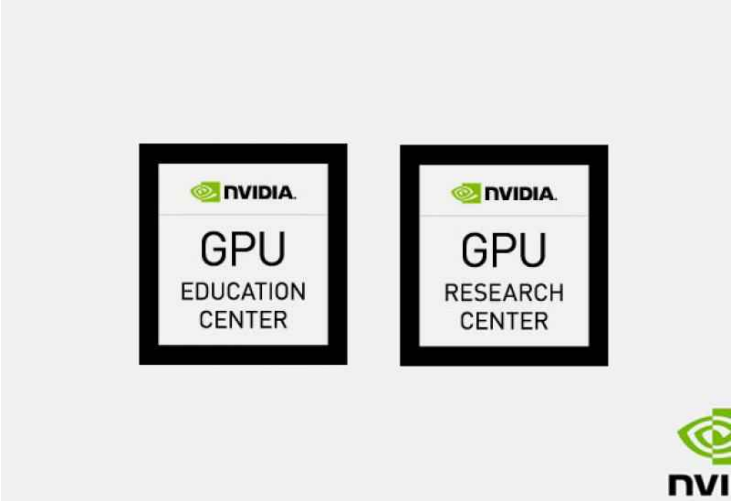
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 06 - Нейронні мережі - Оцінювачі

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО А
Введення в оцінювачі TF
https://drive.google.com/file/d/10C-ypmitQmGkPQt-0oStL3OJGgSSvvvv/view?usp=sharing

ДЕМО В
Створення моделі DNN за допомогою TF Estimators
https://drive.google.com/file/d/1fro49geaFUoQJ4frgLJy04FRuzNeNA7T/view?usp=sharing

ДЕМО С
TF Datasets Benchmark від TF Estimators
https://drive.google.com/file/d/1dALe-tX9pyMjNKXbBikM9L6sEv34ulI4/view?usp=sharing

ДЕМО Д
Передача навчання -Rock Paper Scissors (з використанням NASNetMobile)
https://drive.google.com/file/d/1XvXxDE5OSArPgwZ_bcn3ACfFRWu5rBuV/view?usp=sharing

DEMO A - Introduction to TF Estimators

based on (C) Velayudham, Sakranha, TF Authors works

```
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Connect to Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
! cp /content/drive/MyDrive/2022_COLAB_NN/Lecture_06_TF2_Estimators_CNN_RockPaperS
! ls
```

```
drive  sample_data  winequality-white.csv
```

Loading Data

```
data_url='winequality-white.csv'
data=pd.read_csv(data_url,delimiter=';')
data.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956

Selecting Features/Labels

```
x = data.iloc[:,:-1]
y = data.iloc[:,-1]
```

```
sc = StandardScaler()
x = sc.fit_transform(x)
```

Creating datasets

```
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.3,random_state
```

```
input_shape = xtrain.shape[1]
```

Defining simple FCN Keras model

```
small_model = tf.keras.models.Sequential([
              tf.keras.layers.Dense(64,activation = 'relu',
                                    input_shape = (input_shape,)),
              tf.keras.layers.Dense(1)
])

small_model.compile(loss = 'mse', optimizer = 'adam')
```

```
def input_fn(features, labels, training = True, batch_size = 32):
  #converts inputs to a dataset
  dataset = tf.data.Dataset.from_tensor_slices(({'dense_input':features},labels))

  #shuffle and repeat in a training mode
  if training:
    dataset = dataset.shuffle(1000).repeat()

  #giving inputs in batch for training
  return dataset.batch(batch_size)
```

Convert Keras model to Estimator

```
keras_small_estimator = tf.keras.estimator.model_to_estimator(
    keras_model = small_model, model_dir = 'keras_small_classifier')
```

```
/usr/local/lib/python3.7/dist-packages/keras/backend.py:450: UserWarning: `tf
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
```

Train

```
keras_small_estimator.train(input_fn = lambda:input_fn(xtrain, ytrain), steps = 20
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/pyt
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically both
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
<tensorflow_estimator.python.estimator.estimator.EstimatorV2 at
0x7f96901753d0>
```

Evaluate

```
eval_small_result = keras_small_estimator.evaluate(
    input_fn = lambda:input_fn(xtest, ytest, training = False), steps=1000)
print('Eval result: {}'.format(eval_small_result))
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2057: User
  updates = self.state_updates
Eval result: {'loss': 0.6073161, 'global_step': 2000}
```

Analyze history and metrics

```
%load_ext tensorboard
%tensorboard --logdir ./keras_small_classifier
```

TensorBoard SCALARS GRAP INACTIVE

Show data download links

Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing

0.6

Horizontal Axis

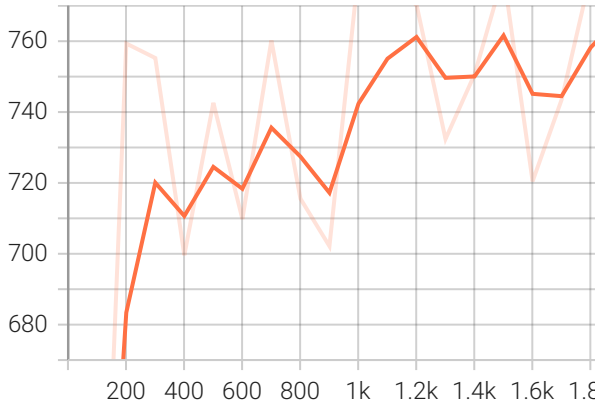
STEP RELATIVE

WALL

Filter tags (regular expressions supported)

global_step

global_step/sec
tag: global_step/sec



Write a regex to filter runs

.

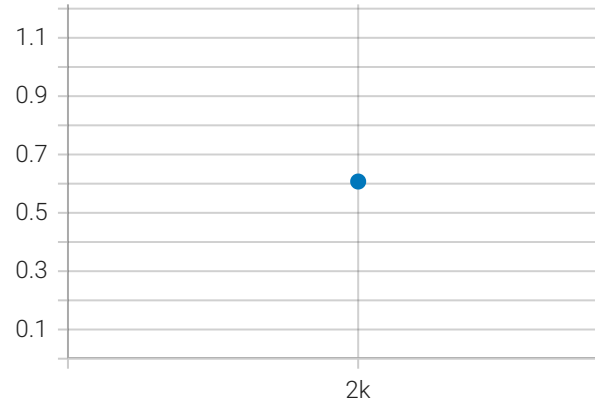
eval

TOGGLE ALL RUNS

./keras_small_classifier

loss

loss
tag: loss



Colab paid products - Cancel contracts here

0s completed at 8:28 PM

DEMO B - Create DNN Model by TF Estimators

based on (C) Velayudham, Sakranha, TF Authors works

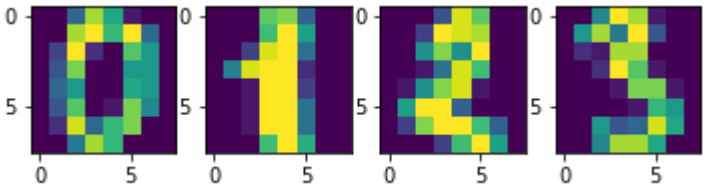
```
import tensorflow as tf
```

Loading Data

```
from sklearn import datasets
digits = datasets.load_digits()
```

```
#plotting sample image
import matplotlib.pyplot as plt
plt.figure(figsize=(1,1))
fig, ax = plt.subplots(1,4)
ax[0].imshow(digits.images[0])
ax[1].imshow(digits.images[1])
ax[2].imshow(digits.images[2])
ax[3].imshow(digits.images[3])
plt.show()
```

```
<Figure size 72x72 with 0 Axes>
```



Preprocessing Data

```
# reshape the data to two dimensions
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
data.shape
```

```
(1797, 64)
```

```
# split into training/testing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.05, shuffle=False)
```

Defining Input Function

```
# create column names for our model input function
columns = ['p_'+ str(i) for i in range(1,65)]
```

```
feature_columns = []
for col in columns:
  feature_columns.append(tf.feature_column.numeric_column(key = col))
```

```
def input_fn(features, labels, training = True, batch_size = 32):
  #converts inputs to a dataset
  dataset = tf.data.Dataset.from_tensor_slices((dict(features),labels))
  #shuffle and repeat in a training mode
  if training:
    dataset=dataset.shuffle(1000).repeat()

  #giving inputs in batches for training
  return dataset.batch(batch_size)
```

Create DNNClassifier Estimator instance

```
classifier = tf.estimator.DNNClassifier(hidden_units = [256, 128, 64],
                                        feature_columns = feature_columns,
                                        optimizer = 'Adagrad',
                                        n_classes = 10,
                                        model_dir = 'classifier')
```

Adding extra hidden layer

without Dropout

```
classifier = tf.estimator.DNNClassifier(hidden_units = [256, 128, 64, 32],
                                        feature_columns = feature_columns,
                                        optimizer = 'Adagrad',
                                        n_classes = 10,
                                        model_dir = 'classifier')
```

with Dropout

```

```

```
'''
classifier = tf.estimator.DNNClassifier(hidden_units = [256, 128, 64, 32],
                                        feature_columns = feature_columns,
                                        optimizer = 'Adagrad',
                                        n_classes = 10,
                                        dropout = 0.2,
                                        model_dir = 'classifier')
'''
```

```
'\nclassifier = tf.estimator.DNNClassifier(hidden_units = [256, 12
8, 64, 32],\n                                                feature_column
s = feature_columns,\n                                                  optim
izer = 'Adagrad',\n                                                  n_classe
```

Model Training

```
# create dataframes for training
import pandas as pd
dftrain = pd.DataFrame(X_train, columns = columns)


classifier.train(input_fn = lambda:input_fn(dftrain,
                                            y_train,
                                            training = True),
                                            steps = 2000)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/pyt
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically both
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/keras/optimize
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to th
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
WARNING:tensorflow:It seems that global step (tf.train.get_global_step) has r
<tensorflow_estimator.python.estimator.canned.dnn.DNNClassifierV2 at
0x7f988f1726d0>
```

Model Evaluation

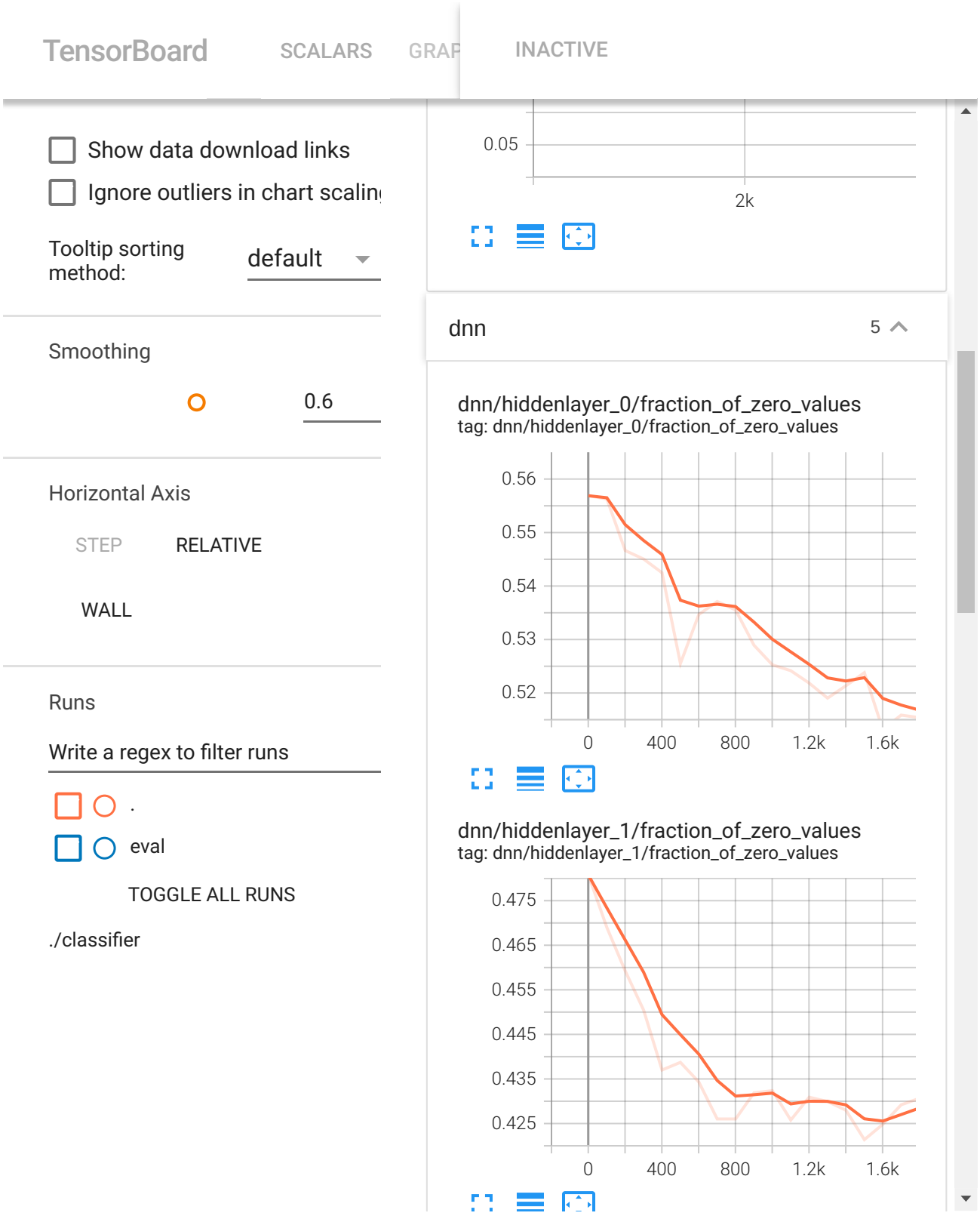
```
# create dataframe for evaluation
dftest = pd.DataFrame(X_test, columns = columns)

eval_result = classifier.evaluate(
    input_fn = lambda:input_fn(dftest, y_test, training = False)
)

eval_result
```

```
{'accuracy': 0.95555556,
 'average_loss': 0.2061766,
```

```
'loss': 0.19756849,
'global_step': 2000}
```

Resume at Tensorboard

```
%load_ext tensorboard
%tensorboard --logdir ./classifier
```



Predicting unseen data

```
# An input function for prediction
def pred_input_fn(features, batch_size = 32):
    # Convert the inputs to a Dataset without labels.
    return tf.data.Dataset.from_tensor_slices(dict(features)).batch(batch_size)

test = dftest.iloc[:2,:] #1st two data points for predictions

expected = y_test[:2].tolist() #expected labels

pred = list(classifier.predict(
    input_fn =  lambda:pred_input_fn(test))
)

for pred_dict, expec in zip(pred, expected):
    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]
    print('predicted class {} , probability of prediction {} , expected label {}'.
```

```
predicted class 8 , probability of prediction 0.9750990867614746 , expected l
predicted class 4 , probability of prediction 0.95527184009552 , expected lab
```

DEMO C - TF Datasets Benchmark by TF Estimators

based on (C) Velayudham, Sakranha, TF Authors works

```
# To avoid the compatibility issue with Tensorflow, cuda and models repo code.
# Try installing the below TensorFlow version and cuda version at the start of col
!pip install tensorflow==2.8
!apt install --allow-change-held-packages libcudnn8=8.1.0.77-1+cuda11.2
```

```
Requirement already satisfied: flatbuffers>=1.12 in /usr/local/lib/python3.
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.7/dist
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: absl-py>=0.4.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python
Requirement already satisfied: libclang>=9.0.1 in /usr/local/lib/python3.7/
Requirement already satisfied: keras-preprocessing>=1.1.1 in /usr/local/lib
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.7
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/pyth
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /us
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/loc
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/pyt
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/di
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/pyth
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/p
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/py
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-p

Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/pytho
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: charset-normalizer<3,>=2 in /usr/local/lib/p
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/pyth
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/
Installing collected packages: tf-estimator-nightly, tensorboard, keras, te
  Attempting uninstall: tensorboard
    Found existing installation: tensorboard 2.10.1
    Uninstalling tensorboard-2.10.1:
      Successfully uninstalled tensorboard-2.10.1
  Attempting uninstall: keras
    Found existing installation: keras 2.10.0
    Uninstalling keras-2.10.0:
      Successfully uninstalled keras-2.10.0
  Attempting uninstall: tensorflow
    Found existing installation: tensorflow 2.10.0
    Uninstalling tensorflow-2.10.0:
      Successfully uninstalled tensorflow-2.10.0
```

```
ERROR: pip's dependency resolver does not currently take into account all t
tfx-bsl 1.10.1 requires tensorflow!=2.0.*,!=2.1.*,!=2.2.*,!=2.3.*,!=2.4.*,!
tensorflow-serving-api 2.10.0 requires tensorflow<3,>=2.10.0, but you have
tensorflow-data-validation 1.10.0 requires tensorflow!=2.0.*,!=2.1.*,!=2.2.
Successfully installed keras-2.8.0 tensorboard-2.8.0 tensorflow-2.8.0+zzzco
Reading package lists... Done
Building dependency tree
Reading state information... Done
libcudnn8 is already the newest version (8.1.0.77-1+cuda11.2).
The following package was automatically installed and is no longer required
  libnvidia-common-460
```

```
import tensorflow as tf
import tensorflow_datasets as tfds
```

TensorFlow Datasets

```
#To get the list of available
tfds.list_builders()
```

```
 'huggingface:psc',
 'huggingface:ptb_text_only',
 'huggingface:pubmed',
 'huggingface:pubmed_qa',
 'huggingface:py_ast',
 'huggingface:qa4mre',
 'huggingface:qa_srl',
 'huggingface:qa_zre',
 'huggingface:qangaroo',
 'huggingface:qanta',
 'huggingface:qasc',
 'huggingface:qasper',
 'huggingface:qed',
 'huggingface:qed_amara',
 'huggingface:quac',
 'huggingface:quail',
 'huggingface:quarel',
 'huggingface:quartz',
 'huggingface:quickdraw',
 'huggingface:quora',
 'huggingface:quoref',
 'huggingface:race',
 'huggingface:re_dial',
 'huggingface:reasoning_bg',
 'huggingface:recipe_nlg',
 'huggingface:reclor',
 'huggingface:red_caps',
 'huggingface:reddit',
 'huggingface:reddit_tifu',
 'huggingface:refresd',
 'huggingface:reuters21578',
 'huggingface:riddle_sense',
 'huggingface:ro_sent',
 'huggingface:ro_sts',
 'huggingface:ro_sts_parallel',
 'huggingface:roman_urdu',
```

```
 'huggingface:roman_urdu_hate_speech',
 'huggingface:ronec',
 'huggingface:ropes',
 'huggingface:rotten_tomatoes',
 'huggingface:russian_super_glue',
 'huggingface:rvl_cdip',
 'huggingface:s2orc',
 'huggingface:samsum',
 'huggingface:sanskrit_classic',
 'huggingface:saudinewsnet',
 'huggingface:sberquad',
 'huggingface:sbu_captions',
 'huggingface:scan',
 'huggingface:scb_mt_enth_2020',
 'huggingface:scene_parse_150',
 'huggingface:schema_guided_dstc8',
 'huggingface:scicite',
 'huggingface:scielo',
 'huggingface:scientific_papers',
 'huggingface:scifact',
 'huggingface:sciq',
 'huggingface:scitail',
```

```
len(tfds.list_builders())
```

```
1122
```

```
#ds, ds_info = tfds.load('cifar10', split='train', with_info=True)
ds, ds_info = tfds.load('fashion_mnist', split='train', with_info=True)
fig = tfds.show_examples(ds, ds_info)
```

Benchmark your datasets

Note: This API is new and only available via

! pip install tfds-nightly

```
#! pip install tfds-nightly
```

```
! nvidia-smi
```

```
Mon Oct  3 18:41:21 2022
+-----------------------------------------------------------------------------
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2
|-------------------------------+----------------------+----------------------
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M.
|                               |                      |               MIG M.
|===============================+======================+======================
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0
| N/A   61C    P0    30W /  70W |    286MiB / 15109MiB |      0%      Default
|                               |                      |                  N/A
+-------------------------------+----------------------+----------------------

+-----------------------------------------------------------------------------
| Processes:
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory
|        ID   ID                                                   Usage
|=============================================================================
+-----------------------------------------------------------------------------
```

```
# Benchmark your datasets - GPU
ds = ds.batch(32).prefetch(1)

tfds.benchmark(ds, batch_size=32)
tfds.benchmark(ds, batch_size=32)  # Second epoch much faster due to auto-caching
```

```
************ Summary ************
```

```
100%                                                1875/1875 [00:04<00:00, 257.26it/s]
```

```
Examples/sec (First included) 12025.68 ex/sec (total: 60032 ex, 4.9
Examples/sec (First only) 364.56 ex/sec (total: 32 ex, 0.09 sec)
Examples/sec (First excluded) 12234.40 ex/sec (total: 60000 ex, 4.9

************ Summary ************
```

```
100%                                                1875/1875 [00:00<00:00, 2256.08it/s]
```

```
# Benchmark your datasets - TPU
ds = ds.batch(32).prefetch(1)

tfds.benchmark(ds, batch_size=32)
tfds.benchmark(ds, batch_size=32)  # Second epoch much faster due to auto-caching
```

```
************ Summary ************
```

```
100%                                                59/59 [00:00<00:00, 125.63it/s]
```

```
Examples/sec (First included) 3414.61 ex/sec (total: 1920 ex, 0.56
Examples/sec (First only) 249.06 ex/sec (total: 32 ex, 0.13 sec)
Examples/sec (First excluded) 4352.14 ex/sec (total: 1888 ex, 0.43

************ Summary ************
```

```
100%                                                59/59 [00:00<00:00, 174.71it/s]
```

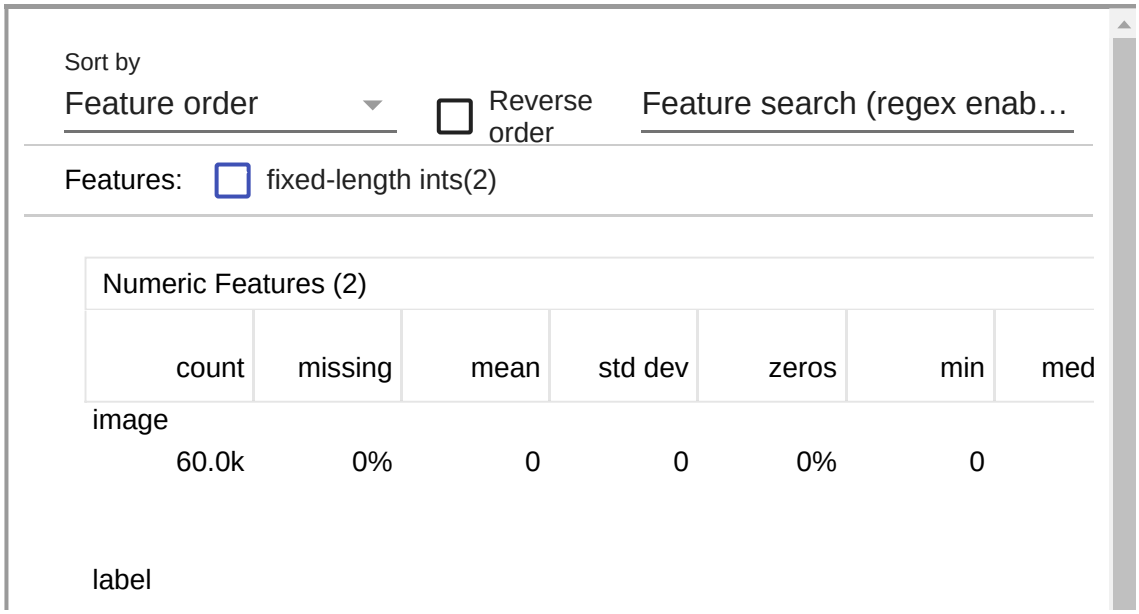
```
Examples/sec (First included) 4876.42 ex/sec (total: 1920 ex, 0.39
Examples/sec (First only) 269.12 ex/sec (total: 32 ex, 0.12 sec)
Examples/sec (First excluded) 6869.81 ex/sec (total: 1888 ex, 0.27
```

BenchmarkResult:

	duration	num_examples	avg
first+lasts	0.393731	1920	4876.422340
first	0.118906	32	269.121261

```
#! pip install tensorflow_data_validation
```

```
# Display the datasets statistics on a Colab/Jupyter notebook
tfds.show_statistics(ds_info)
```



Import VGG16 module

```
keras_Vgg16 = tf.keras.applications.VGG16(
    input_shape=(160, 160, 3), include_top=False)
keras_Vgg16.trainable = False
```

Create Keras model by adding layers to VGG16 model

```
estimator_model = tf.keras.Sequential([
    keras_Vgg16,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    tf.keras.layers.Dense(1)
])
```

```
keras_Vgg16.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 160, 160, 3)]     0

 block1_conv1 (Conv2D)       (None, 160, 160, 64)      1792

 block1_conv2 (Conv2D)       (None, 160, 160, 64)      36928

 block1_pool (MaxPooling2D)  (None, 80, 80, 64)        0

 block2_conv1 (Conv2D)       (None, 80, 80, 128)       73856

 block2_conv2 (Conv2D)       (None, 80, 80, 128)       147584
```

```
 block2_pool (MaxPooling2D)  (None, 40, 40, 128)       0

 block3_conv1 (Conv2D)       (None, 40, 40, 256)       295168

 block3_conv2 (Conv2D)       (None, 40, 40, 256)       590080

 block3_conv3 (Conv2D)       (None, 40, 40, 256)       590080

 block3_pool (MaxPooling2D)  (None, 20, 20, 256)       0

 block4_conv1 (Conv2D)       (None, 20, 20, 512)       1180160

 block4_conv2 (Conv2D)       (None, 20, 20, 512)       2359808

 block4_conv3 (Conv2D)       (None, 20, 20, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 10, 10, 512)       0

 block5_conv1 (Conv2D)       (None, 10, 10, 512)       2359808

 block5_conv2 (Conv2D)       (None, 10, 10, 512)       2359808

 block5_conv3 (Conv2D)       (None, 10, 10, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 5, 5, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

```
estimator_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 5, 5, 512)         14714688

 global_average_pooling2d (G  (None, 512)              0
 lobalAveragePooling2D)

 dense (Dense)               (None, 256)               131328

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 14,846,273
Trainable params: 131,585
Non-trainable params: 14,714,688
_________________________________________________________________
```

Compile

```
```

```
# Compile the model
estimator_model.compile(
    optimizer = 'adam',
    loss=tf.keras.losses.BinaryCrossentropy(from_logits = True),
    metrics = ['accuracy'])
```

Create Estimator

```
est_vgg16 = tf.keras.estimator.model_to_estimator(keras_model = estimator_model,
                                                  model_dir = 'classifier')
```

Data preprocessing

```
IMG_SIZE = 160
import tensorflow_datasets as tfds
def preprocess(image, label):
  image = tf.cast(image, tf.float32)
  #image = (image/127.5) - 1
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  return image, label
```

Input function

```
def train_input_fn(batch_size):
  data = tfds.load('cats_vs_dogs', as_supervised=True)
  train_data = data['train']
  train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
  return train_data
```

Training

```
#%%time
est_vgg16.train(input_fn = lambda: train_input_fn(32), steps = 500)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/pyt
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically both
/usr/local/lib/python3.7/dist-packages/keras/backend.py:450: UserWarning: `tf
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/pyt
Instructions for updating:
Use standard file utilities to get mtimes.
<tensorflow_estimator.python.estimator.estimator.EstimatorV2 at
0x7fba65713650>
```

Evaluation

```
est_vgg16.evaluate(input_fn = lambda: train_input_fn(32), steps=10)
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2057: User
  updates = self.state_updates
{'accuracy': 0.934375, 'loss': 0.3812843, 'global_step': 500}
```

Monitoring

```
%load_ext tensorboard
%tensorboard --logdir ./classifier
```

TensorBoard SCALARS INACTIVE

Show data download links

Ignore outliers in chart scalin

Tooltip sorting default

Smoothing 0.6

Horizontal Axis

STEP RELATIVE

WALL

Runs

Write a regex to filter runs

.

eval

TOGGLE ALL RUNS

./classifier

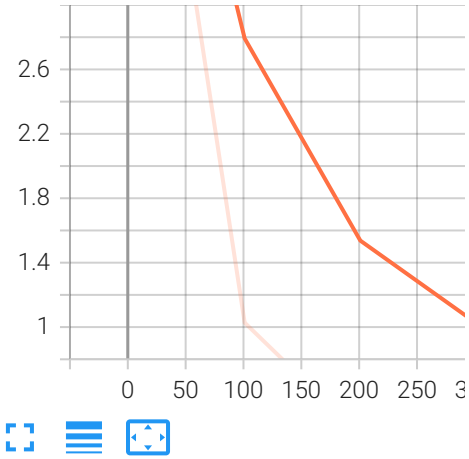
Filter tags (regular expressions su...

accuracy

loss

loss_1

loss_1
tag: loss_1



✓ 0s completed at 9:42 PM

DEMO D - Transfer Learning - Rock Paper Scissors (using NASNetMobile)

based on (C) Ng, Moroney, Mingxing Tan, Quoc V. Le, Rawlani, and **Oleksii Trekhleb** ("Our Man in Uber" :))

Experiment overview

In this experiment we will build a Convolutional Neural Network (CNN) model using Tensorflow to recognize Rock-Paper-Scissors signs (gestures) on the photo.

Instead of training the model from scratch we will use **transfer learning** method (look at DEMOs in previous lectures) a family of TF2-Keras-Applications models. Here we will actually use NASNetMobile and other models which are pre-trained on the ImageNet dataset, a large dataset of 1.4M images and 1000 classes of web images.

Importing dependencies

```
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np
import platform
import datetime
import os
import math
import random

print('Python version:', platform.python_version())
print('Tensorflow version:', tf.__version__)
print('Keras version:', tf.keras.__version__)
```

```
Python version: 3.7.14
Tensorflow version: 2.8.2
Keras version: 2.8.0
```

Configuring TensorBoard

We will use TensorBoard as a helper to debug the model training process.

```
# Load the TensorBoard notebook extension.
```

```
# %reload_ext tensorboard
%load_ext tensorboard

# Clear any logs from previous runs.
!rm -rf ./logs/
```

Dataset - Example

We will download Rock-Paper-Scissors dataset from TensorFlow Datasets collection. To do that we loaded a `tensorflow_datasets` module.

`tensorflow_datasets` defines a collection of datasets ready-to-use with TensorFlow.

Each dataset is defined as a tfds.core.DatasetBuilder, which encapsulates the logic to download the dataset and construct an input pipeline, as well as contains the dataset documentation (version, splits, number of examples, etc.).

```
# See available datasets
tfds.list_builders()
```

```
      'huggingface:quac',
      'huggingface:quail',
      'huggingface:quarel',
      'huggingface:quartz',
      'huggingface:quickdraw',
      'huggingface:quora',
      'huggingface:quoref',
      'huggingface:race',
      'huggingface:re_dial',
      'huggingface:reasoning_bg',
      'huggingface:recipe_nlg',
      'huggingface:reclor',
      'huggingface:red_caps',
      'huggingface:reddit',
      'huggingface:reddit_tifu',
      'huggingface:refresd',
      'huggingface:reuters21578',
      'huggingface:riddle_sense',
      'huggingface:ro_sent',
      'huggingface:ro_sts',
      'huggingface:ro_sts_parallel',
      'huggingface:roman_urdu',
      'huggingface:roman_urdu_hate_speech',
      'huggingface:ronec',
      'huggingface:ropes',
      'huggingface:rotten_tomatoes',
      'huggingface:russian_super_glue',
      'huggingface:rvl_cdip',
      'huggingface:s2orc',
      'huggingface:samsum',
      'huggingface:sanskrit_classic',
      'huggingface:saudinewsnet',
      'huggingface:sberquad',
      'huggingface:sbu_captions',
      'huggingface:scan',
      'huggingface:scb_mt_enth_2020',
```

```
'huggingface:scene_parse_150',
'huggingface:schema_guided_dstc8',
'huggingface:scicite',
'huggingface:scielo',
'huggingface:scientific_papers',
'huggingface:scifact',
'huggingface:sciq',
'huggingface:scitail',
'huggingface:scitldr',
'huggingface:search_qa',
'huggingface:sede',
'huggingface:selqa',
'huggingface:sem_eval_2010_task_8',
'huggingface:sem_eval_2014_task_1',
'huggingface:sem_eval_2018_task_1',
'huggingface:sem_eval_2020_task_11',
'huggingface:sent_comp',
'huggingface:senti_lex',
'huggingface:senti_ws',
'huggingface:sentiment140',
'huggingface:sepedi_ner',
'huggingface:sesotho_ner_corpus',
```

We will use the classic dataset by Moroney:

- Title: rock_paper_scissors
- Description: Images of hands playing rock, paper, scissor game.
- Homepage: http://laurencemoroney.com/rock-paper-scissors-dataset
- Source code: `tfds.image_classification.RockPaperScissors`
- Versions: 3.0.0 (default): New split API (https://tensorflow.org/datasets/splits)
- Download size: 219.53 MiB
- Image Examples:

Rock 

Paper 

Scissors 

Loading the dataset

```
DATASET_NAME = 'rock_paper_scissors'

(dataset_train_raw, dataset_test_raw), dataset_info = tfds.load(
    name=DATASET_NAME,
    data_dir='tmp',
    with_info=True,
    as_supervised=True,
    split=[tfds.Split.TRAIN, tfds.Split.TEST],
)
```

Downloading and preparing dataset 219.53 MiB (download: 219.53 MiB, generated

Dl Completed...: 100% 2/2 [00:05<00:00, 2.84s/ url]

Dl Size...: 100% 219/219 [00:05<00:00, 46.67 MiB/s]

Dataset rock paper scissors downloaded and prepared to tmp/rock paper scissor

```
print('Raw train dataset:', dataset_train_raw)
print('Raw train dataset size:', len(list(dataset_train_raw)), '\n')

print('Raw test dataset:', dataset_test_raw)
print('Raw test dataset size:', len(list(dataset_test_raw)), '\n')
```

```
Raw train dataset: <PrefetchDataset element_spec=(TensorSpec(shape=(300, 300,
Raw train dataset size: 2520

Raw test dataset: <PrefetchDataset element_spec=(TensorSpec(shape=(300, 300,
Raw test dataset size: 372
```

```
dataset_info
```

```
tfds.core.DatasetInfo(
    name='rock_paper_scissors',
    full_name='rock_paper_scissors/3.0.0',
    description="""
    Images of hands playing rock, paper, scissor game.
    """,
    homepage='http://laurencemoroney.com/rock-paper-scissors-dataset',
    data_path='tmp/rock_paper_scissors/3.0.0',
    file_format=tfrecord,
    download_size=219.53 MiB,
    dataset_size=219.23 MiB,
    features=FeaturesDict({
        'image': Image(shape=(300, 300, 3), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=3),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=372, num_shards=1>,
        'train': <SplitInfo num_examples=2520, num_shards=2>,
    },
    citation="""@ONLINE {rps,
    author = "Laurence Moroney",
    title = "Rock, Paper, Scissors Dataset",
    month = "feb",
    year = "2019",
    url = "http://laurencemoroney.com/rock-paper-scissors-dataset"
    }""",
)
```

```
NUM_TRAIN_EXAMPLES = dataset_info.splits['train'].num_examples
NUM_TEST_EXAMPLES = dataset_info.splits['test'].num_examples
NUM_CLASSES = dataset_info.features['label'].num_classes

print('Number of TRAIN examples:', NUM_TRAIN_EXAMPLES)
print('Number of TEST examples:', NUM_TEST_EXAMPLES)
print('Number of label classes:', NUM_CLASSES)
```

```
Number of TRAIN examples: 2520
Number of TEST examples: 372
```

```
Number of label classes: 3
```

```
INPUT_IMG_SIZE_ORIGINAL = dataset_info.features['image'].shape[0]
INPUT_IMG_SHAPE_ORIGINAL = dataset_info.features['image'].shape

# For some models only some sizes are possible, for example:
# for NASNetMobile - 224, ...
INPUT_IMG_SIZE_REDUCED = 224
INPUT_IMG_SHAPE_REDUCED = (
    INPUT_IMG_SIZE_REDUCED,
    INPUT_IMG_SIZE_REDUCED,
    INPUT_IMG_SHAPE_ORIGINAL[2]
)

# Here we may switch between bigger or smaller image sized that we will train our
INPUT_IMG_SIZE = INPUT_IMG_SIZE_REDUCED
INPUT_IMG_SHAPE = INPUT_IMG_SHAPE_REDUCED

print('Input image size (original):', INPUT_IMG_SIZE_ORIGINAL)
print('Input image shape (original):', INPUT_IMG_SHAPE_ORIGINAL)
print('\n')
print('Input image size (reduced):', INPUT_IMG_SIZE_REDUCED)
print('Input image shape (reduced):', INPUT_IMG_SHAPE_REDUCED)
print('\n')
print('Input image size:', INPUT_IMG_SIZE)
print('Input image shape:', INPUT_IMG_SHAPE)
```

```
Input image size (original): 300
Input image shape (original): (300, 300, 3)


Input image size (reduced): 224
Input image shape (reduced): (224, 224, 3)


Input image size: 224
Input image shape: (224, 224, 3)
```

```
# Function to convert label ID to labels string.
get_label_name = dataset_info.features['label'].int2str
```

```
print(get_label_name(0));
print(get_label_name(1));
print(get_label_name(2));
```

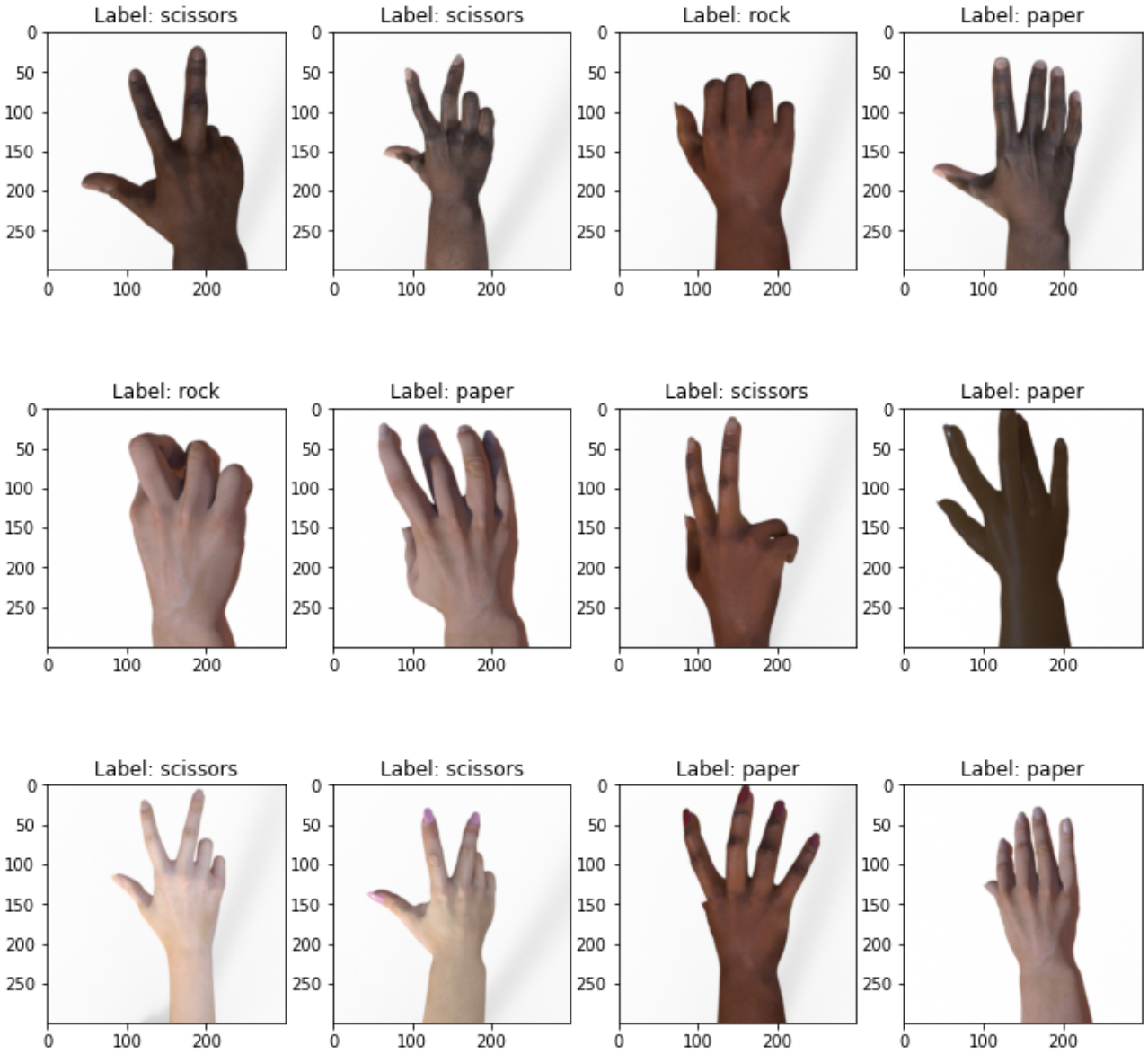
```
rock
paper
scissors
```

Exploring the dataset

```
```

```
def preview_dataset(dataset):
    plt.figure(figsize=(12, 12))
    plot_index = 0
    for features in dataset.take(12):
        (image, label) = features
        plot_index += 1
        plt.subplot(3, 4, plot_index)
        # plt.axis('Off')
        label = get_label_name(label.numpy())
        plt.title('Label: %s' % label)
        plt.imshow(image.numpy())
```

```
# Explore raw training dataset images.
preview_dataset(dataset_train_raw)
```




```
# Explore what values are used to represent the image.
(first_image, first_lable) = list(dataset_train_raw.take(1))[0]
print('Label:', first_lable.numpy(), '\n')
```

```
print('Image shape:', first_image.numpy().shape, '\n')
print(first_image.numpy())
```

```
Label: 2

Image shape: (300, 300, 3)

[[[254 254 254]
  [253 253 253]
  [254 254 254]
  ...
  [251 251 251]
  [250 250 250]
  [250 250 250]]

 [[254 254 254]
  [254 254 254]
  [253 253 253]
  ...
  [250 250 250]
  [251 251 251]
  [249 249 249]]

 [[254 254 254]
  [254 254 254]
  [254 254 254]
  ...
  [251 251 251]
  [250 250 250]
  [252 252 252]]

 ...

 [[252 252 252]
  [251 251 251]
  [252 252 252]
  ...
  [247 247 247]
  [249 249 249]
  [248 248 248]]

 [[253 253 253]
  [253 253 253]
  [251 251 251]
  ...
  [248 248 248]
  [248 248 248]
  [248 248 248]]

 [[252 252 252]
  [253 253 253]
  [252 252 252]
  ...
  [248 248 248]
  [247 247 247]
  [250 250 250]]]
```

Pre-processing the dataset

```
def format_example(image, label):
    # Make image color values to be float.
    image = tf.cast(image, tf.float32)
    # Make image color values to be in [0..1] range.
    image = image / 255.
    # Make sure that image has a right size
    image = tf.image.resize(image, [INPUT_IMG_SIZE, INPUT_IMG_SIZE])
    return image, label
```

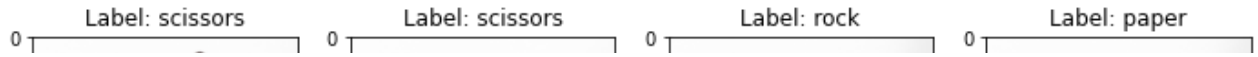
```
dataset_train = dataset_train_raw.map(format_example)
dataset_test = dataset_test_raw.map(format_example)
```

```
# Explore what values are used to represent the image.
(first_image, first_lable) = list(dataset_train.take(1))[0]
print('Label:', first_lable.numpy(), '\n')
print('Image shape:', first_image.numpy().shape, '\n')
print(first_image.numpy())
```

```
Label: 2

Image shape: (224, 224, 3)

[[[0.995526   0.995526   0.995526  ]
  [0.9941408  0.9941408  0.9941408 ]
  [0.99597746 0.99597746 0.99597746]
  ...
  [0.9869748  0.9869748  0.9869748 ]
  [0.98237604 0.98237604 0.98237604]
  [0.97995263 0.97995263 0.97995263]]

 [[0.99607843 0.99607843 0.99607843]
  [0.99509835 0.99509835 0.99509835]
  [0.99578613 0.99578613 0.99578613]
  ...
  [0.98232853 0.98232853 0.98232853]
  [0.98235357 0.98235357 0.98235357]
  [0.9824342  0.9824342  0.9824342 ]]

 [[0.99607843 0.99607843 0.99607843]
  [0.99438554 0.99438554 0.99438554]
  [0.9955736  0.9955736  0.9955736 ]
  ...
  [0.982799   0.982799   0.982799  ]
  [0.97900224 0.97900224 0.97900224]
  [0.98414266 0.98414266 0.98414266]]

 ...

 [[0.9886986  0.9886986  0.9886986 ]
  [0.98788357 0.98788357 0.98788357]
  [0.98773044 0.98773044 0.98773044]
  ...
```

```
       [0.97477514 0.97477514 0.97477514]
       [0.9725384  0.9725384  0.9725384 ]
       [0.96988803 0.96988803 0.96988803]]

      [[0.98982257 0.98982257 0.98982257]
       [0.9872209  0.9872209  0.9872209 ]
       [0.98630947 0.98630947 0.98630947]
       ...
       [0.9689198  0.9689198  0.9689198 ]
       [0.97251344 0.97251344 0.97251344]
       [0.9728876  0.9728876  0.9728876 ]]

      [[0.98945296 0.98945296 0.98945296]
       [0.9898225  0.9898225  0.9898225 ]
       [0.98757    0.98757    0.98757   ]
       ...
       [0.9692227  0.9692227  0.9692227 ]
       [0.9709499  0.9709499  0.9709499 ]
       [0.9774043  0.9774043  0.9774043 ]]]
```

```
# Explore preprocessed training dataset images.
preview_dataset(dataset_train)
```

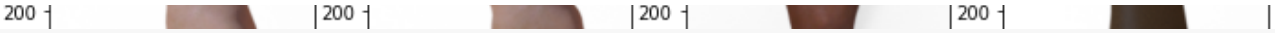


Data augmentation

One of the way to fight the model overfitting and to generalize the model to a broader set of examples is to augment the training data.

As you saw from the previous section all training examples have a white background and vertically positioned right hands. But what if the image with the hand will be horizontally positioned or what if the background will not be that bright. What if instead of a right hand the model will see a left hand. To make our model a little bit more universal we're going to flip and rotate images and also to adjust background colors.

You may read more about a Simple and efficient data augmentations using the Tensorfow tf.Data and Dataset API.



```
def augment_flip(image: tf.Tensor) -> tf.Tensor:
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    return image
```

```
def augment_color(image: tf.Tensor) -> tf.Tensor:
    image = tf.image.random_hue(image, max_delta=0.08)
    image = tf.image.random_saturation(image, lower=0.7, upper=1.3)
    image = tf.image.random_brightness(image, 0.05)
    image = tf.image.random_contrast(image, lower=0.8, upper=1)
    image = tf.clip_by_value(image, clip_value_min=0, clip_value_max=1)
    return image
```

```
def augment_rotation(image: tf.Tensor) -> tf.Tensor:
    # Rotate 0, 90, 180, 270 degrees
    return tf.image.rot90(
        image,
        tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32)
    )
```

```
def augment_inversion(image: tf.Tensor) -> tf.Tensor:
    random = tf.random.uniform(shape=[], minval=0, maxval=1)
    if random > 0.5:
        image = tf.math.multiply(image, -1)
        image = tf.math.add(image, 1)
    return image
```

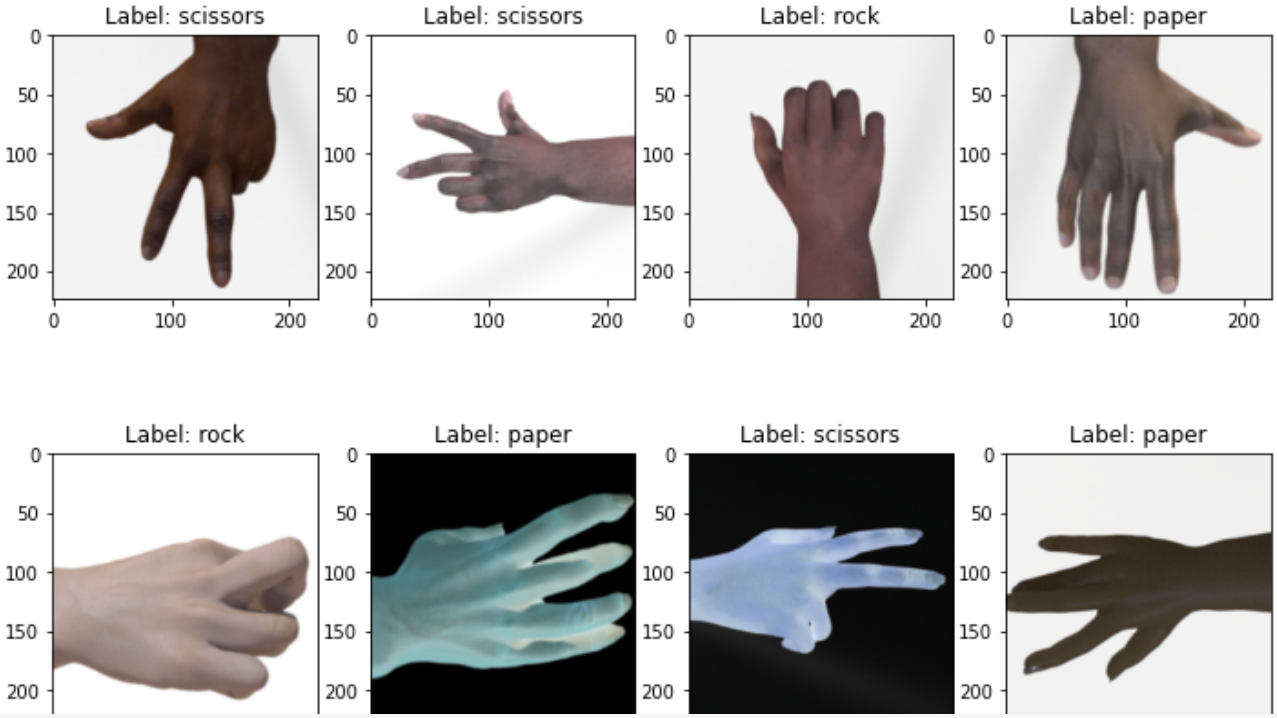
```
def augment_zoom(image: tf.Tensor, min_zoom=0.8, max_zoom=1.0) -> tf.Tensor:
    image_width, image_height, image_colors = image.shape
    crop_size = (image_width, image_height)

    # Generate crop settings, ranging from a 1% to 20% crop.
```

```
    scales = list(np.arange(min_zoom, max_zoom, 0.01))
    boxes = np.zeros((len(scales), 4))

    for i, scale in enumerate(scales):
        x1 = y1 = 0.5 - (0.5 * scale)
        x2 = y2 = 0.5 + (0.5 * scale)
        boxes[i] = [x1, y1, x2, y2]

    def random_crop(img):
        # Create different crops for an image
        crops = tf.image.crop_and_resize(
            [img],
            boxes=boxes,
            box_indices=np.zeros(len(scales)),
            crop_size=crop_size
        )
        # Return a random crop
        return crops[tf.random.uniform(shape=[], minval=0, maxval=len(scales), dty

    choice = tf.random.uniform(shape=[], minval=0., maxval=1., dtype=tf.float32)

    # Only apply cropping 50% of the time
    return tf.cond(choice < 0.5, lambda: image, lambda: random_crop(image))
```

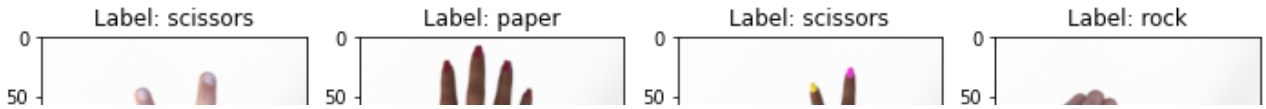
```
def augment_data(image, label):
    image = augment_flip(image)
    image = augment_color(image)
    image = augment_rotation(image)
    image = augment_zoom(image)
    image = augment_inversion(image)
    return image, label
```

```
dataset_train_augmented = dataset_train.map(augment_data)
```

```
# Explore augmented training dataset.
preview_dataset(dataset_train_augmented)
```




```
# Explore test dataset.
preview_dataset(dataset_test)
```



Data shuffling and batching

We don't want our model to learn anything from the order or grouping of the images in the dataset. To avoid that we will shuffle the training examples. Also we're going to split the training set by batches to speed up training process and make it less memory consuming.

```
BATCH_SIZE = 800

dataset_train_augmented_shuffled = dataset_train_augmented.shuffle(
    buffer_size=NUM_TRAIN_EXAMPLES
)

dataset_train_augmented_shuffled = dataset_train_augmented.batch(
    batch_size=BATCH_SIZE
)

# Prefetch will enable the input pipeline to asynchronously fetch batches while yo
dataset_train_augmented_shuffled = dataset_train_augmented_shuffled.prefetch(
    buffer_size=tf.data.experimental.AUTOTUNE
)

dataset_test_shuffled = dataset_test.batch(BATCH_SIZE)
```

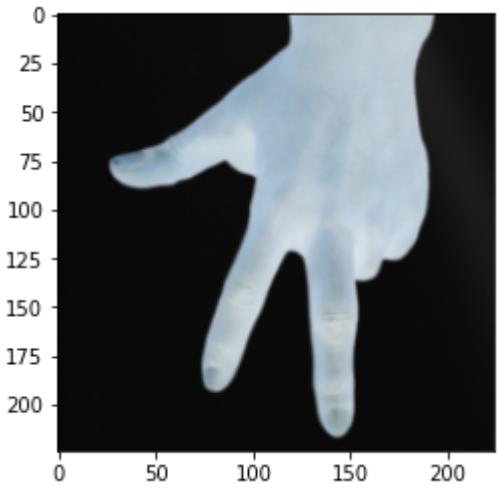
```
print(dataset_train_augmented_shuffled)
print(dataset_test_shuffled)
```

```
<PrefetchDataset element_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf
<BatchDataset element_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.fl
```

```
# Debugging the batches using conversion to Numpy arrays.
batches = tfds.as_numpy(dataset_train_augmented_shuffled)
for batch in batches:
    image_batch, label_batch = batch
    print('Label batch shape:', label_batch.shape, '\n')
    print('Image batch shape:', image_batch.shape, '\n')
    print('Label batch:', label_batch, '\n')

    for batch_item_index in range(len(image_batch)):
        print('First batch image:', image_batch[batch_item_index], '\n')
        plt.imshow(image_batch[batch_item_index])
        plt.show()
        # Break to shorten the output.
        break
    # Break to shorten the output.
    break
```

```
Label batch shape: (800,)

Image batch shape: (800, 224, 224, 3)

Label batch: [2 2 0 1 0 1 2 1 2 2 1 1 2 1 1 1 1 1 1 1 0 0 0 0 1 1 2 2 2 0 0
 2 1 0 0 0 0 0 1 1 2 2 0 0 2 1 1 0 0 1 2 1 0 0 0 0 1 2 1 1 2 2 1 1 1 1 1 2
 0 0 2 1 0 1 0 0 1 1 1 1 2 1 1 0 0 2 2 1 0 0 1 1 2 1 1 0 0 0 2 0 0 1 1 2 0
 2 0 1 1 1 2 0 1 0 1 2 1 0 1 2 2 0 2 1 0 0 1 0 1 0 1 2 1 2 2 1 0 2 0 1 1 2
 0 2 2 1 0 1 2 2 1 1 0 2 0 0 1 1 0 1 2 2 0 0 2 1 1 0 1 2 0 1 1 1 2 0 2 1 2
 1 1 1 2 2 2 1 0 2 0 1 0 1 2 0 0 2 0 1 1 0 2 2 2 1 1 1 0 1 0 2 0 0 1 1 1 2
 1 2 1 2 2 0 2 1 0 1 0 0 2 1 1 0 2 2 2 0 1 1 1 2 0 1 0 2 1 1 2 1 2 2 0 1 2
 2 0 2 1 0 2 0 0 1 0 2 2 0 0 2 2 0 0 2 2 1 0 0 0 2 1 1 0 2 0 1 1 1 2 1 1 0
 1 1 2 2 2 1 2 0 0 0 2 0 2 0 0 0 0 2 2 0 1 0 0 1 1 0 1 1 0 2 1 0 2 0 1 1 0
 2 1 0 0 1 2 2 0 1 1 2 2 2 0 2 2 2 0 2 1 2 0 2 1 0 1 1 1 0 2 0 1 0 1 0 0 0
 0 1 2 1 0 2 2 0 2 2 2 1 2 1 2 0 1 0 0 0 1 1 1 1 1 2 1 1 1 0 1 0 2 0 0 1 0
 0 2 0 0 1 1 1 1 2 2 2 1 1 0 1 2 0 1 2 0 0 1 1 0 2 2 1 2 2 1 1 0 2 1 2 2 0
 1 0 0 1 2 0 2 1 1 2 1 1 2 0 2 1 1 1 1 1 2 0 0 0 0 1 1 0 0 2 2 2 1 0 2 1 2
 0 1 2 0 1 2 0 2 1 0 0 1 1 2 0 2 2 0 1 1 2 0 1 2 2 0 1 1 2 1 2 1 0 1 2 1 1
 1 0 1 1 1 2 0 0 1 0 1 2 2 0 0 0 0 2 0 0 0 2 2 2 0 2 2 2 1 2 1 1 1 0 0 1 2
 0 2 0 1 1 0 0 2 1 0 0 1 2 0 0 0 1 1 2 1 1 0 2 0 0 1 2 1 1 0 0 0 1 2 1 1 0
 2 0 1 1 0 2 0 2 0 2 2 1 0 1 1 1 2 1 0 0 2 0 0 2 0 1 0 2 2 2 1 1 0 2 1 0 0
 2 0 1 1 0 2 2 0 1 2 0 2 2 1 2 0 2 2 2 2 2 0 0 2 2 1 2 0 2 0 1 2 0 2 0 0 2
 0 2 2 0 1 1 0 0 2 0 1 1 1 1 0 2 1 0 1 1 2 2 1 0 0 1 1 0 2 2 1 1 2 2 2 1 2
 1 2 1 2 1 2 1 2 0 1 1 1 1 2 0 1 2 2 1 2 1 2 1 2 1 2 0 0 1 0 0 1 0 2 1 1 2
 0 1 1 2 1 0 1 2 0 0 0 2 0 0 2 2 2 0 0 0 1 1 0 1 1 1 1 1 1 0 1 2 1 1 2 2 1
 0 0 0 0 1 1 1 2 2 0 2 0 0 2 2 1 1 2 2 2 2 1 1]

First batch image: [[[0.03961003 0.04078156 0.04158181]
  [0.03945327 0.0406248  0.04142505]
  [0.03937632 0.04054785 0.0413481 ]
  ...
  [0.05511546 0.05628705 0.05708724]
  [0.05506533 0.05623692 0.05703712]
  [0.05474198 0.05591357 0.05671376]]

 [[0.03894454 0.04011607 0.04091632]
  [0.03992707 0.04109859 0.04189885]
  [0.04030651 0.04147804 0.04227829]
  ...
  [0.05373991 0.05491149 0.05571169]
  [0.05399573 0.05516732 0.05596751]
  [0.05498832 0.05615991 0.05696011]]

 [[0.03742933 0.03860086 0.03940111]
  [0.04090953 0.04208106 0.04288131]
  [0.04145235 0.04262388 0.04342413]
  ...
  [0.05520302 0.05637455 0.0571748 ]
  [0.05519873 0.05637026 0.05717051]
  [0.05594653 0.05711806 0.05791831]]

 ...

 [[0.03455669 0.03572822 0.03652847]
  [0.03422618 0.03539771 0.03619796]
  [0.03446275 0.03563428 0.03643453]
  ...
  [0.0422284  0.04339993 0.04420018]
  [0.04246092 0.04363251 0.0444327 ]
  [0.0427838  0.04395533 0.04475558]]
```

```
[[0.03543866 0.03661019 0.03741044]
 [0.03501242 0.03618395 0.03698421]
 [0.03423661 0.03540814 0.03620839]
 ...
 [0.04182118 0.04299271 0.04379296]
 [0.04188967 0.04306126 0.04386145]
 [0.04121393 0.04238546 0.04318571]]

[[0.0322209  0.03339243 0.03419268]
 [0.03201157 0.0331831  0.03398335]
 [0.03320897 0.0343805  0.03518075]
 ...
 [0.04205447 0.043226   0.04402626]
 [0.04196447 0.043136   0.04393625]
 [0.04137576 0.04254729 0.04334754]]]
```



Creating the model

Loading model

We don't want to use the top classification layer of the pre-trained model as it contains 1000 classes when we need only 3 (`rock`, `paper` and `scissors`). We will specify that by setting a `include_top` parameter to `False`.

You may read more about Keras models on Keras Documentation

```
base_model = tf.keras.applications.NASNetMobile(
  input_shape=INPUT_IMG_SHAPE,
  include_top=False,
  weights='imagenet',
  pooling='avg'
)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
19996672/19993432 [==============================] - 1s 0us/step
20004864/19993432 [==============================] - 1s 0us/step
```

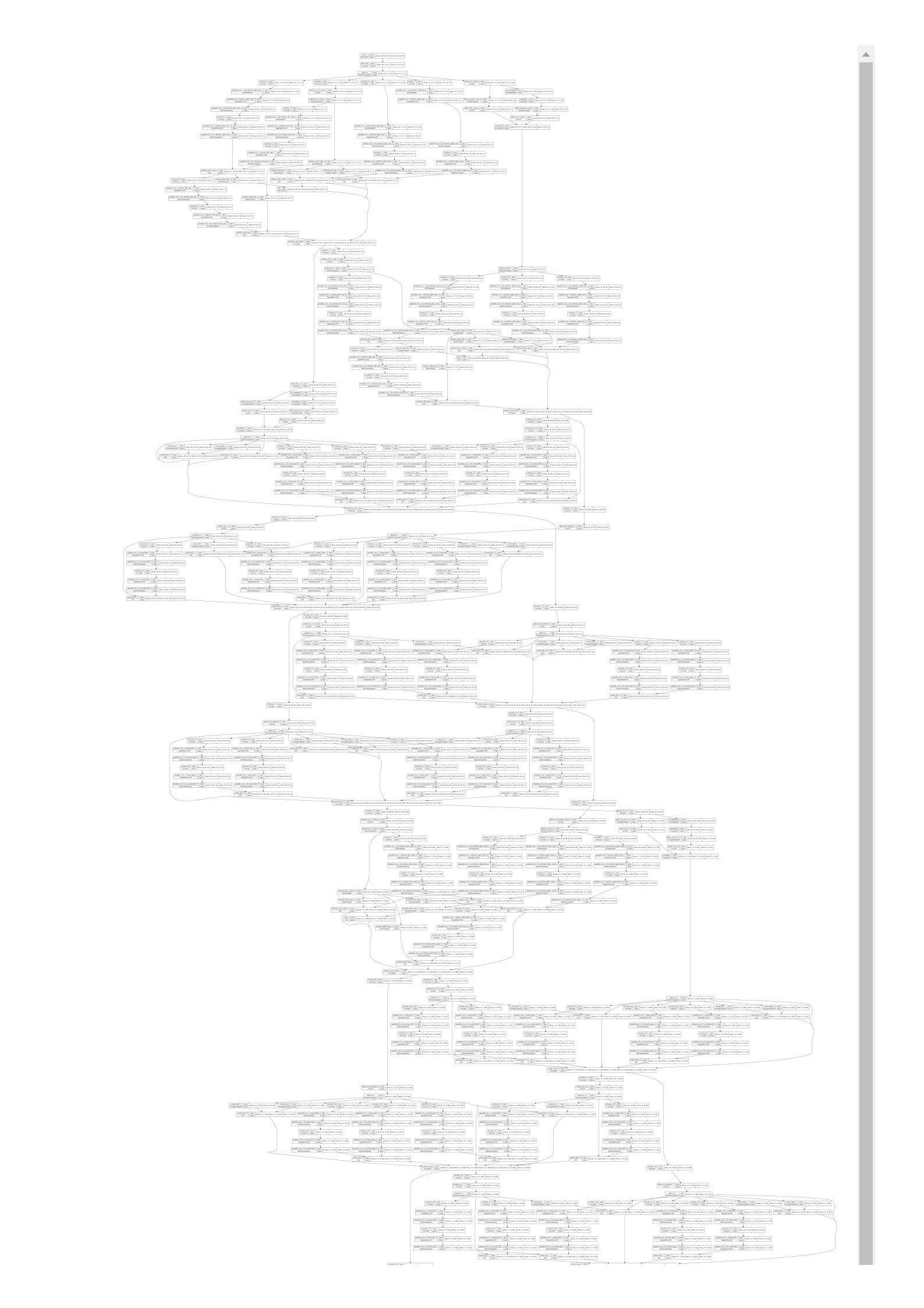
```
# Freezing the base model since we don't want to re-train it.
# We're only interesting in its feature extraction.
base_model.trainable = False
```

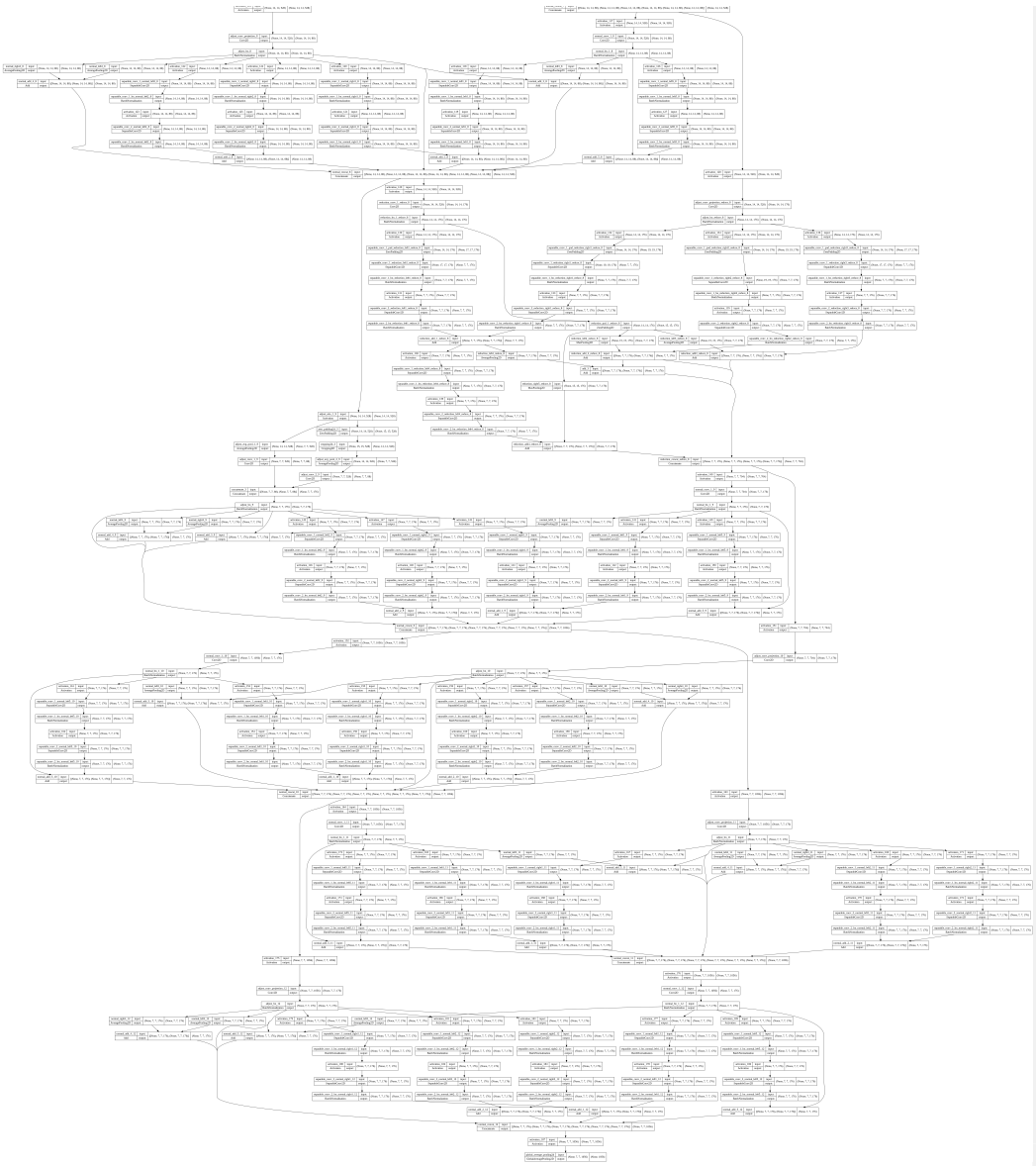
```
base_model.summary()
```

```
                                                                 'separabl
                                                                t2_10[0][0

 normal_add_3_10 (Add)          (None, 7, 7, 176)    0           ['normal_l
                                                                 'adjust_b

 normal_add_4_10 (Add)          (None, 7, 7, 176)    0           ['normal_l
                                                                 'normal_r

 normal_add_5_10 (Add)          (None, 7, 7, 176)    0           ['separabl
                                                                5_10[0][0]
                                                                 'normal_b

 normal_concat_10 (Concatenate)  (None, 7, 7, 1056)  0           ['adjust_b
                                                                 'normal_a
                                                                 'normal_a
                                                                 'normal_a
                                                                 'normal_a
                                                                 'normal_a

 activation_163 (Activation)    (None, 7, 7, 1056)   0           ['normal_c

 activation_164 (Activation)    (None, 7, 7, 1056)   0           ['normal_c

 adjust_conv_projection_11 (Con  (None, 7, 7, 176)   185856      ['activati
 v2D)

 normal_conv_1_11 (Conv2D)      (None, 7, 7, 176)    185856      ['activati

 adjust_bn_11 (BatchNormalizati  (None, 7, 7, 176)   704         ['adjust_c
 on)                                                             ']

 normal_bn_1_11 (BatchNormaliza  (None, 7, 7, 176)   704         ['normal_c
 tion)

 activation_165 (Activation)    (None, 7, 7, 176)    0           ['normal_b

 activation_167 (Activation)    (None, 7, 7, 176)    0           ['adjust_b

 activation_169 (Activation)    (None, 7, 7, 176)    0           ['adjust_b

 activation_171 (Activation)    (None, 7, 7, 176)    0           ['adjust_b

 activation_173 (Activation)    (None, 7, 7, 176)    0           ['normal_b

 separable_conv_1_normal_left1_  (None, 7, 7, 176)   35376       ['activati
 11 (SeparableConv2D)
```

```
separable_conv_1_normal_right1  (None, 7, 7, 176)   32560       ['activati
_11 (SeparableConv2D)

separable_conv_1_normal_left2_  (None, 7, 7, 176)   35376       ['activati
11 (SeparableConv2D)

separable_conv_1_normal_right2  (None, 7, 7, 176)   32560       ['activati
_11 (SeparableConv2D)

separable_conv_1_normal_left5   (None, 7, 7, 176)   32560       ['activati
```

```
tf.keras.utils.plot_model(
    base_model,
    show_shapes=True,
    show_layer_names=True,
)
```





Adding a classification head

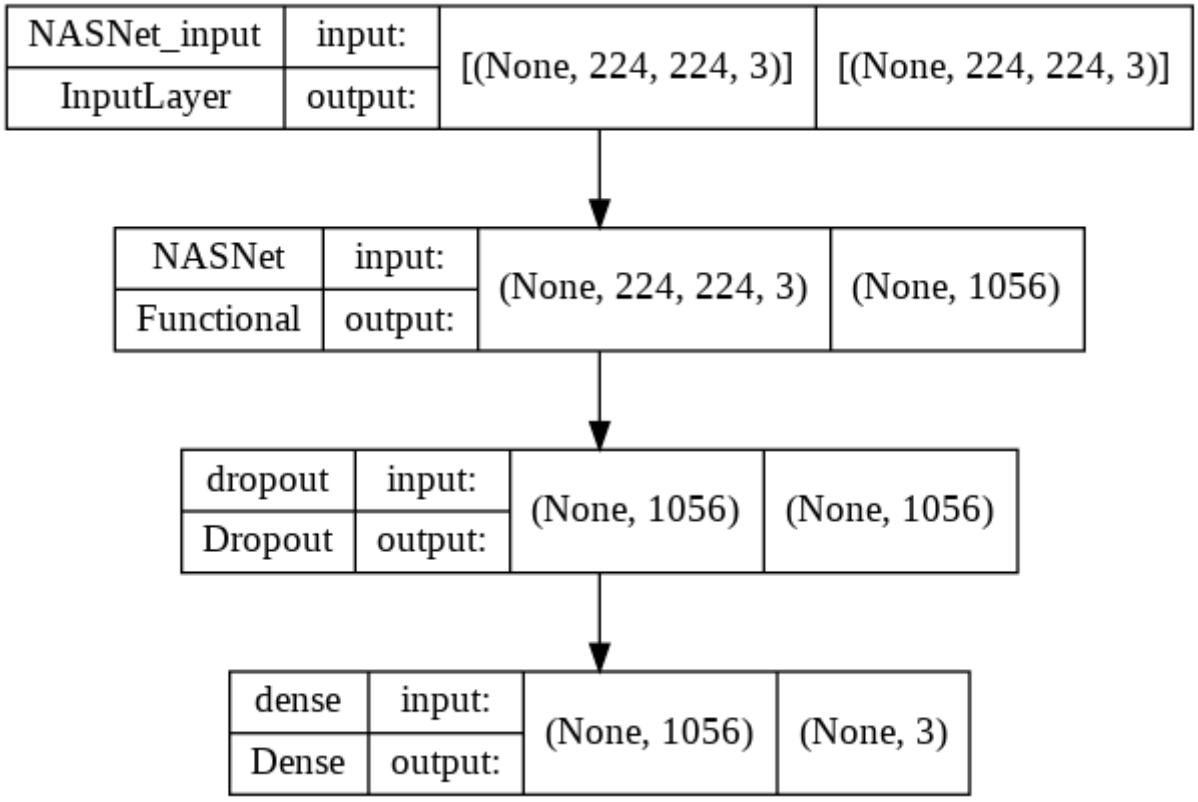
```
model = tf.keras.models.Sequential()

model.add(base_model)

# model.add(tf.keras.layers.GlobalAveragePooling2D())

model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.Dense(
    units=NUM_CLASSES,
    activation=tf.keras.activations.softmax,
    kernel_regularizer=tf.keras.regularizers.l2(l=0.01)
))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 NASNet (Functional)         (None, 1056)              4269716

 dropout (Dropout)           (None, 1056)              0

 dense (Dense)               (None, 3)                 3171

=================================================================
Total params: 4,272,887
Trainable params: 3,171
```

```
Non-trainable params: 4,269,716
```

```
tf.keras.utils.plot_model(
    model,
    show_shapes=True,
    show_layer_names=True,
)
```



Compiling the model

```
# adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
rmsprop_optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001)

model.compile(
    optimizer=rmsprop_optimizer,
    loss=tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['accuracy']
)
```

Training the model

```
steps_per_epoch = NUM_TRAIN_EXAMPLES // BATCH_SIZE
validation_steps = NUM_TEST_EXAMPLES // BATCH_SIZE if NUM_TEST_EXAMPLES // BATCH_S

print('steps_per_epoch:', steps_per_epoch)
print('validation_steps:', validation_steps)
```

```
    steps_per_epoch: 3
    validation_steps: 1
```

```
!rm -rf tmp/checkpoints
!rm -rf logs
```

```
# Preparing callbacks.
os.makedirs('logs/fit', exist_ok=True)
tensorboard_log_dir = 'logs/fit/' + datetime.datetime.now().strftime('%Y%m%d-%H%M%
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir=tensorboard_log_dir,
    histogram_freq=1
)

os.makedirs('tmp/checkpoints', exist_ok=True)
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath='tmp/checkpoints/weights.{epoch:02d}-{val_loss:.2f}.hdf5'
)

early_stopping_callback = tf.keras.callbacks.EarlyStopping(
    patience=10,
    monitor='val_accuracy'
    # monitor='val_loss'
)
```

```
initial_epochs = 20
```

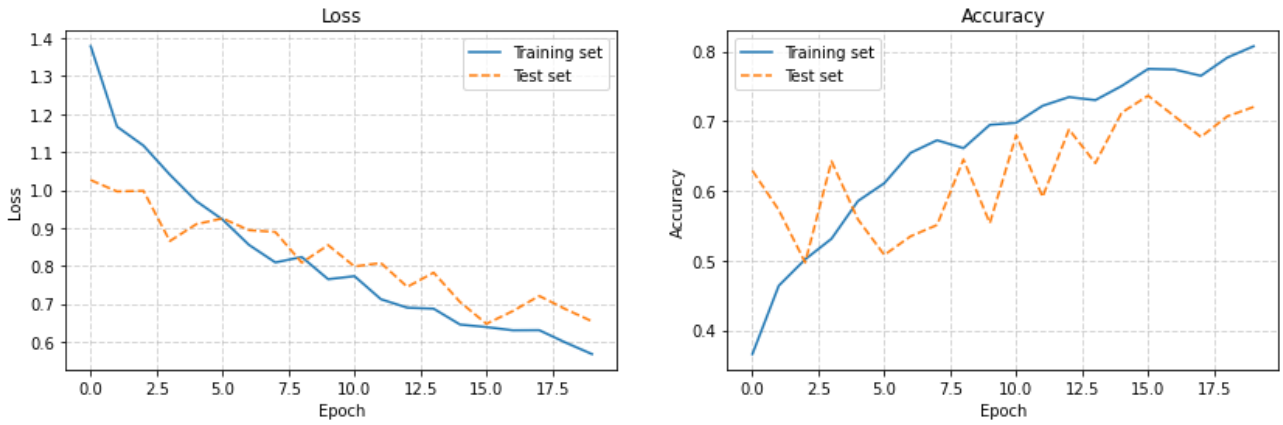
```
training_history = model.fit(
    x=dataset_train_augmented_shuffled.repeat(),
    validation_data=dataset_test_shuffled.repeat(),
    epochs=initial_epochs,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[
        # model_checkpoint_callback,
        # early_stopping_callback,
        tensorboard_callback
    ],
    verbose=2
)
```

```
    Epoch 1/20
    3/3 - 49s - loss: 1.3788 - accuracy: 0.3658 - val_loss: 1.0261 - val_accuracy
    Epoch 2/20
    3/3 - 25s - loss: 1.1668 - accuracy: 0.4640 - val_loss: 0.9962 - val_accuracy
    Epoch 3/20
    3/3 - 23s - loss: 1.1169 - accuracy: 0.5017 - val_loss: 0.9978 - val_accuracy
    Epoch 4/20
    3/3 - 17s - loss: 1.0405 - accuracy: 0.5314 - val_loss: 0.8654 - val_accuracy
    Epoch 5/20
    3/3 - 31s - loss: 0.9708 - accuracy: 0.5854 - val_loss: 0.9098 - val_accuracy
    Epoch 6/20
    3/3 - 25s - loss: 0.9220 - accuracy: 0.6110 - val_loss: 0.9252 - val_accuracy
```

```
Epoch 7/20
3/3 - 20s - loss: 0.8553 - accuracy: 0.6547 - val_loss: 0.8942 - val_accuracy
Epoch 8/20
3/3 - 18s - loss: 0.8091 - accuracy: 0.6727 - val_loss: 0.8893 - val_accuracy
Epoch 9/20
3/3 - 30s - loss: 0.8235 - accuracy: 0.6612 - val_loss: 0.8085 - val_accuracy
Epoch 10/20
3/3 - 25s - loss: 0.7650 - accuracy: 0.6948 - val_loss: 0.8547 - val_accuracy
Epoch 11/20
3/3 - 20s - loss: 0.7729 - accuracy: 0.6977 - val_loss: 0.7985 - val_accuracy
Epoch 12/20
3/3 - 17s - loss: 0.7122 - accuracy: 0.7221 - val_loss: 0.8077 - val_accuracy
Epoch 13/20
3/3 - 31s - loss: 0.6899 - accuracy: 0.7346 - val_loss: 0.7447 - val_accuracy
Epoch 14/20
3/3 - 25s - loss: 0.6873 - accuracy: 0.7302 - val_loss: 0.7823 - val_accuracy
Epoch 15/20
3/3 - 20s - loss: 0.6455 - accuracy: 0.7506 - val_loss: 0.7048 - val_accuracy
Epoch 16/20
3/3 - 17s - loss: 0.6388 - accuracy: 0.7750 - val_loss: 0.6474 - val_accuracy
Epoch 17/20
3/3 - 31s - loss: 0.6303 - accuracy: 0.7742 - val_loss: 0.6810 - val_accuracy
Epoch 18/20
3/3 - 26s - loss: 0.6307 - accuracy: 0.7651 - val_loss: 0.7209 - val_accuracy
Epoch 19/20
3/3 - 20s - loss: 0.5980 - accuracy: 0.7913 - val_loss: 0.6859 - val_accuracy
Epoch 20/20
3/3 - 17s - loss: 0.5676 - accuracy: 0.8076 - val_loss: 0.6541 - val_accuracy
```

```
def render_training_history(training_history):
    loss = training_history.history['loss']
    val_loss = training_history.history['val_loss']

    accuracy = training_history.history['accuracy']
    val_accuracy = training_history.history['val_accuracy']

    plt.figure(figsize=(14, 4))

    plt.subplot(1, 2, 1)
    plt.title('Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.plot(loss, label='Training set')
    plt.plot(val_loss, label='Test set', linestyle='--')
    plt.legend()
    plt.grid(linestyle='--', linewidth=1, alpha=0.5)

    plt.subplot(1, 2, 2)
    plt.title('Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.plot(accuracy, label='Training set')
    plt.plot(val_accuracy, label='Test set', linestyle='--')
    plt.legend()
    plt.grid(linestyle='--', linewidth=1, alpha=0.5)
```

```
    plt.show()
render_training_history(training_history)
```



Model fine tuning

We may try to unfreeze some of the top layers of the `base_model` and to train it a little bit more so to adjust top layers to our Rock-Paper-Scissors dataset.

```
# Un-freeze the top layers of the model
base_model.trainable = True

print("Number of layers in the base model: ", len(base_model.layers))
```

```
Number of layers in the base model:  770
```

```
# Fine tune from this layer onwards.
# fine_tune_at = 149  # MobileNetV2
fine_tune_at = 752

# Freeze all the layers before the `fine_tune_at` layer
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable =  False

# Compile the model using a much-lower training rate.
adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
rmsprop_optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.0001)
model.compile(
    optimizer = rmsprop_optimizer,
    loss=tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['accuracy']
)

model.summary()
```

```
Model: "sequential"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 NASNet (Functional)         (None, 1056)              4269716

 dropout (Dropout)           (None, 1056)              0

 dense (Dense)               (None, 3)                 3171

=================================================================
Total params: 4,272,887
Trainable params: 70,051
Non-trainable params: 4,202,836
_________________________________________________________________
```

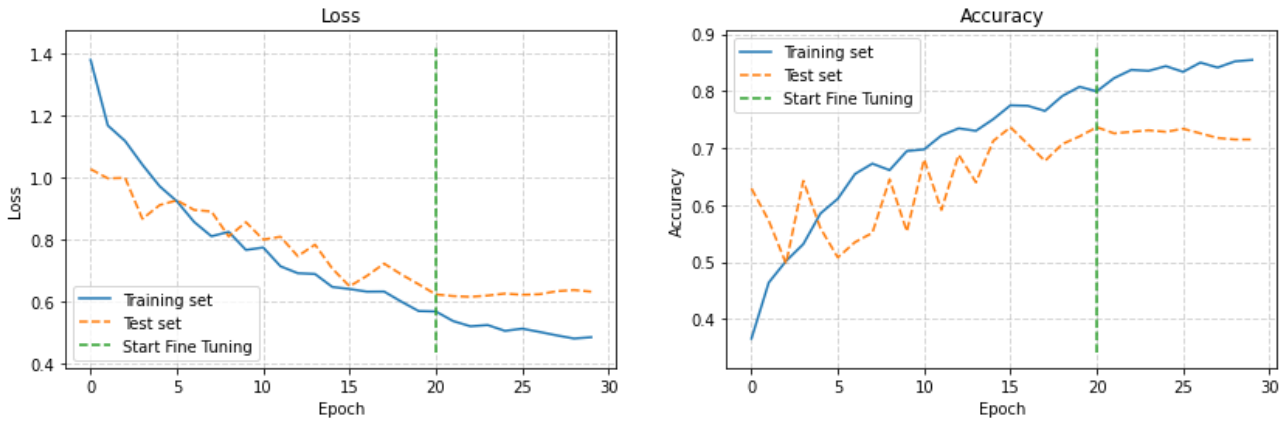
```
# The number of additional epochs during which we're going to fine tune the model.
fine_tuning_epochs = 10
```

```
training_history_fine = model.fit(
    x=dataset_train_augmented_shuffled.repeat(),
    validation_data=dataset_test_shuffled.repeat(),
    epochs=initial_epochs + fine_tuning_epochs,
    initial_epoch=initial_epochs,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[tensorboard_callback],
    verbose=1
)
```

```
Epoch 21/30
3/3 [==============================] - 46s 12s/step - loss: 0.5662 - accuracy
Epoch 22/30
3/3 [==============================] - 25s 12s/step - loss: 0.5352 - accuracy
Epoch 23/30
3/3 [==============================] - 21s 9s/step - loss: 0.5184 - accuracy:
Epoch 24/30
3/3 [==============================] - 17s 7s/step - loss: 0.5224 - accuracy:
Epoch 25/30
3/3 [==============================] - 31s 11s/step - loss: 0.5036 - accuracy
Epoch 26/30
3/3 [==============================] - 24s 12s/step - loss: 0.5107 - accuracy
Epoch 27/30
3/3 [==============================] - 20s 9s/step - loss: 0.5002 - accuracy:
Epoch 28/30
3/3 [==============================] - 17s 7s/step - loss: 0.4890 - accuracy:
Epoch 29/30
3/3 [==============================] - 31s 11s/step - loss: 0.4789 - accuracy
Epoch 30/30
3/3 [==============================] - 25s 12s/step - loss: 0.4831 - accuracy
```

```
loss = training_history.history['loss'] + training_history_fine.history['loss']
val_loss = training_history.history['val_loss'] + training_history_fine.history['v

accuracy = training_history.history['accuracy'] + training_history_fine.history['a
val_accuracy = training_history.history['val_accuracy'] + training_history_fine.hi
```

```
plt.figure(figsize=(14, 4))

plt.subplot(1, 2, 1)
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.plot(loss, label='Training set')
plt.plot(val_loss, label='Test set', linestyle='--')
plt.plot(
[initial_epochs, initial_epochs],
plt.ylim(),
label='Start Fine Tuning',
linestyle='--'
)
plt.legend()
plt.grid(linestyle='--', linewidth=1, alpha=0.5)

plt.subplot(1, 2, 2)
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.plot(accuracy, label='Training set')
plt.plot(val_accuracy, label='Test set', linestyle='--')
plt.plot(
[initial_epochs, initial_epochs],
plt.ylim(),
label='Start Fine Tuning',
linestyle='--'
)
plt.legend()
plt.grid(linestyle='--', linewidth=1, alpha=0.5)

plt.show()
```



Debugging the training with TensorBoard

Нейронні мережі

Лекція_07

********** **********

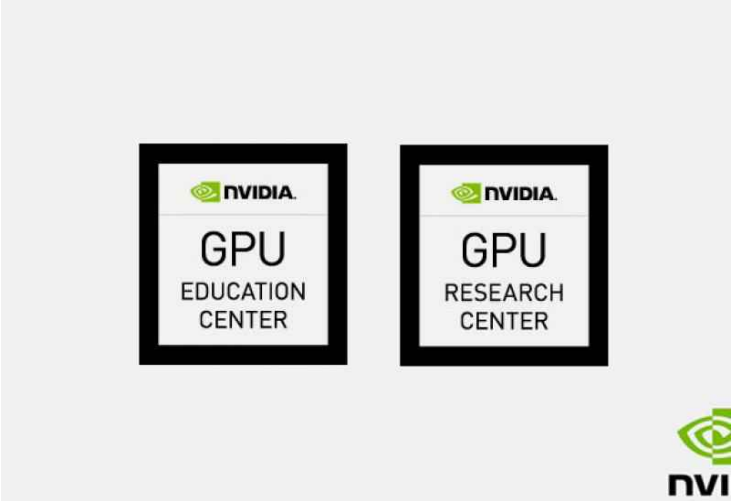
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 07- Нейронні мережі - Розгортання моделі

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО А -ЦП
Приклад розгортання моделі глибокого навчання - MNIST WebApp (Flask + Google Colab)
https://drive.google.com/file/d/1ywWNaf8Y2MUG526p1tiKi6lHyDkcmz3C/view?usp=sharing

ДЕМО В -GPU
Приклад розгортання моделі глибокого навчання - MNIST WebApp (Flask + Google Colab)
https://drive.google.com/file/d/11eReb0X2kJ3KScPHNM5I1XpLq2R560V0/view?usp=sharing

ДЕМО С-TPU
Приклад розгортання моделі глибокого навчання - MNIST WebApp (Flask + Google Colab)
https://drive.google.com/file/d/1X8soRab064l5ROqCv1z8JSDr3JJUBohK/view?usp=sharing

Lecture 7 - DEMO A - CPU - Deep Learning Model Deployment Example - MNIST WebApp (Flask + Google Colab)

based on (C) Tensorflow Authors Team, Parsaniya, Heaton, Jadhav and other works

Connect to Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Go to Project Folder at Google Drive and Check It

```
%cd 'drive/MyDrive/2022_COLAB_NN/Lecture_07_DL_Web-app'
! ls
```

```
/content/drive/MyDrive/2022_COLAB_NN/Lecture_07_DL_Web-app
generated_image                                 MNIST_test_images
Lecture_07_DL_Web-app.zip                       model
Lecture_07_MNIST_DEMO_A_web_app_CPU_EMPTY.ipynb  static
Lecture_07_MNIST_DEMO_B_web_app_GPU_EMPTY.ipynb  templates
Lecture_07_MNIST_DEMO_C_web_app_TPU_EMPTY.ipynb  uploads
```

Install Flask

```
!pip install flask-ngrok
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Collecting flask-ngrok
  Downloading flask_ngrok-0.0.25-py3-none-any.whl (3.1 kB)
Requirement already satisfied: Flask>=0.8 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: itsdangerous<2.0,>=0.24 in /usr/local/lib/pyth
Requirement already satisfied: Jinja2<3.0,>=2.10.1 in /usr/local/lib/python3.
Requirement already satisfied: click<8.0,>=5.1 in /usr/local/lib/python3.7/di
Requirement already satisfied: Werkzeug<2.0,>=0.15 in /usr/local/lib/python3.
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /us
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
```

```
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/
Installing collected packages: flask-ngrok
Successfully installed flask-ngrok-0.0.25
```

Import Libraries

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from flask import Flask, flash, redirect, render_template, request, url_for, send_
from flask_ngrok import run_with_ngrok
from tensorflow.keras.models import load_model
```

Load Trained Model

```
mnist_model = load_model('model/mnist.h5')
```

Configure Web-app

```
app = Flask(__name__)
run_with_ngrok(app)
app.secret_key = 'Putin_Huylo'

app.config["MNIST_BAR"] = "generated_image/mnist_vis"
app.config["IMAGES"] = "upload"

@app.route('/')
def home():
    flash("Try CNN Model Trained on MNIST-dataset for Single Digit Prediction...")
    return render_template('index.html')

@app.route('/mnist/')
def mnist_home():
    return render_template('mnist.html')

@app.route('/mnistprediction/', methods=['GET', 'POST'])
def mnist_prediction():
    if request.method == "POST":
        if not request.files['file'].filename:
            flash("No File Found")
        else:
            f = request.files['file']
            f.save("uploads/"+f.filename)
            image_gray  = cv2.imread("uploads/"+f.filename, cv2.IMREAD_GRAYSCALE)
            img_resize = cv2.resize(image_gray,(28,28))
```

Saved successfully!

```
            image_bw = cv2.threshold(img_resize, 75, 255, cv2.THRESH_BINARY)[1]
            bitwise_not_image = cv2.bitwise_not(image_bw, mask=None)
            pred_img = np.reshape(bitwise_not_image,(1,28,28,1))/255.0

            predictions = mnist_model.predict(pred_img)
            number = int(np.argmax(predictions))
            print(number)

            plt.figure()
            y_pos = np.arange(10)
            plt.bar(y_pos, predictions[0])
            plt.savefig('generated_image/mnist_vis/'+f.filename)

            return str(number)

@app.route("/get-mnist-image/<image_name>")
def get_mnist_image(image_name):
    try:
        return send_from_directory(app.config["MNIST_BAR"], filename=image_name)
    except FileNotFoundError:
        abort(404)
```

```
# Install pyngrok
!pip install pyngrok==4.1.1

# Register, get 'your authtoken', and replace my token below:
# !ngrok authtoken 'your authtoken'
!ngrok authtoken '2FdbDL8Rak9en9IT4S3pSeMjq0I_6Ntx7LfKFyeS9qLSFAoks'
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Collecting pyngrok==4.1.1
  Downloading pyngrok-4.1.1.tar.gz (18 kB)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: PyYAML in /usr/local/lib/python3.7/dist-packag
Building wheels for collected packages: pyngrok
  Building wheel for pyngrok (setup.py) ... done
  Created wheel for pyngrok: filename=pyngrok-4.1.1-py3-none-any.whl size=159
  Stored in directory: /root/.cache/pip/wheels/b1/d9/12/045a042fee3127dc40ba6
Successfully built pyngrok
Installing collected packages: pyngrok
Successfully installed pyngrok-4.1.1
Authtoken saved to configuration file: /root/.ngrok2/ngrok.yml
```

Start Web-app

After start ...

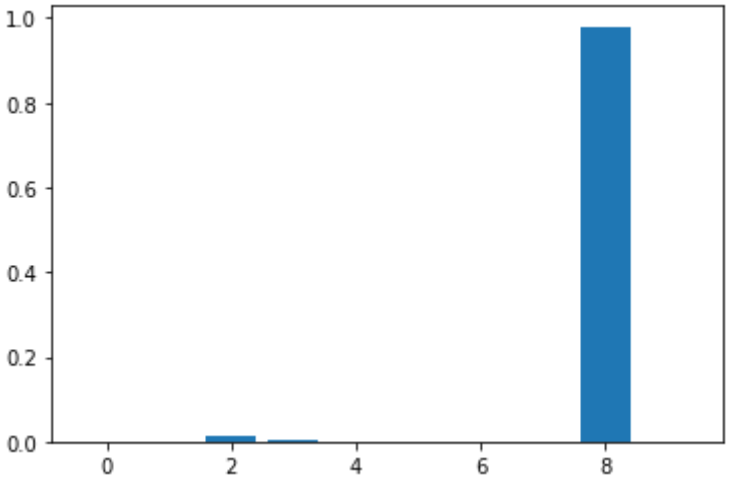
- click on the link in the row like:

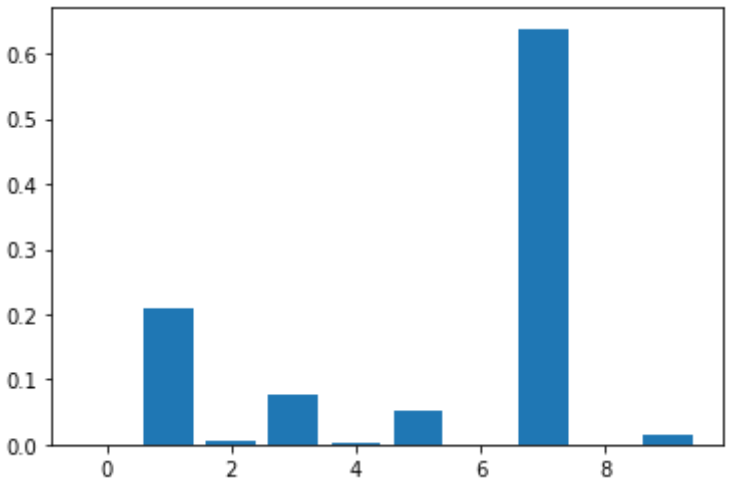
Saved successfully!

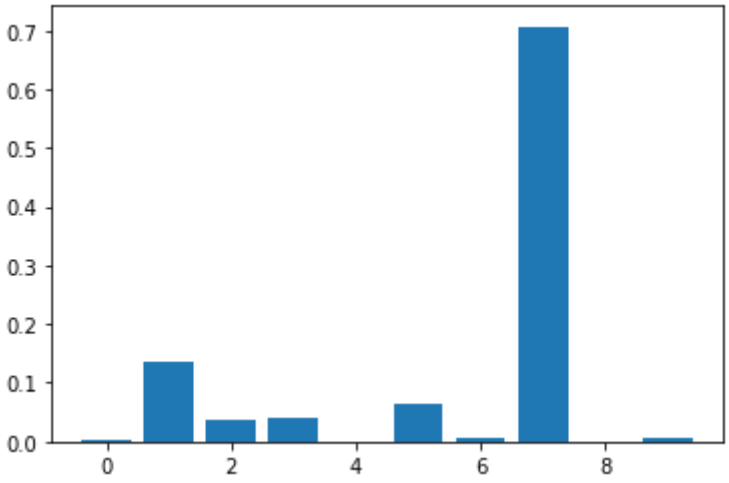
- load the local images of single digit numbers and obtain predictions;
- try images of different quality.

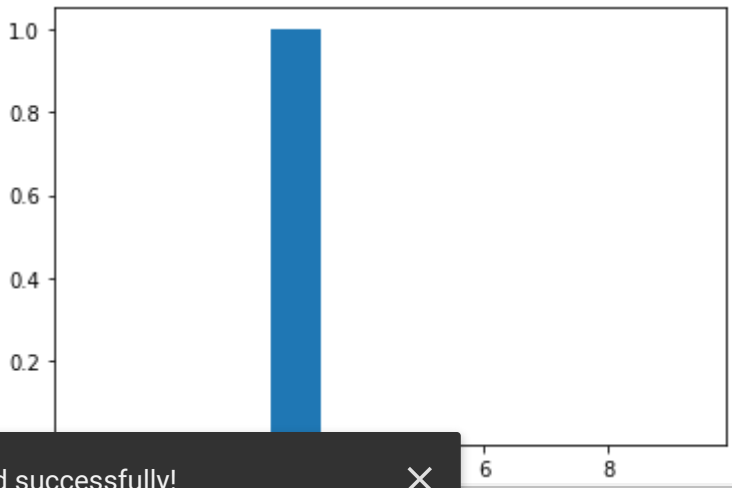
IMPORTANT: this Web-app is cloud-based and ... **some delay can be observed!**

```
app.run()
```







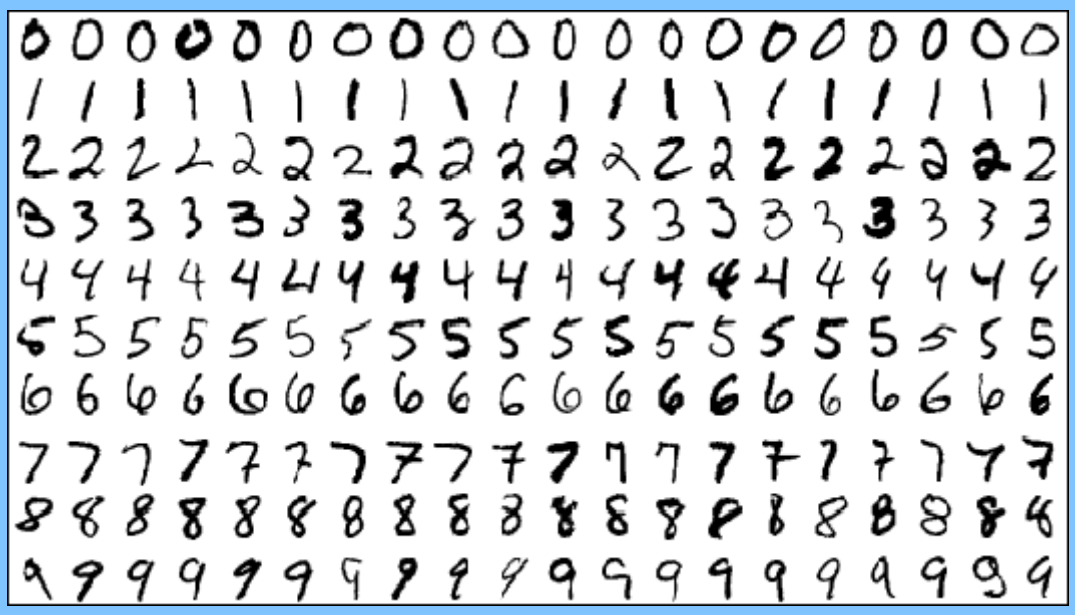


Saved successfully!

Home MNIST

message : Try CNN Model Trained on MNIST-dataset for Single Digit Prediction... ×

Deployed Model



Predict MNIST number

Home MNIST

MNIST

Upload Image

Browse Image...

Predict MNIST

Image Preview

your image

Prediction

Prediction bar-plot

Home MNIST

MNIST

Upload Image

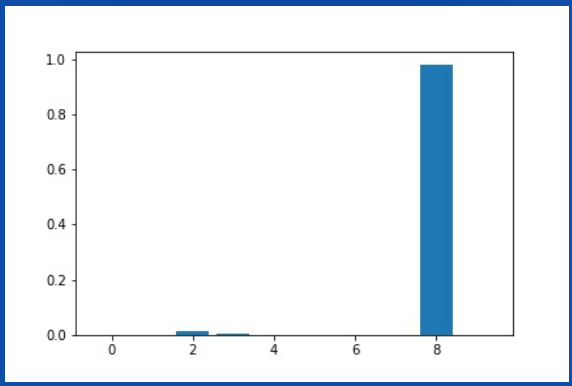
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 8



Home MNIST

MNIST

Upload Image

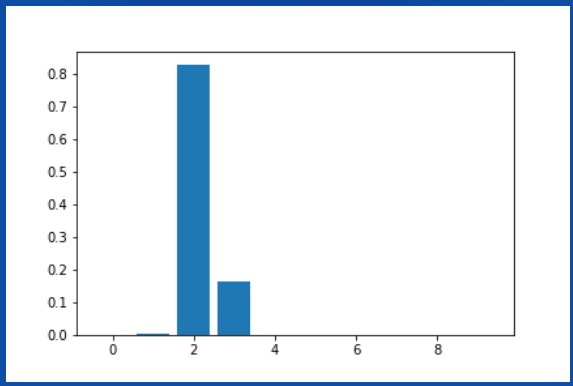
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 2



Home MNIST

MNIST

Upload Image

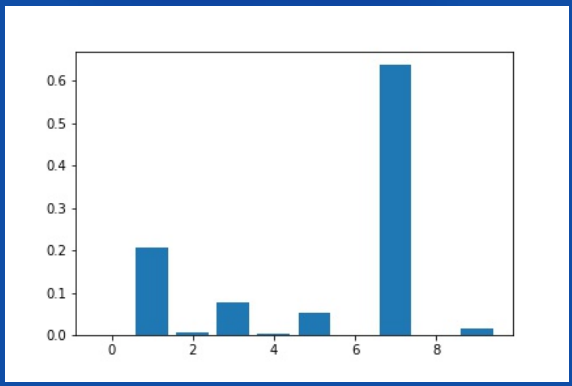
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 7



Home MNIST

MNIST

Upload Image

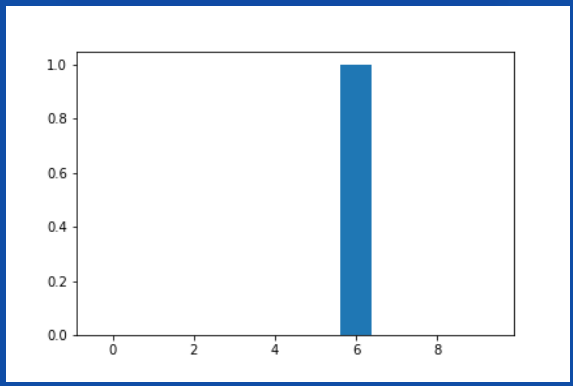
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 6



Home MNIST

MNIST

Upload Image

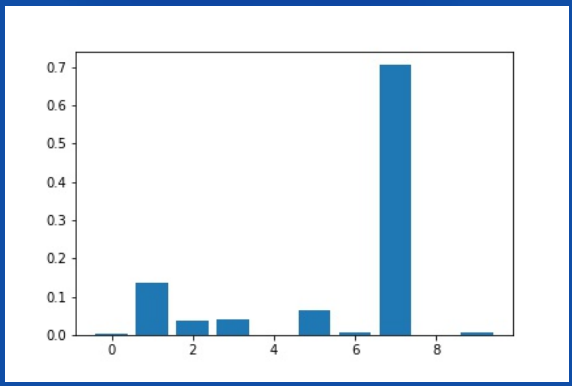
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 7



Home MNIST

MNIST

Upload Image

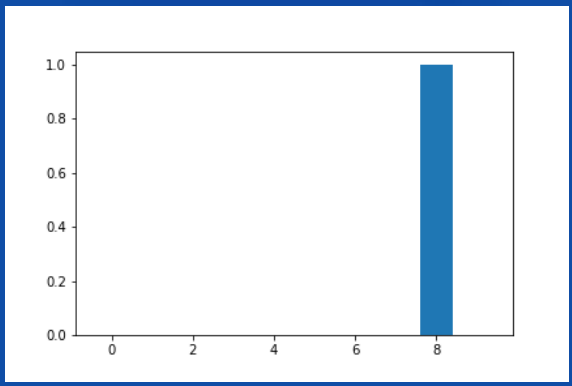
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 8



Home MNIST

MNIST

Upload Image

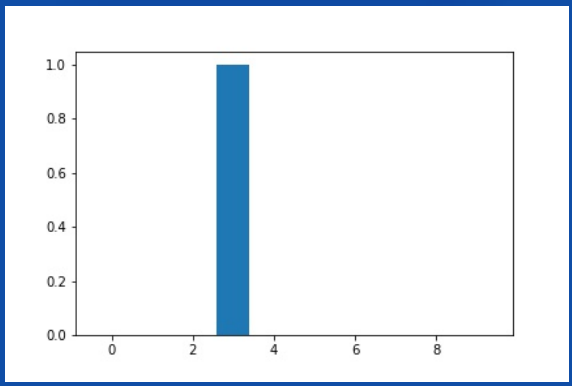
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 3



Home MNIST

MNIST

Upload Image

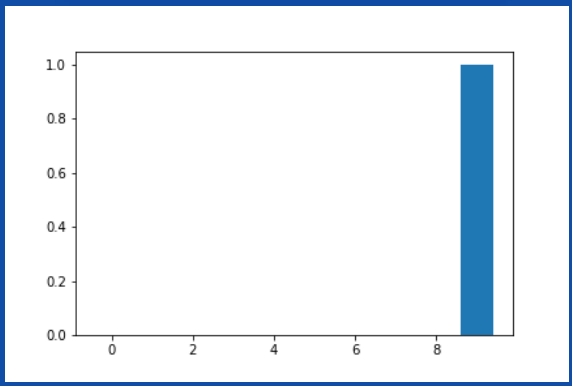
Browse Image...

Predict MNIST

Image Preview

Prediction

Predicted Number : 9



Lecture 7 - DEMO B - GPU - Deep Learning Model Deployment Example - MNIST WebApp (Flask + Google Colab)

based on (C) Tensorflow Authors Team, Parsaniya, Heaton, Jadhav and other works

```
! nvidia-smi
```

```
Mon Oct  3 19:43:07 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   43C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Connect to Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Go to Project Folder at Google Drive and Check It

```
%cd 'drive/MyDrive/2022_COLAB_NN/Lecture_07_DL_Web-app'
! ls
```

```
/content/drive/MyDrive/2022_COLAB_NN/Lecture_07_DL_Web-app
generated_image                                 MNIST_test_images
Lecture_07_DL_Web-app.zip                       model
```

Saved successfully!

```
_app_CPU_EMPTY.ipynb  static
_app_GPU_EMPTY.ipynb  templates
Lecture_07_MNIST_DEMO_C_web_app_TPU_EMPTY.ipynb  uploads
```

Install Flask

```
!pip install flask-ngrok
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Collecting flask-ngrok
  Downloading flask_ngrok-0.0.25-py3-none-any.whl (3.1 kB)
Requirement already satisfied: Flask>=0.8 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: itsdangerous<2.0,>=0.24 in /usr/local/lib/pyth
Requirement already satisfied: Jinja2<3.0,>=2.10.1 in /usr/local/lib/python3.
Requirement already satisfied: Werkzeug<2.0,>=0.15 in /usr/local/lib/python3.
Requirement already satisfied: click<8.0,>=5.1 in /usr/local/lib/python3.7/di
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/d
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /us
Installing collected packages: flask-ngrok
Successfully installed flask-ngrok-0.0.25
```

Import Libraries

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from flask import Flask, flash, redirect, render_template, request, url_for, send_
from flask_ngrok import run_with_ngrok
from tensorflow.keras.models import load_model
```

Load Trained Model

```
mnist_model = load_model('model/mnist.h5')
```

```
mnist_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320
```

```
 conv2d_1 (Conv2D)           (None, 24, 24, 32)        9248

 max_pooling2d (MaxPooling2D  (None, 12, 12, 32)       0
 )

 dropout (Dropout)           (None, 12, 12, 32)        0

 conv2d_2 (Conv2D)           (None, 12, 12, 64)        18496

 conv2d_3 (Conv2D)           (None, 12, 12, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 6, 6, 64)          0

 conv2d_4 (Conv2D)           (None, 6, 6, 128)         73856

 dropout_2 (Dropout)         (None, 6, 6, 128)         0

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 128)               589952

 batch_normalization (BatchN  (None, 128)              512
 ormalization)

 dropout_3 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 730,602
Trainable params: 730,346
Non-trainable params: 256
_________________________________________________________________
```

Configure Web-app

```
app = Flask(__name__)
run_with_ngrok(app)
app.secret_key = 'ACAB_таки_да_ACAB'

app.config["MNIST_BAR"] = "generated_image/mnist_vis"
app.config["IMAGES"] = "upload"

@app.route('/')
def home():
    flash("Try CNN Model Trained on MNIST-dataset for Single Digit Prediction...")
    return render_template('index.html')

@app.route('/mnist/')
def mnist_home():
    return render_template('mnist.html')
```

Saved successfully!

```
methods=['GET', 'POST'])
def mnist_prediction():
    if request.method == "POST":
        if not request.files['file'].filename:
            flash("No File Found")
        else:
            f =  request.files['file']
            f.save("uploads/"+f.filename)
            image_gray  = cv2.imread("uploads/"+f.filename, cv2.IMREAD_GRAYSCALE)
            img_resize = cv2.resize(image_gray,(28,28))
            image_bw = cv2.threshold(img_resize, 75, 255, cv2.THRESH_BINARY)[1]
            bitwise_not_image = cv2.bitwise_not(image_bw, mask=None)
            pred_img = np.reshape(bitwise_not_image,(1,28,28,1))/255.0

            predictions = mnist_model.predict(pred_img)
            number = int(np.argmax(predictions))
            print(number)

            plt.figure()
            y_pos = np.arange(10)
            plt.bar(y_pos, predictions[0])
            plt.savefig('generated_image/mnist_vis/'+f.filename)

            return str(number)

@app.route("/get-mnist-image/<image_name>")
def get_mnist_image(image_name):
    try:
        return send_from_directory(app.config["MNIST_BAR"], filename=image_name)
    except FileNotFoundError:
        abort(404)
```

```
# Install pyngrok
!pip install pyngrok==4.1.1

# Register, get 'your authtoken', and replace my token below:
# !ngrok authtoken 'your authtoken'
!ngrok authtoken '2FdbDL8Rak9en9IT4S3pSeMjq0I_6Ntx7LfKFyeS9qLSFAoks'
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Collecting pyngrok==4.1.1
  Downloading pyngrok-4.1.1.tar.gz (18 kB)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: PyYAML in /usr/local/lib/python3.7/dist-packag
Building wheels for collected packages: pyngrok
  Building wheel for pyngrok (setup.py) ... done
  Created wheel for pyngrok: filename=pyngrok-4.1.1-py3-none-any.whl size=15
  Stored in directory: /root/.cache/pip/wheels/b1/d9/12/045a042fee3127dc40ba
Successfully built pyngrok
Installing collected packages: pyngrok
Successfully installed pyngrok-4.1.1
Authtoken saved to configuration file: /root/.ngrok2/ngrok.yml
```

After start ...

- click on the link in the row like:

Running on [your_website_at_ngrok.io]

- follow the web-user interface:
    - load the local images of single digit numbers and obtain predictions;
    - try images of different quality.

IMPORTANT: this Web-app is cloud-based and ... **some delay can be observed!**

```
app.run()
```

```
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deplo
   Use a production WSGI server instead.
 * Debug mode: off
INFO:werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Running on http://dbd3-34-72-230-125.ngrok.io
 * Traffic stats available on http://127.0.0.1:4040
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:44:12] "GET / HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:44:12] "GET /static/css/main.css
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:44:12] "GET /static/css/header_fi
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:44:13] "GET /static/image/mnist-s
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:44:13] "GET /favicon.ico HTTP/1.1
```

or Self-Guided Experiments:

- try to train, save and use other *.h5 model (like it was described in the previous DEMOs),
- try to use other datasets and related models,
- try to port the web-app to your local environment,
- ...

Lecture 8 - DEMO C - TPU - Deep Learning Model Deployment Example - MNIST WebApp (Flask + Google Colab)

based on (C) Tensorflow Authors Team, Parsaniya, Heaton, Jadhav and other works

Connect to Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Go to Project Folder at Google Drive and Check It

```
%cd 'drive/MyDrive/2022_COLAB_NN/Lecture_07_DL_Web-app'
! ls
```

```
/content/drive/MyDrive/2022_COLAB_NN/Lecture_07_DL_Web-app
generated_image                                 MNIST_test_images
Lecture_07_DL_Web-app.zip                       model
Lecture_07_MNIST_DEMO_A_web_app_CPU_EMPTY.ipynb  static
Lecture_07_MNIST_DEMO_B_web_app_GPU_EMPTY.ipynb  templates
Lecture_07_MNIST_DEMO_C_web_app_TPU_EMPTY.ipynb  uploads
```

Install Flask

```
!pip install flask-ngrok
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Requirement already satisfied: flask-ngrok in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: Flask>=0.8 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: itsdangerous<2.0,>=0.24 in /usr/local/lib/pyth
Requirement already satisfied: click<8.0,>=5.1 in /usr/local/lib/python3.7/di
Requirement already satisfied: Werkzeug<2.0,>=0.15 in /usr/local/lib/python3.
Requirement already satisfied: Jinja2<3.0,>=2.10.1 in /usr/local/lib/python3.
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /us
```

Import Libraries

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from flask import Flask, flash, redirect, render_template, request, url_for, send_
from flask_ngrok import run_with_ngrok
from tensorflow.keras.models import load_model
```

Load Trained Model

```
mnist_model = load_model('model/mnist.h5')
```

Configure Web-app

```
app = Flask(__name__)
run_with_ngrok(app)
app.secret_key = 'ACAB_таки_да_ACAB'

app.config["MNIST_BAR"] = "generated_image/mnist_vis"
app.config["IMAGES"] = "upload"

@app.route('/')
def home():
    flash("Try CNN Model Trained on MNIST-dataset for Single Digit Prediction...")
    return render_template('index.html')

@app.route('/mnist/')
def mnist_home():
    return render_template('mnist.html')

@app.route('/mnistprediction/', methods=['GET', 'POST'])
def mnist_prediction():
    if request.method == "POST":
        if not request.files['file'].filename:
            flash("No File Found")
        else:
            f =  request.files['file']
            f.save("uploads/"+f.filename)
            image_gray  = cv2.imread("uploads/"+f.filename, cv2.IMREAD_GRAYSCALE)
            img_resize = cv2.resize(image_gray,(28,28))
            image_bw = cv2.threshold(img_resize, 75, 255, cv2.THRESH_BINARY)[1]
            bitwise_not_image = cv2.bitwise_not(image_bw, mask=None)
            pred_img = np.reshape(bitwise_not_image,(1,28,28,1))/255.0
```

```
            predictions = mnist_model.predict(pred_img)
            number = int(np.argmax(predictions))
            print(number)

            plt.figure()
            y_pos = np.arange(10)
            plt.bar(y_pos, predictions[0])
            plt.savefig('generated_image/mnist_vis/'+f.filename)

            return str(number)

@app.route("/get-mnist-image/<image_name>")
def get_mnist_image(image_name):
    try:
        return send_from_directory(app.config["MNIST_BAR"], filename=image_name)
    except FileNotFoundError:
        abort(404)
```

```
# Install pyngrok
!pip install pyngrok==4.1.1

# Register, get 'your authtoken', and replace my token below:
# !ngrok authtoken 'your authtoken'
!ngrok authtoken '2FdbDL8Rak9en9IT4S3pSeMjq0I_6Ntx7LfKFyeS9qLSFAoks'
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Collecting pyngrok==4.1.1
  Downloading pyngrok-4.1.1.tar.gz (18 kB)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: PyYAML in /usr/local/lib/python3.7/dist-packag
Building wheels for collected packages: pyngrok
  Building wheel for pyngrok (setup.py) ... done
  Created wheel for pyngrok: filename=pyngrok-4.1.1-py3-none-any.whl size=15
  Stored in directory: /root/.cache/pip/wheels/b1/d9/12/045a042fee3127dc40ba
Successfully built pyngrok
Installing collected packages: pyngrok
Successfully installed pyngrok-4.1.1
Authtoken saved to configuration file: /root/.ngrok2/ngrok.yml
```

Start Web-app

After start ...

- click on the link in the row like:

Running on [your_website_at_ngrok.io]

- follow the web-user interface:
    - load the local images of single digit numbers and obtain predictions;
    - try images of different quality.

IMPORTANT: this Web-app is cloud-based and ... **some delay can be observed!**

```
app.run()
```

```
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deplo
   Use a production WSGI server instead.
 * Debug mode: off
INFO:werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Running on http://3a76-35-222-189-46.ngrok.io
 * Traffic stats available on http://127.0.0.1:4040
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:48:01] "GET / HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:48:02] "GET /static/css/main.css
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:48:02] "GET /static/css/header_fi
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:48:02] "GET /static/image/mnist-s
INFO:werkzeug:127.0.0.1 - - [03/Oct/2022 19:48:02] "GET /favicon.ico HTTP/1.1
```

Some Possible Tasks for Self-Guided Experiments:

- try to train, save and use other *.h5 model (like it was described in the previous DEMOs),
- try to use other datasets and related models,
- try to port the web-app to your local environment,
- ...

Нейронні мережі

Лекція_08

********** **********

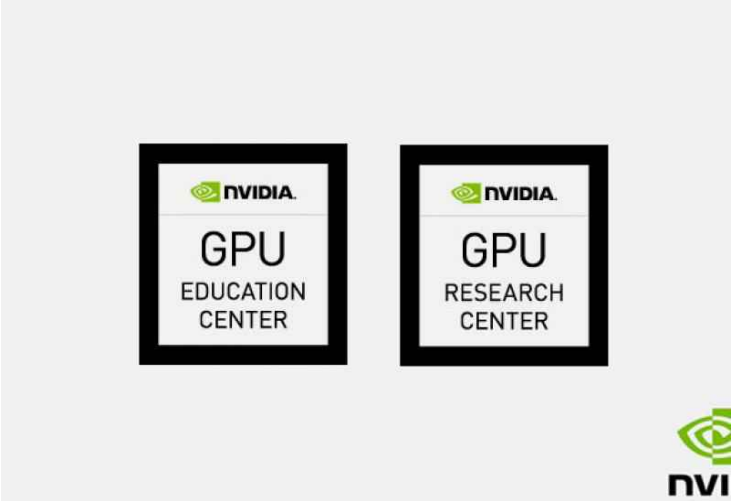
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 08- Нейронні мережі -Сучасні CNN

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1
AlexNet
https://drive.google.com/file/d/1mcviYuqLPvfg_H-hsKdkJi3wQ_RP40oT/view?usp=sharing

ДЕМО2
VGG
https://drive.google.com/file/d/1yNCtW2iIwpYDk5Tfy3d9RScl-3qLyWev/view?usp=sharing

ДЕМО4
GoogLeNet
https://drive.google.com/file/d/1rXpYBvEl_gCw4srqCMmtvU_t4Pi5qj5S/view?usp=sharing

ДЕМО6
ResNet
https://drive.google.com/file/d/1RYM83dzlKnvLKXFHkj0vupDPx_cn4C9a/view?usp=sharing

ДЕМО7
DenseNet
https://drive.google.com/file/d/1iY_jpm3JAlap6PQIr9MAn3vJCRbUzG7q/view?usp=sharing

Lecture 08. Modern CNN: DEMO 1. AlexNet

(C) partially based on 7.1. Deep Convolutional Neural Networks (AlexNet) from d2l Open Source book.

Preparatory Actions

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 8.3MB/s
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: ipython>=4.0.0; python_version >= "3.3" in /
Requirement already satisfied: nbformat>=4.2.0 in /usr/local/lib/python3.7/
Requirement already satisfied: traitlets>=4.3.1 in /usr/local/lib/python3.7
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: jupyter-client>=4.1 in /usr/local/lib/python
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-
```

```
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
```

```
import tensorflow as tf
```

Part 1.Theory - AlexNet

Although CNNs were well known in the computer vision and machine learning communities following the introduction of LeNet, they did not immediately dominate the field. Although LeNet achieved good results on early small datasets, the performance and feasibility of training CNNs on larger, more realistic datasets had yet to be established. In fact, for much of the intervening time between the early 1990s and the watershed results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines.

For computer vision, this comparison is perhaps not fair. That is although the inputs to convolutional networks consist of raw or lightly-processed (e.g., by centering) pixel values, practitioners would never feed raw pixels into traditional models. Instead, typical computer vision pipelines consisted of manually engineering feature extraction pipelines. Rather than *learn the features*, the features were *crafted*. Most of the progress came from having more clever ideas for features, and the learning algorithm was often relegated to an afterthought.

Although some neural network accelerators were available in the 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer CNNs with a large number of parameters. Moreover, datasets were still relatively small. Added to these obstacles, key tricks for training neural networks including parameter initialization heuristics, clever variants of stochastic gradient descent, non-squashing activation functions, and effective regularization techniques were still missing.

Thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:

1. Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state-of-the-art).
2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the serendipitous discoveries of lucky graduate students.
3. Feed the data through a standard set of feature extractors such as the SIFT (scale-invariant feature transform) Lowe, 2004, the SURF (speeded up robust features) Bay et al., 2006, or any number of other hand-tuned pipelines.
4. Dump the resulting representations into your favorite classifier, likely a linear model or kernel method, to train a classifier.

If you spoke to machine learning researchers, they believed that machine learning was both important and beautiful. Elegant theories proved the properties of various classifiers. The field of machine learning was thriving, rigorous, and eminently useful. However, if you spoke to a computer vision researcher, you would hear a very different story. The dirty truth of image recognition, they would tell you, is that features, not learning algorithms, drove progress. Computer vision researchers justifiably believed that a slightly bigger or cleaner dataset or a slightly improved feature-extraction pipeline mattered far more to the final accuracy than any learning algorithm.

Learning Representations

Another way to cast the state of affairs is that the most important part of the pipeline was the representation. And up until 2012 the representation was calculated mechanically. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper. SIFT Lowe, 2004, SURF Bay et al., 2006, HOG (histograms of oriented gradient) Dalal & Triggs, 2005, bags of visual words and similar feature extractors ruled the roost.

Another group of researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber, had different plans. They believed that features themselves ought to be learned. Moreover, they believed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters. In the case of an image, the lowest layers might come to detect edges, colors, and textures. Indeed, Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton proposed a new variant of a CNN, *AlexNet*, that achieved excellent performance in the 2012 ImageNet challenge. AlexNet was named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper Krizhevsky et al., 2012.

Interestingly in the lowest layers of the network, the model learned feature extractors that resembled some traditional filters. Fig. 7.1.1 is reproduced from the AlexNet paper Krizhevsky et al., 2012 and describes lower-level image descriptors.

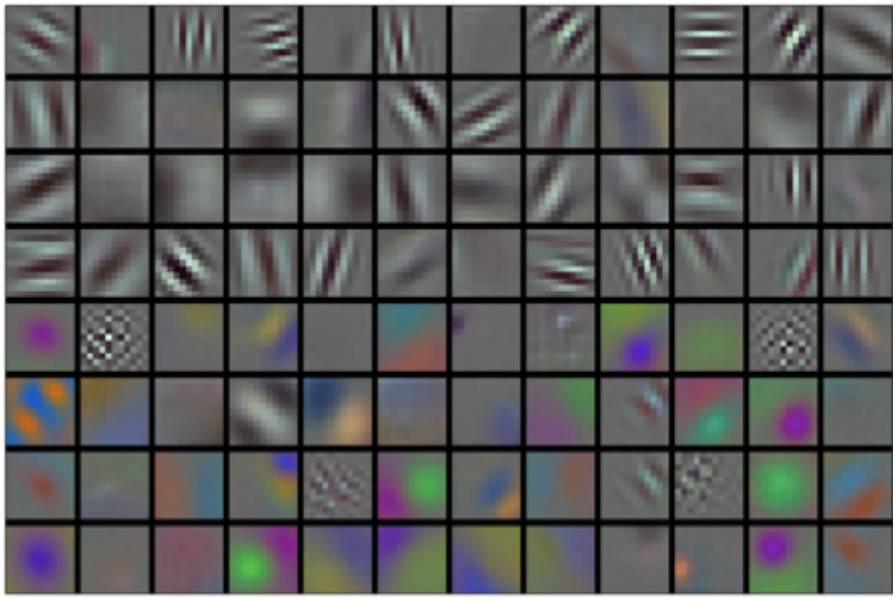


Fig. 7.1.1.

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and so on. Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees. Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents such that data belonging to different categories can be easily separated.

While the ultimate breakthrough for many-layered CNNs came in 2012, a core group of researchers had dedicated themselves to this idea, attempting to learn hierarchical representations of visual data for many years. The ultimate breakthrough in 2012 can be attributed to two key factors.

Missing Ingredient: Data

Deep models with many layers require large amounts of data in order to enter the regime where they significantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods). However, given the limited storage capacity of computers, the relative expense of sensors, and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets. Numerous papers addressed the UCI collection of datasets, many of which contained only hundreds or (a few) thousands of images captured in unnatural settings with low resolution.

In 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, 1000 each from 1000 distinct categories of objects. The researchers, led by Fei-Fei Li, who introduced this dataset leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category. This scale was unprecedented. The associated competition, dubbed the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered.

Missing Ingredient: Hardware

Deep learning models are voracious consumers of compute cycles. Training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations. This is one of the main reasons why in the 1990s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred.

Graphical processing units (GPUs) proved to be a game changer in making deep learning feasible. These chips had long been developed for accelerating graphics processing to benefit computer games. In particular, they were optimized for high throughput 4×4 matrix-vector products, which are needed for many computer graphics tasks. Fortunately, this math is strikingly similar to that required to calculate convolutional layers. Around that time, NVIDIA and ATI had begun optimizing GPUs for general computing operations, going as far as to market them as *general-purpose GPUs* (GPGPU).

To provide some intuition, consider the cores of a modern microprocessor (CPU). Each of the cores is fairly powerful running at a high clock frequency and sporting large caches (up to several megabytes of L3). Each core is well-suited to executing a wide range of instructions, with branch predictors, a deep pipeline, and other bells and whistles that enable it to run a large variety of programs. This apparent strength, however, is also its Achilles heel: general-purpose cores are very expensive to build. They require lots of chip area, a sophisticated support structure (memory interfaces, caching logic between cores, high-speed interconnects, and so on), and they are comparatively bad at any single task. Modern laptops have up to 4 cores, and even high-end servers rarely exceed 64 cores, simply because it is not cost effective.

By comparison, GPUs consist of $100 \sim 1000$ small processing elements (the details differ somewhat between NVIDIA, ATI, ARM and other chip vendors), often grouped into larger groups (NVIDIA calls them warps). While each core is relatively weak, sometimes even running at sub-1GHz clock frequency, it is the total number of such cores that makes GPUs orders of magnitude faster than CPUs. For instance, NVIDIA's recent Volta generation offers up to 120 TFlops per chip for specialized instructions (and up to 24 TFlops for more general-purpose ones), while floating point performance of CPUs has not exceeded 1 TFlop to date. The reason for why this is possible is actually quite simple: first, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4 times faster (a typical number), you can use 16 GPU cores at $1/4$ the speed, which yields $16 \times 1/4 = 4$ times the performance. Furthermore, GPU cores are much simpler (in fact, for a long time they were not even *able* to execute general-purpose code), which makes them more energy efficient. Last, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10 times as wide as many CPUs.

Missing Ingredient (the firts and last one): Deep Neaural Network

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep CNN that could run on GPU hardware. They realized that the computational bottlenecks in CNNs, convolutions and matrix multiplications, are all operations that could be parallelized in hardware. Using two NVIDIA GTX 580s with 3GB of memory, they implemented fast convolutions. The code cuda-convnet was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

AlexNet

AlexNet, which employed an 8-layer CNN, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin. This network showed, for the first time, that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision.

The architectures of AlexNet and LeNet are very similar, as Fig. 7.1.2 illustrates. Note that we provide a slightly streamlined version of AlexNet removing some of the design quirks that were needed in 2012 to make the model fit on two small GPUs.

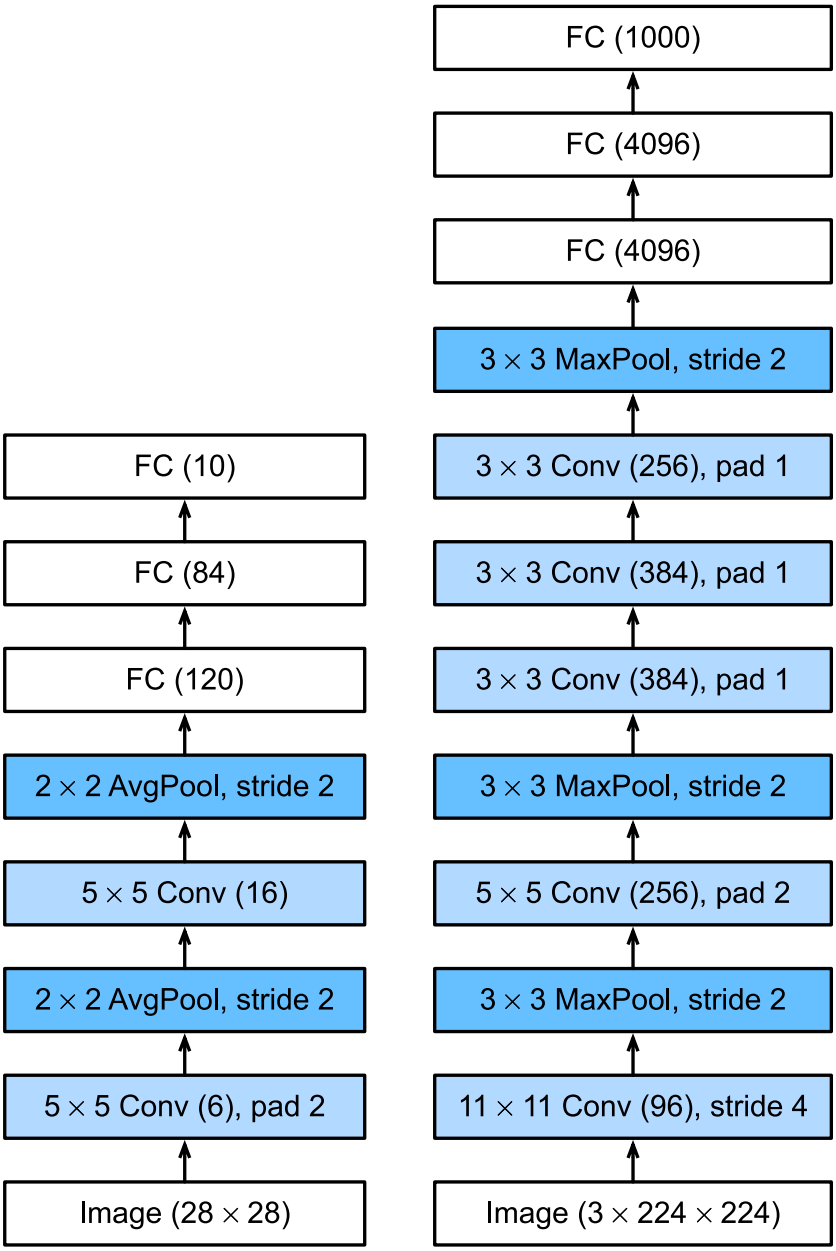


Fig. 7.1.2. From LeNet (left) to AlexNet (right).

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer. Second, AlexNet used the ReLU instead of the sigmoid as its activation function. Let us delve into the details below.

Architecture

In AlexNet's first layer, the convolution window shape is 11×11. Since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×5, followed by 3×3. In addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×3 and a stride of 2. Moreover, AlexNet has ten times more convolution channels than LeNet.

After the last convolutional layer there are two fully-connected layers with 4096 outputs. These two huge fully-connected layers produce model parameters of nearly 1 GB. Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could be responsible for storing and computing only its half of the model. Fortunately, GPU memory is comparatively abundant now, so we rarely need to break up models across GPUs these days (our version of the AlexNet model deviates from the original paper in this aspect).

Activation Functions

Besides, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found in the sigmoid activation function. On the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods. This is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that backpropagation cannot continue to update some of the model parameters. In contrast, the gradient of the ReLU activation function in the positive interval is always 1. Therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

Capacity Control and Preprocessing

AlexNet controls the model complexity of the fully-connected layer by dropout, while LeNet only uses weight decay. To augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes. This makes the model more robust and the larger sample size effectively reduces overfitting.

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 4.2MB/s
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: ipython in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.7/d
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: traitlets in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: jinja2>=2.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: nbformat>=4.4 in /usr/local/lib/python3.7/di
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: tornado>=4.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
```

```
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
```

```
import tensorflow as tf
from d2l import tensorflow as d2l

def net():
    return tf.keras.models.Sequential([
        # Here, we use a larger 11 x 11 window to capture objects. At the same
        # time, we use a stride of 4 to greatly reduce the height and width of
        # the output. Here, the number of output channels is much larger than
        # that in LeNet
        tf.keras.layers.Conv2D(filters=96, kernel_size=11, strides=4,
                               activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
        # Make the convolution window smaller, set padding to 2 for consistent
        # height and width across the input and output, and increase the
        # number of output channels
        tf.keras.layers.Conv2D(filters=256, kernel_size=5, padding='same',
                               activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
        # Use three successive convolutional layers and a smaller convolution
        # window. Except for the final convolutional layer, the number of
        # output channels is further increased. Pooling layers are not used to
        # reduce the height and width of input after the first two
        # convolutional layers
        tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding='same',
                               activation='relu'),
        tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding='same',
                               activation='relu'),
        tf.keras.layers.Conv2D(filters=256, kernel_size=3, padding='same',
                               activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
        tf.keras.layers.Flatten(),
        # Here, the number of outputs of the fully-connected layer is several
        # times larger than that in LeNet. Use the dropout layer to mitigate
        # overfitting
        tf.keras.layers.Dense(4096, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(4096, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        # Output layer. Since we are using Fashion-MNIST, the number of
        # classes is 10, instead of 1000 as in the paper
        tf.keras.layers.Dense(10)])
```

We construct a single-channel data example with both height and width of 224 to observe the output shape of each layer. It matches the AlexNet architecture in Fig. 7.1.2.

```
%%time

X = tf.random.uniform((1, 224, 224, 1))
for layer in net().layers:
```

```
    X = layer(X)
    print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

```
Conv2D output shape:     (1, 54, 54, 96)
MaxPooling2D output shape:       (1, 26, 26, 96)
Conv2D output shape:     (1, 26, 26, 256)
MaxPooling2D output shape:       (1, 12, 12, 256)
Conv2D output shape:     (1, 12, 12, 384)
Conv2D output shape:     (1, 12, 12, 384)
Conv2D output shape:     (1, 12, 12, 256)
MaxPooling2D output shape:       (1, 5, 5, 256)
Flatten output shape:    (1, 6400)
Dense output shape:      (1, 4096)
Dropout output shape:    (1, 4096)
Dense output shape:      (1, 4096)
Dropout output shape:    (1, 4096)
Dense output shape:      (1, 10)
CPU times: user 80.6 ms, sys: 9.03 ms, total: 89.7 ms
Wall time: 85.1 ms
```

Reading the Dataset

Although AlexNet is trained on ImageNet in the paper, we use Fashion-MNIST here since training an ImageNet model to convergence could take hours or days even on a modern GPU. One of the problems with applying AlexNet directly on Fashion-MNIST is that its images have lower resolution (28×28 pixels) than ImageNet images. To make things work, we upsample them to 224×224 (generally not a smart practice, but we do it here to be faithful to the AlexNet architecture). We perform this resizing with the `resize` argument in the `d2l.load_data_fashion_mnist` function.

```
%%time

batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
4423680/4422102 [==============================] - 0s 0us/step
CPU times: user 907 ms, sys: 251 ms, total: 1.16 s
Wall time: 1.34 s
```

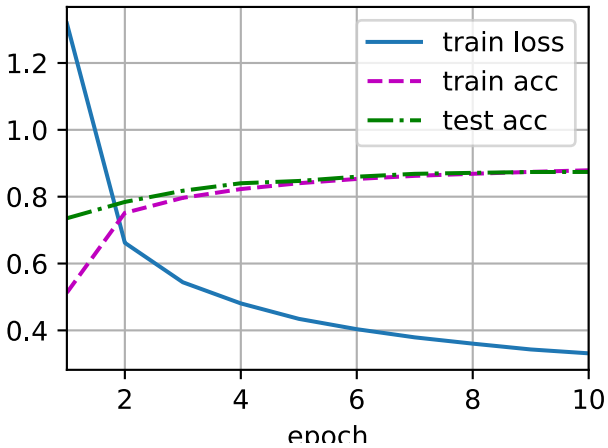
Training

Now, we can start training AlexNet. Compared with LeNet, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher

image resolution, and the more costly convolutions.

```
%%time

lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.331, train acc 0.879, test acc 0.874
773.3 examples/sec on /GPU:0
CPU times: user 9min 48s, sys: 4min 44s, total: 14min 33s
Wall time: 13min 47s
```



Summary

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale ImageNet dataset.
- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.
- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.
- Dropout, ReLU, and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

Colab paid products - Cancel contracts here

Lecture 08. Modern CNN: DEMO 2. VGG

(C) partially based on 7.2. Networks Using Blocks (VGG) from d2l Open Source book.

Preparatory Actions

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 3.4MB/s
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/pyt
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: traitlets in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-client>=4.1 in /usr/local/lib/python
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: ipython>=4.0.0; python_version >= "3.3" in /
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: nbformat>=4.2.0 in /usr/local/lib/python3.7/
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: tornado>=4.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
```

```
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
```

```
import tensorflow as tf
```

Part 1. Theory - Networks Using Blocks (VGG)

While AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks. In the following sections, we will introduce several heuristic concepts commonly used to design deep networks.

Progress in this field mirrors that in chip design where engineers went from placing transistors to logical elements to logic blocks. Similarly, the design of neural network architectures had grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, in their eponymously-named *VGG* network. It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

VGG Blocks

The basic building block of classic CNNs is a sequence of the following: (i) a convolutional layer with padding to maintain the resolution, (ii) a nonlinearity such as a ReLU, (iii) a pooling layer such as a max pooling layer. One VGG block consists of a sequence of convolutional layers, followed by a max pooling layer for spatial downsampling. In the original VGG paper Simonyan & Zisserman, 2014, the authors employed convolutions with 3×3 kernels with padding of 1 (keeping height and width) and 2×2 max pooling with stride of 2 (halving the resolution after each block). In the code below, we define a function called `vgg_block` to implement one VGG block.

The function takes two arguments corresponding to the number of convolutional layers `num_convs` and the number of output channels `num_channels`.

```
import tensorflow as tf
from d2l import tensorflow as d2l

def vgg_block(num_convs, num_channels):
    blk = tf.keras.models.Sequential()
    for _ in range(num_convs):
        blk.add(
            tf.keras.layers.Conv2D(num_channels, kernel_size=3,
                                   padding='same', activation='relu'))
    blk.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
    return blk
```

VGG Network

Like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers and the second consisting of fully-connected layers. This is depicted in Fig. 7.2.1.

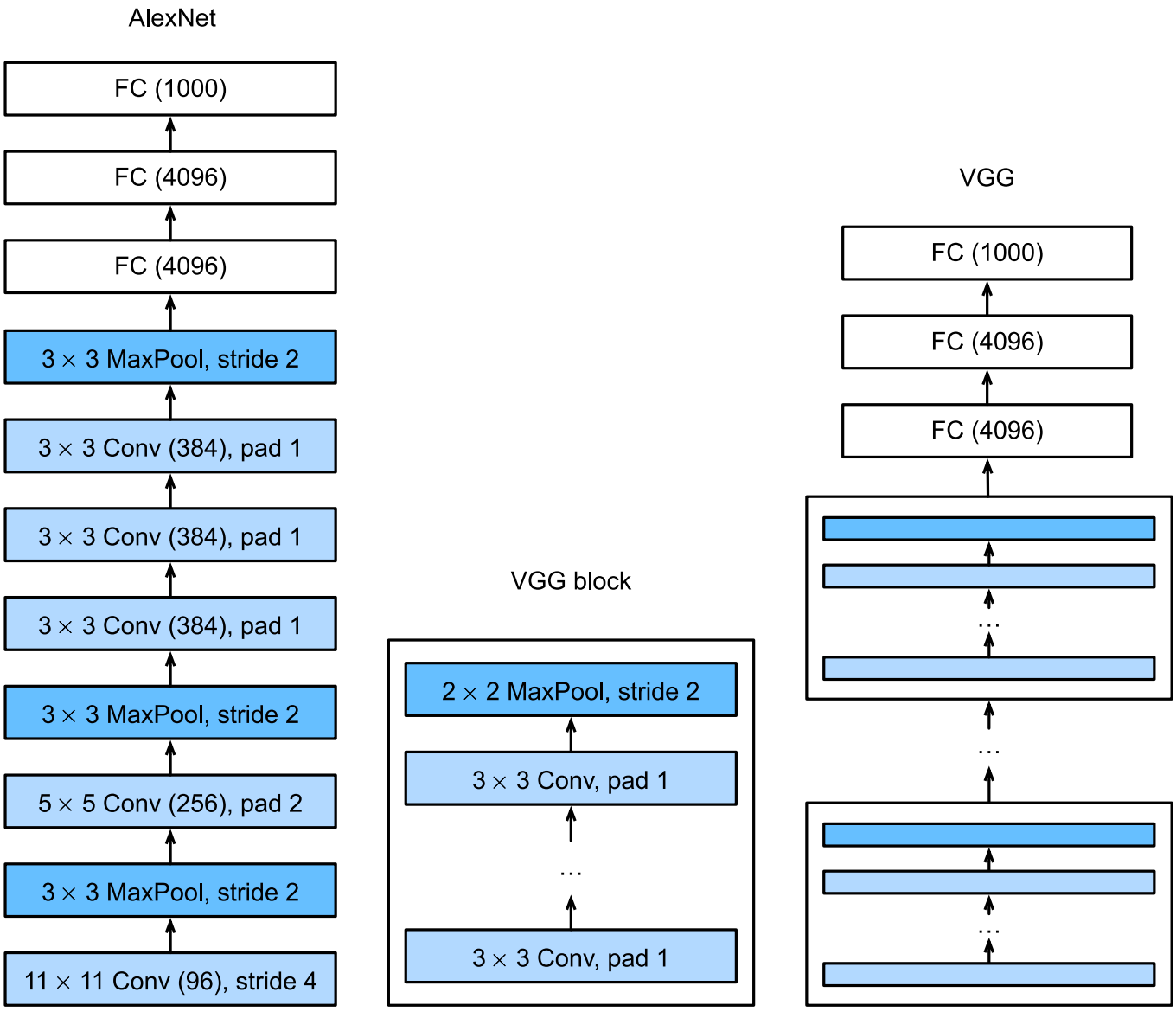


Fig. 7.2.1. From AlexNet to VGG that is designed from building blocks.

The convolutional part of the network connects several VGG blocks from Fig. 7.2.1 (also defined in the `vgg_block` function) in succession. The following variable `conv_arch` consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments required to call the `vgg_block` function. The fully-connected part of the VGG network is identical to that covered in AlexNet.

The original VGG network had 5 convolutional blocks, among which the first two have one convolutional layer each and the latter three contain two convolutional layers each. The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512. Since this network uses 8 convolutional layers and 3 fully-connected layers, it is often called VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

The following code implements VGG-11. This is a simple matter of executing a for-loop over `conv_arch`.

```
def vgg(conv_arch):
    net = tf.keras.models.Sequential()
    # The convulational part
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully-connected part
    net.add(
        tf.keras.models.Sequential([
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(4096, activation='relu'),
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(4096, activation='relu'),
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(10)]))
    return net

net = vgg(conv_arch)
```

Next, we will construct a single-channel data example with a height and width of 224 to observe the output shape of each layer.

```
X = tf.random.uniform((1, 224, 224, 1))
for blk in net.layers:
    X = blk(X)
    print(blk.__class__.__name__, 'output shape:\t', X.shape)
```

```
Sequential output shape:         (1, 112, 112, 64)
Sequential output shape:         (1, 56, 56, 128)
```

```
Sequential output shape:         (1, 28, 28, 256)
Sequential output shape:         (1, 14, 14, 512)
Sequential output shape:         (1, 7, 7, 512)
Sequential output shape:         (1, 10)
```

As you can see, we halve height and width at each block, finally reaching a height and width of 7 before flattening the representations for processing by the fully-connected part of the network.

Training

Since VGG-11 is more computationally-heavy than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
# Recall that this has to be a function that will be passed to
# `d2l.train_ch6()` so that model building/compiling need to be within
# `strategy.scope()` in order to utilize the CPU/GPU devices that we have
net = lambda: vgg(small_conv_arch)
```

Apart from using a slightly larger learning rate, the model training process is similar to that of AlexNet.

CPU training - just for fun

```
%%time
# CPU
# Wall time: 6h 45min for 0-4 epochs
# Approx. 1h 39min per epoch

lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
4423680/4422102 [==============================] - 0s 0us/step
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-8-54838e4bcf98> in <module>()
----> 1 get_ipython().run_cell_magic('time', '', '# CPU\n# Wall time: 6h
45min for 0-4 epochs\n# Approx. 1h 39min per epoch\n\nlr, num_epochs,
batch_size = 0.05, 10, 128\ntrain_iter, test_iter =
d2l.load_data_fashion_mnist(batch_size, resize=224) \nd2l.train_ch6(net,
train_iter, test_iter, num_epochs, lr, d2l.try_gpu())')

                                10 frames
<decorator-gen-53> in time(self, line, cell, local_ns)

<timed exec> in <module>()

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/execute.py in
quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     58     ctx.ensure_initialized()
     59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name,
op_name,
---> 60                                         inputs, attrs, num_outputs)
     61   except core._NotOkStatusException as e:
     62     if name is not None:
```

GPU training

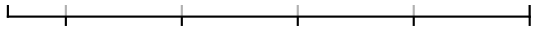
SEARCH STACK OVERFLOW

```
%%time
# GPU - when you change Runtime Type you lost local variables and you should re-ru
# Wall time: 10min for 0-4 epochs
# Approx. 2min per epoch

lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.178, train acc 0.934, test acc 0.919
440.3 examples/sec on /GPU:0
CPU times: user 14min 34s, sys: 10min 1s, total: 24min 35s
Wall time: 24min 4s
```

Part 2.Estimators + TF2-Keras Models

From TF2-Keras Models -> by Transfer Learning (TL) with ImageNet weights



Import VGG16 from TF2-Keras

```
keras_Vgg16 = tf.keras.applications.VGG16(
    input_shape=(160, 160, 3), include_top=False)
keras_Vgg16.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
58892288/58889256 [==============================] - 1s 0us/step
```

Create TL-model by adding layers to VGG16 model

```
estimator_model = tf.keras.Sequential([
    keras_Vgg16,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    #tf.keras.layers.Dense(1)
    tf.keras.layers.Dense(10) #, activation=tf.nn.softmax)
])
```

```
keras_Vgg16.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 160, 160, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 160, 160, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 80, 80, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 80, 80, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 80, 80, 128)       147584
```

```
block2_pool (MaxPooling2D)   (None, 40, 40, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 40, 40, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 20, 20, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 20, 20, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 10, 10, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

```
estimator_model.summary()
```

```
Model: "sequential_21"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 5, 5, 512)         14714688
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense_9 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_10 (Dense)             (None, 10)                2570
=================================================================
Total params: 14,848,586
Trainable params: 133,898
Non-trainable params: 14,714,688
_________________________________________________________________
```

Compile

```
estimator_model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=Tru
                metrics=['sparse_categorical_accuracy'])
```

Create Estimator

```
est_vgg16 = tf.keras.estimator.model_to_estimator(keras_model = estimator_model, m
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using the Keras model provided.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Using config: {'_model_dir': 'vgg16_tl', '_tf_random_seed': N
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
```

Data preprocessing

```
IMG_SIZE = 160
import tensorflow_datasets as tfds
import numpy as np
def preprocess(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but VGG was trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255.0)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))

# Converting Labels to one hot encoded format
#train_Y_one_hot = to_categorical(train_Y)
#test_Y_one_hot = to_categorical(test_Y)
  return image, label
```

Input function

```
def train_input_fn(batch_size):
  #data = tfds.load('cats_vs_dogs', as_supervised=True)
  data = tfds.load('fashion_mnist', as_supervised=True)
  train_data = data['train']
  train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
  return train_data
```

Training

```
%%time

# GPU
# Start: 16:34

est_vgg16.train(input_fn = lambda: train_input_fn(32), steps = 2500)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/p
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically bot
Downloading and preparing dataset fashion_mnist/3.0.1 (download: 29.45 MiB,
```

Dl Completed...: 100%	4/4 [00:05<00:00, 1.36s/ url]
Dl Size...: 100%	29/29 [00:05<00:00, 5.36 MiB/s]
Extraction completed...: 100%	4/4 [00:05<00:00, 1.33s/ file]
	60000/0 [00:52<00:00, 1161.38 examples/s]

```
Shuffling and writing examples to /root/tensorflow_datasets/fashion_mnist/3
```

92%	55419/60000 [00:00<00:00, 24185.53 examples/s]
	10000/0 [00:08<00:00, 1146.31 examples/s]

```
Shuffling and writing examples to /root/tensorflow_datasets/fashion_mnist/3
```

0%	0/10000 [00:00<?, ? examples/s]

```
Dataset fashion_mnist downloaded and prepared to /root/tensorflow_datasets/
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:4
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting from: vgg16_tl/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting from: vgg16_tl/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-started 30 variables.
INFO:tensorflow:Warm-started 30 variables.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Saving checkpoints for 0 into vgg16_tl/model.ckpt.
INFO:tensorflow:Saving checkpoints for 0 into vgg16_tl/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:loss = 3.0846012, step = 0
INFO:tensorflow:loss = 3.0846012, step = 0
INFO:tensorflow:global_step/sec: 6.86572
```

```
est_vgg16.evaluate(input_fn = lambda: train_input_fn(32), steps=50)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/trainir
  warnings.warn('`Model.state_updates` will be removed in a future version. '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-03-25T07:23:03Z
INFO:tensorflow:Starting evaluation at 2021-03-25T07:23:03Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from vgg16_tl/model.ckpt-1875
INFO:tensorflow:Restoring parameters from vgg16_tl/model.ckpt-1875
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [5/50]
```

```
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Inference Time : 7.69404s
INFO:tensorflow:Inference Time : 7.69404s
INFO:tensorflow:Finished evaluation at 2021-03-25-07:23:11
INFO:tensorflow:Finished evaluation at 2021-03-25-07:23:11
INFO:tensorflow:Saving dict for global step 1875: global_step = 1875, loss =
INFO:tensorflow:Saving dict for global step 1875: global_step = 1875, loss =
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1875: vgg16_
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1875: vgg16_
{'global_step': 1875,
 'loss': 0.42696586,
 'sparse_categorical_accuracy': 0.84375}
```

Analyze history and metrics

```
%reload_ext tensorboard
%tensorboard --logdir ./vgg16_tl
```

Summary

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.
- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.
- In their VGG paper, Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e., 3×3) were more effective than fewer layers of wider convolutions.

From TF2-Keras Models -> by Learning with Random Weights

Import VGG16 from TF2-Keras

```
keras_Vgg16 = tf.keras.applications.VGG16(
    input_shape=(160, 160, 3),
    include_top=True, weights=None, classes=10) # differences!
keras_Vgg16.trainable = True
```

Create TL-model by adding layers to VGG16 model

```
estimator_model = keras_Vgg16
```

```
keras_Vgg16.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 160, 160, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 160, 160, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 80, 80, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 80, 80, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 80, 80, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 40, 40, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 40, 40, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 20, 20, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 20, 20, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 10, 10, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 10, 10, 512)       2359808
```

```
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 12800)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              52432896
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 10)                40970
=================================================================
Total params: 83,969,866
Trainable params: 83,969,866
Non-trainable params: 0
_________________________________________________________________
```

```
estimator_model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 160, 160, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 160, 160, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 80, 80, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 80, 80, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 80, 80, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 40, 40, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 40, 40, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 20, 20, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 20, 20, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 10, 10, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
```

```
block5_pool (MaxPooling2D)   (None, 5, 5, 512)        0
_________________________________________________________________
flatten (Flatten)            (None, 12800)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              52432896
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 10)                40970
=================================================================
Total params: 83,969,866
Trainable params: 83,969,866
Non-trainable params: 0
_________________________________________________________________
```

Compile

```
estimator_model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=Tru
                metrics=['sparse_categorical_accuracy'])
```

Create Estimator

```
est_vgg16 = tf.keras.estimator.model_to_estimator(keras_model = estimator_model, m
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using default config.
INFO:tensorflow:Using the Keras model provided.
INFO:tensorflow:Using the Keras model provided.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Using config: {'_model_dir': 'vgg16_none', '_tf_random_seed':
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
INFO:tensorflow:Using config: {'_model_dir': 'vgg16_none', '_tf_random_seed':
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
```

Data preprocessing

```
IMG_SIZE = 160
import tensorflow_datasets as tfds
```

```
import numpy as np
def preprocess(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but VGG was trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255.0)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))

# Converting Labels to one hot encoded format
#train_Y_one_hot = to_categorical(train_Y)
#test_Y_one_hot = to_categorical(test_Y)
  return image, label
```

Input function

```
def train_input_fn(batch_size):
  #data = tfds.load('cats_vs_dogs', as_supervised=True)
  data = tfds.load('fashion_mnist', as_supervised=True)
  train_data = data['train']
  train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
  return train_data
```

Training

```
! rm -r /content/vgg16_random
```

```
rm: cannot remove '/content/vgg16_random': No such file or directory
```

```
%%time

# GPU
# Start: 17:00

est_vgg16.train(input_fn = lambda: train_input_fn(32), steps = 500)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:4
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting from: vgg16_none/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting from: vgg16_none/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-started 32 variables.
INFO:tensorflow:Warm-started 32 variables.
```

```
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Saving checkpoints for 0 into vgg16_none/model.ckpt.
INFO:tensorflow:Saving checkpoints for 0 into vgg16_none/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:loss = 2.3028667, step = 0
INFO:tensorflow:loss = 2.3028667, step = 0
INFO:tensorflow:global_step/sec: 2.16233
INFO:tensorflow:global_step/sec: 2.16233
INFO:tensorflow:loss = 2.3051372, step = 100 (46.257 sec)
INFO:tensorflow:loss = 2.3051372, step = 100 (46.257 sec)
INFO:tensorflow:global_step/sec: 2.20004
INFO:tensorflow:global_step/sec: 2.20004
INFO:tensorflow:loss = 2.3041759, step = 200 (45.452 sec)
INFO:tensorflow:loss = 2.3041759, step = 200 (45.452 sec)
INFO:tensorflow:global_step/sec: 2.20082
INFO:tensorflow:global_step/sec: 2.20082
INFO:tensorflow:loss = 2.3087263, step = 300 (45.439 sec)
INFO:tensorflow:loss = 2.3087263, step = 300 (45.439 sec)
INFO:tensorflow:global_step/sec: 2.19977
INFO:tensorflow:global_step/sec: 2.19977
INFO:tensorflow:loss = 2.3008556, step = 400 (45.454 sec)
INFO:tensorflow:loss = 2.3008556, step = 400 (45.454 sec)
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 500..
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 500..
INFO:tensorflow:Saving checkpoints for 500 into vgg16_none/model.ckpt.
INFO:tensorflow:Saving checkpoints for 500 into vgg16_none/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 500...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 500...
INFO:tensorflow:Loss for final step: 2.3051324.
INFO:tensorflow:Loss for final step: 2.3051324.
CPU times: user 1min 45s, sys: 1min 25s, total: 3min 10s
Wall time: 4min 1s
<tensorflow_estimator.python.estimator.estimator.EstimatorV2 at
```

```
est_vgg16.evaluate(input_fn = lambda: train_input_fn(32), steps=50)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/trainir
  warnings.warn('`Model.state_updates` will be removed in a future version. '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-03-25T07:27:24Z
INFO:tensorflow:Starting evaluation at 2021-03-25T07:27:24Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
```

```
INFO:tensorflow:Restoring parameters from vgg16_none/model.ckpt-500
INFO:tensorflow:Restoring parameters from vgg16_none/model.ckpt-500
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Inference Time : 8.05782s
INFO:tensorflow:Inference Time : 8.05782s
INFO:tensorflow:Finished evaluation at 2021-03-25-07:27:32
INFO:tensorflow:Finished evaluation at 2021-03-25-07:27:32
INFO:tensorflow:Saving dict for global step 500: global_step = 500, loss = 2.
INFO:tensorflow:Saving dict for global step 500: global_step = 500, loss = 2.
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 500: vgg16_r
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 500: vgg16_r
{'global_step': 500, 'loss': 2.3023472, 'sparse_categorical_accuracy':
0.10375}
```

Analyze history and metrics

```
%reload_ext tensorboard
%tensorboard --logdir ./vgg16_none
```

Summary

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.
- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.

- In their VGG paper, Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e., 3×3) were more effective than fewer layers of wider convolutions.

Part 3.Keras - forever! :)

```
import matplotlib.pylab as plt

# Plot Training History
def plot_history(hist, label):
  plt.figure()
  plt.ylabel("Loss (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,2])
  plt.plot(hist["loss"])
  plt.plot(hist["val_loss"])
  plt.title(label + ' - Loss')

  plt.figure()
  plt.ylabel("Accuracy (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,1])
  plt.plot(hist["accuracy"])
  plt.plot(hist["val_accuracy"])
  plt.title(label + ' - Accuracy')
```

FSL, train->full(model), image=min(75x75), weights=None

- From Scratch Learning (FSL),
- train->full(model),
- image=min(75x75),
- weights=None.

Time: 1 epoch ---> 5 min

```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
model = tf.keras.applications.VGG16(
    input_shape=(75, 75, 3), # Input size must be at least 32x32, BUT ... I tried
    include_top=True, weights=None) # differences!
```

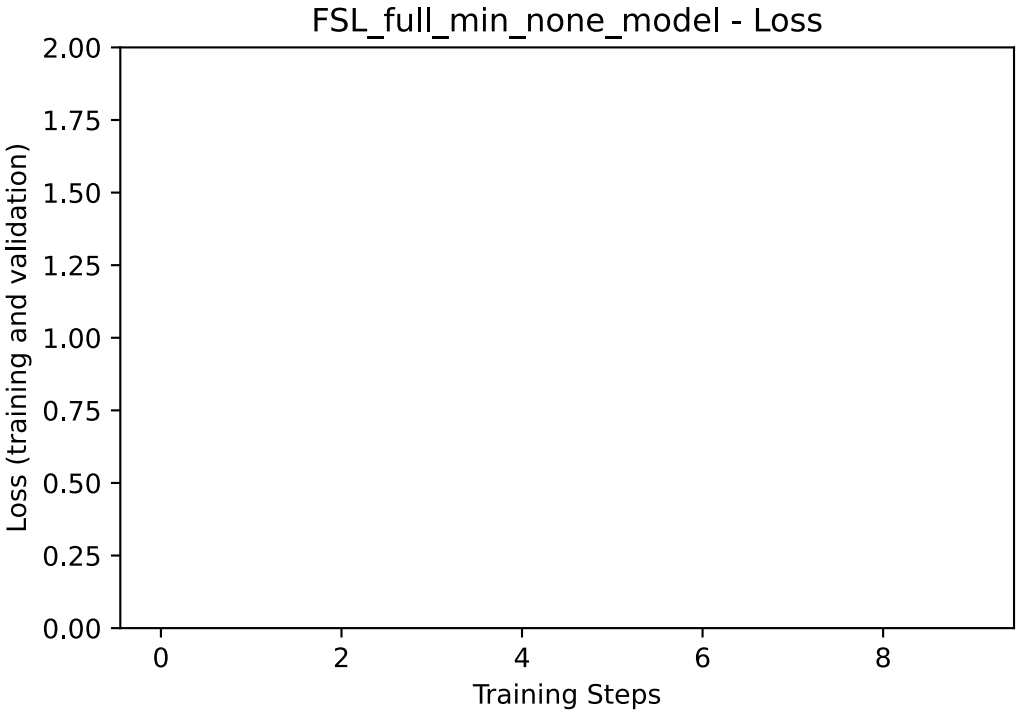
```
FSL_full_min_none_model = model

FSL_full_min_none_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()
# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
IMG_SIZE = 75 # for Inception
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
FSL_full_min_none_history = FSL_full_min_none_model.fit(train_batches, epochs=10,
                    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

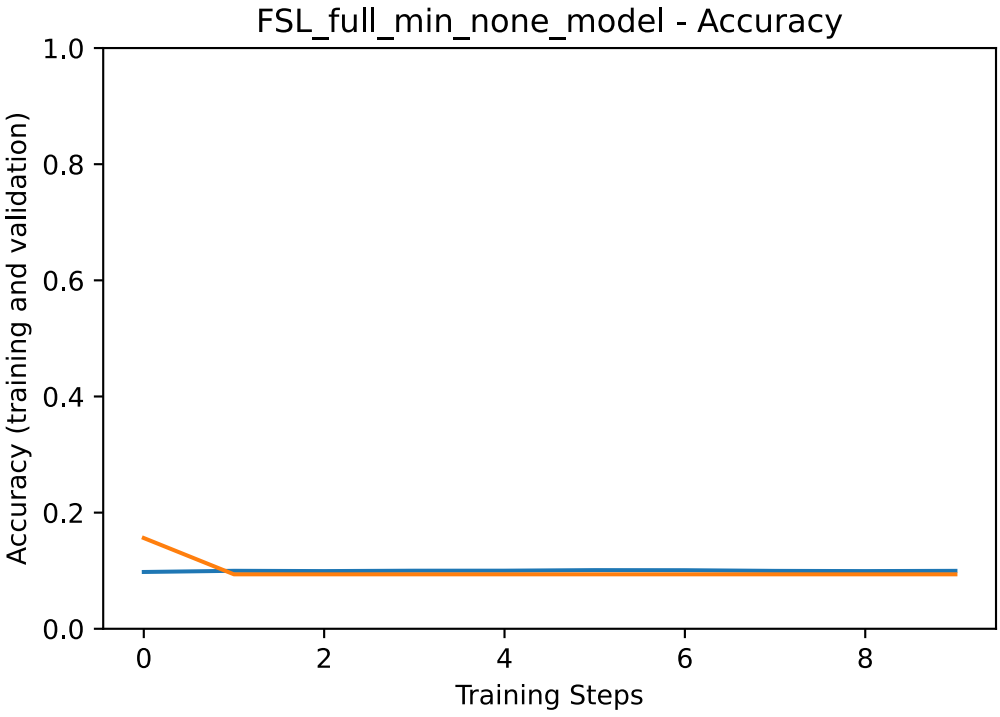
```
Epoch 1/10
1875/1875 [==============================] - 309s 163ms/step - loss: 6.5276 -
Epoch 2/10
1875/1875 [==============================] - 301s 160ms/step - loss: 2.3071 -
Epoch 3/10
1875/1875 [==============================] - 302s 161ms/step - loss: 2.3052 -
Epoch 4/10
1875/1875 [==============================] - 301s 161ms/step - loss: 2.3044 -
Epoch 5/10
1875/1875 [==============================] - 301s 161ms/step - loss: 2.3038 -
Epoch 6/10
1875/1875 [==============================] - 301s 160ms/step - loss: 2.3035 -
Epoch 7/10
1875/1875 [==============================] - 300s 160ms/step - loss: 2.3031 -
```

```
Epoch 8/10
1875/1875 [==============================] - 300s 160ms/step - loss: 2.3031 -
Epoch 9/10
1875/1875 [==============================] - 300s 160ms/step - loss: 2.3031 -
Epoch 10/10
1875/1875 [==============================] - 300s 160ms/step - loss: 2.3030 -
```

Plot History

```
plot_history(FSL_full_min_none_history.history, 'FSL_full_min_none_model')
```





TL, train->full(model), image=large(224x224), weights=ImageNet

1 epoch ---> 25 min - NOT FINISHED :)

```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 224 # for VGG: input size must be at least 224x224
model = tf.keras.applications.VGG16(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=True, weights='imagenet') # differences!

TL_full_large_imagenet_model = model

TL_full_large_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_large_imagenet_history = TL_full_large_imagenet_model.fit(
```

```
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
553467904/553467096 [==============================] - 5s 0us/step
Epoch 1/10
  34/1875 [..............................] - ETA: 26:13 - loss: 26.6716 - acc
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-41-1d97ebd8ac6d> in <module>()
     49 TL_full_large_imagenet_history = TL_full_large_imagenet_model.fit(
     50     train_batches, epochs=10,
---> 51     validation_data=validation_batches, validation_steps=1)
     52 # LOAD PHASE END #

                                 6 frames
/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/execute.py in
quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     58     ctx.ensure_initialized()
     59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name,
op_name,
---> 60                                         inputs, attrs, num_outputs)
     61   except core._NotOkStatusException as e:
     62     if name is not None:

KeyboardInterrupt:
```

Plot History

```
plot_history(TL_full_large_imagenet_history.history, 'TL_full_large_imagenet')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-39-23e8603d96e9> in <module>()
----> 1 plot_history(TL_full_large_imagenet_history.history,
'TL_full_large_imagenet')

NameError: name 'TL_full_large_imagenet_history' is not defined
```

SEARCH STACK OVERFLOW

TL, train->top(layer), image=min(32x32), weights->ImageNet

1 epoch ---> 45-38 sec

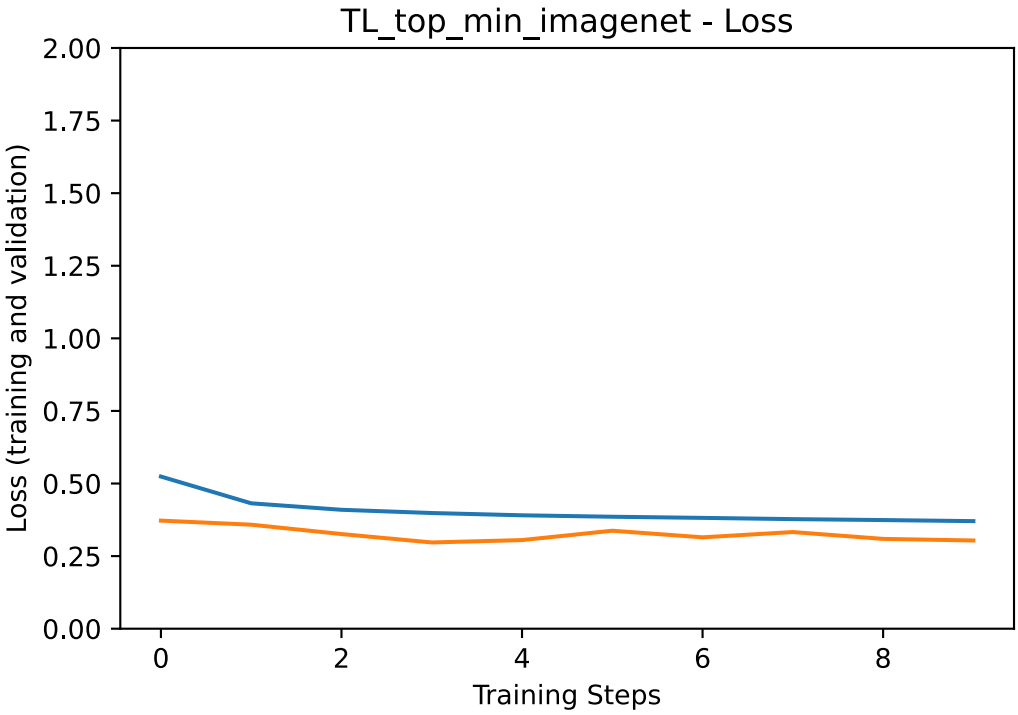
```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
```

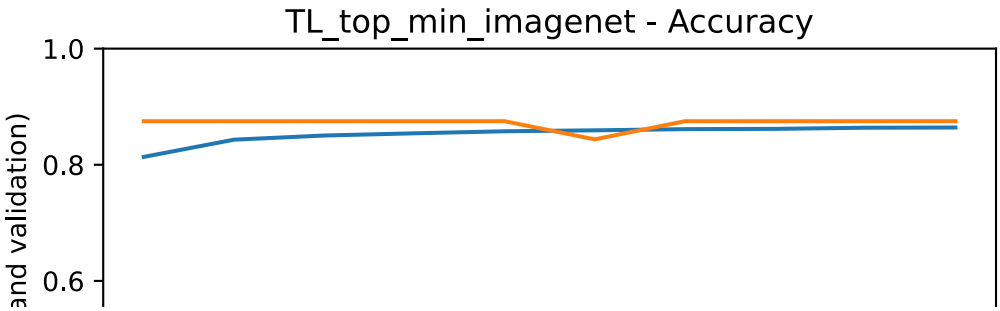
```
IMG_SIZE = 32 # for VGG: input size must be at least 32x32
model = tf.keras.applications.VGG16(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,  # train->top(layer) !
    weights='imagenet') # weights->ImageNet !

TL_top_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    #tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = False  # differences!

TL_top_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_top_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_top_min_imagenet_history = TL_top_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Model: "sequential_23"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
global_average_pooling2d_2 ( (None, 512)               0
_________________________________________________________________
dense_13 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_14 (Dense)             (None, 10)                2570
=================================================================
Total params: 14,848,586
Trainable params: 133,898
Non-trainable params: 14,714,688
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 44s 23ms/step - loss: 0.6416 - a
Epoch 2/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.4365 - a
Epoch 3/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.4095 - a
Epoch 4/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.3972 - a
Epoch 5/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.3898 - a
Epoch 6/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.3831 - a
Epoch 7/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3802 - a
Epoch 8/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.3741 - a
Epoch 9/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.3716 - a
Epoch 10/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.3685 - a
```

Plot History

```
plot_history(TL_top_min_imagenet_history.history, 'TL_top_min_imagenet')
```





TL, train->full(model), image=min(32x32), weights=ImageNet

1 epoch ---> 2 min

```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 32 # for Inception: input size must be at least 32x32
model = tf.keras.applications.VGG16(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False, weights='imagenet') # differences!

TL_full_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = True

TL_full_min_imagenet_model.compile(optimizer='adam',
```

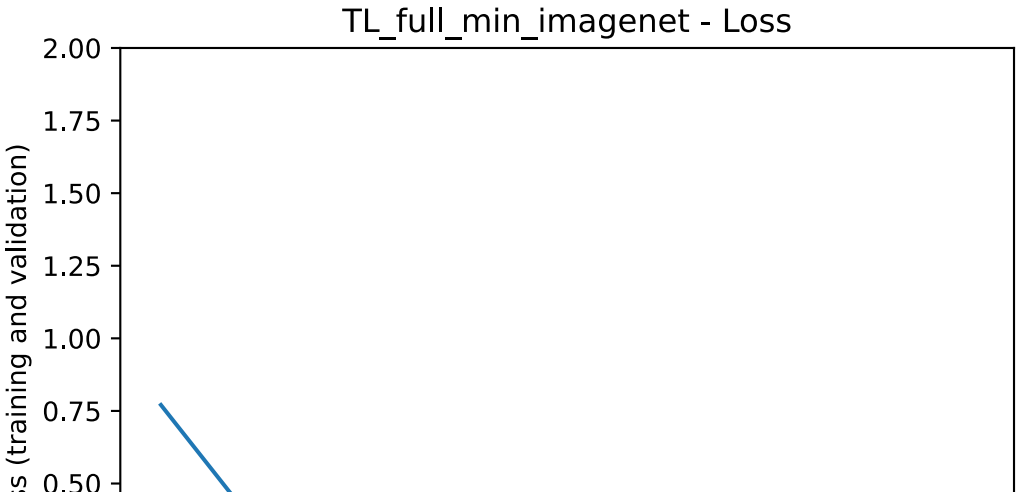
```
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_full_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_min_imagenet_history = TL_full_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Model: "sequential_24"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
global_average_pooling2d_3 ( (None, 512)               0
_________________________________________________________________
dense_15 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_6 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_16 (Dense)             (None, 10)                2570
=================================================================
Total params: 14,848,586
Trainable params: 14,848,586
```

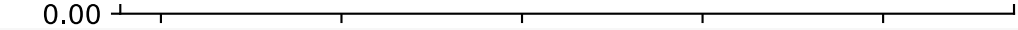
```
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 121s 64ms/step - loss: 1.1983 -
Epoch 2/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.3983 -
Epoch 3/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.3395 -
Epoch 4/10
1875/1875 [==============================] - 117s 62ms/step - loss: 0.2987 -
Epoch 5/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.2657 -
Epoch 6/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.2424 -
Epoch 7/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.2264 -
Epoch 8/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.2218 -
Epoch 9/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.1911 -
Epoch 10/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.1868 -
```

Plot History

```
plot_history(TL_full_min_imagenet_history.history, 'TL_full_min_imagenet')
```



Plot ALL training histories




```
# Plot ALL Training Histories
def plot_all_histories(hist, label):
  plt.figure()
  plt.ylabel("Loss (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
    plt.plot(h["val_loss"], label=l)
  plt.title('Loss (validation)')
  plt.legend(loc=2)

  plt.figure()
  plt.ylabel("Accuracy (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1.2])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
    plt.plot(h["val_accuracy"], label=l)
  plt.title('Accuracy (validation)')
  plt.legend(loc=3)
```

```
hist = [
        FSL_full_min_none_history.history,
        #TL_full_large_imagenet_history.history,
        TL_full_min_imagenet_history.history,
        TL_top_min_imagenet_history.history
        ]

label = [
        'FSL_full_min_none',
        #'TL_full_large_imagenet',
        'TL_full_min_imagenet',
```

Lecture 08. Modern CNN:

DEMO 4. GoogLeNet - Inception

(C) partially based on 7.4. Networks with Parallel Concatenations (GoogLeNet) from d2l Open Source book.

Preparatory Actions

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 7.2MB/s
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/pyt
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: nbformat>=4.4 in /usr/local/lib/python3.7/di
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: jinja2>=2.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.7/d
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: jupyter-client>=5.2.0 in /usr/local/lib/pyth
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-
```

```
Requirement already satisfied: ipython in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
```

Part 1.Theory - Networks with Parallel Concatenations (GoogLeNet)

In 2014, *GoogLeNet* won the ImageNet Challenge, proposing a structure that combined the strengths of NiN and paradigms of repeated blocks Szegedy et al., 2015. One focus of the paper was to address the question of which sized convolution kernels are best. After all, previous popular networks employed choices as small as 1×1 and as large as 11×11. One insight in this paper was that sometimes it can be advantageous to employ a combination of variously-sized kernels. In this section, we will introduce GoogLeNet, presenting a slightly simplified version of the original model: we omit a few ad-hoc features that were added to stabilize training but are unnecessary now with better training algorithms available.

Inception Blocks

The basic convolutional block in GoogLeNet is called an *Inception block*, likely named due to a quote from the movie *Inception* ("We need to go deeper"), which launched a viral meme.

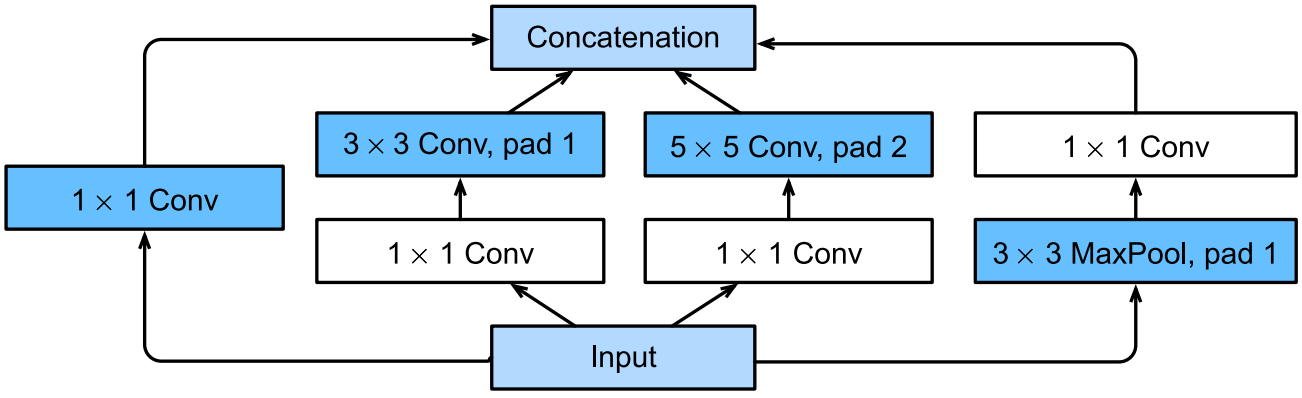


Fig. 7.4.1 Structure of the Inception block.

As depicted in Fig. 7.4.1, the inception block consists of four parallel paths. The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different spatial sizes. The middle two paths perform a 1×1 convolution on the input to reduce the number of channels, reducing the model's complexity. The fourth path uses a 3×3 maximum pooling layer, followed by a 1×1 convolutional layer to change the number of channels. The four paths all use appropriate padding to give the input and output the same height and width. Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output. The commonly-tuned hyperparameters of the Inception block are the number of output channels per layer.

```
import tensorflow as tf
from d2l import tensorflow as d2l

class Inception(tf.keras.Model):
    # `c1`--`c4` are the number of output channels for each path
    def __init__(self, c1, c2, c3, c4):
        super().__init__()
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = tf.keras.layers.Conv2D(c1, 1, activation='relu')
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = tf.keras.layers.Conv2D(c2[0], 1, activation='relu')
        self.p2_2 = tf.keras.layers.Conv2D(c2[1], 3, padding='same',
                                           activation='relu')
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = tf.keras.layers.Conv2D(c3[0], 1, activation='relu')
        self.p3_2 = tf.keras.layers.Conv2D(c3[1], 5, padding='same',
                                           activation='relu')
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = tf.keras.layers.MaxPool2D(3, 1, padding='same')
        self.p4_2 = tf.keras.layers.Conv2D(c4, 1, activation='relu')

    def call(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # Concatenate the outputs on the channel dimension
        return tf.keras.layers.Concatenate()([p1, p2, p3, p4])
```

To gain some intuition for why this network works so well, consider the combination of the filters. They explore the image in a variety of filter sizes. This means that details at different extents can be recognized efficiently by filters of different sizes. At the same time, we can allocate different amounts of parameters for different filters.

GoogLeNet Model

As shown in Fig. 7.4.2, GoogLeNet uses a stack of a total of 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduces the dimensionality. The first module is similar to AlexNet and LeNet. The stack of blocks is inherited from VGG and the global average pooling avoids a stack of fully-connected layers at the end.

Fig. 7.4.2. The GoogLeNet architecture.

We can now implement GoogLeNet piece by piece. The first module uses a 64-channel 7×7 convolutional layer.

```
def b1():
    return tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(64, 7, strides=2, padding='same',
                               activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
```

The second module uses two convolutional layers: first, a 64-channel 1×1 convolutional layer, then a 3×3 convolutional layer that triples the number of channels. This corresponds to the second path in the Inception block.

```
def b2():
    return tf.keras.Sequential([
        tf.keras.layers.Conv2D(64, 1, activation='relu'),
        tf.keras.layers.Conv2D(192, 3, padding='same', activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
```

The third module connects two complete Inception blocks in series. The number of output channels of the first Inception block is $64 + 128 + 32 + 32 = 256$, and the number-of-output-channel ratio among the four paths is $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. The second and third paths first reduce the number of input channels to $96/192 = 1/2$ and $16/192 = 1/12$, respectively, and then connect the second convolutional layer. The number of output channels of the second Inception block is increased to $128 + 192 + 96 + 64 = 480$, and the number-of-output-channel ratio among the four paths is $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. The second and third paths first reduce the number of input channels to $128/256 = 1/2$ and $32/256 = 1/8$, respectively.

```
def b3():
    return tf.keras.models.Sequential([
        Inception(64, (96, 128), (16, 32), 32),
        Inception(128, (128, 192), (32, 96), 64),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
```

The fourth module is more complicated. It connects five Inception blocks in series, and they have $192+208+48+64=512$, $160+224+64+64=512$, $128+256+64+64=512$, $112+288+64+64=528$, and $256+320+128+128=832$ output channels, respectively. The number of channels assigned to these paths is similar to that in the third module: the second path with the 3×3 convolutional layer outputs the largest number of channels, followed by the first path with only the 1×1 convolutional layer, the third path with the 5×5 convolutional layer, and the fourth path with the 3×3 maximum pooling layer. The second and third paths will first reduce the number of channels according to the ratio. These ratios are slightly different in different Inception blocks.

```
def b4():
    return tf.keras.Sequential([
        Inception(192, (96, 208), (16, 48), 64),
        Inception(160, (112, 224), (24, 64), 64),
        Inception(128, (128, 256), (24, 64), 64),
        Inception(112, (144, 288), (32, 64), 64),
        Inception(256, (160, 320), (32, 128), 128),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
```

The fifth module has two Inception blocks with $256+320+128+128=832$ and $384+384+128+128=1024$ output channels. The number of channels assigned to each path is the same as that in the third and fourth modules, but differs in specific values. It should be noted that the fifth block is followed by the output layer. This block uses the global average pooling layer to change the height and width of each channel to 1, just as in NiN. Finally, we turn the output into a two-dimensional array followed by a fully-connected layer whose number of outputs is the number of label classes.

```
def b5():
    return tf.keras.Sequential([
        Inception(256, (160, 320), (32, 128), 128),
        Inception(384, (192, 384), (48, 128), 128),
        tf.keras.layers.GlobalAvgPool2D(),
        tf.keras.layers.Flatten()])

# Recall that this has to be a function that will be passed to
# `d2l.train_ch6()` so that model building/compiling need to be within
# `strategy.scope()` in order to utilize the CPU/GPU devices that we have
def net():
    return tf.keras.Sequential([
        b1(), b2(), b3(),
        b4(), b5(), tf.keras.layers.Dense(10)])
```

The GoogLeNet model is computationally complex, so it is not as easy to modify the number of channels as in VGG. To have a reasonable training time on Fashion-MNIST, we reduce the input

height and width from 224 to 96. This simplifies the computation. The changes in the shape of the output between the various modules are demonstrated below.

```
X = tf.random.uniform(shape=(1, 96, 96, 1))
for layer in net().layers:
    X = layer(X)
    print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

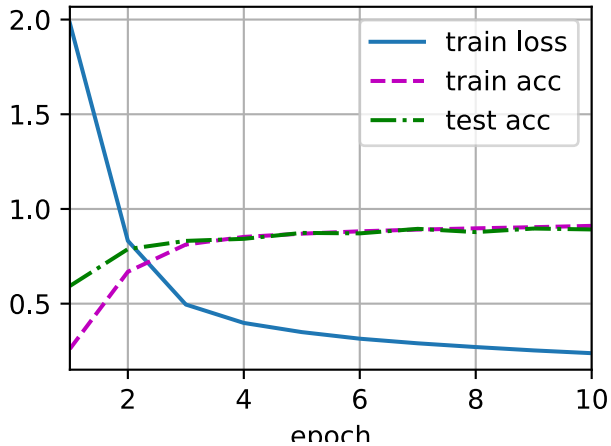
```
Sequential output shape:         (1, 24, 24, 64)
Sequential output shape:         (1, 12, 12, 192)
Sequential output shape:         (1, 6, 6, 480)
Sequential output shape:         (1, 3, 3, 832)
Sequential output shape:         (1, 1024)
Dense output shape:      (1, 10)
```

Training

As before, we train our model using the Fashion-MNIST dataset. We transform it to 96×96 pixel resolution before invoking the training procedure.

```
%%time

# 20 min for 10 epochs
# 2 min per epoch

lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.239, train acc 0.911, test acc 0.892
1155.9 examples/sec on /GPU:0
CPU times: user 7min 43s, sys: 2min 38s, total: 10min 21s
Wall time: 9min 13s
```



Summary

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers. 1×1 convolutions reduce channel dimensionality on a per-pixel level. Maximum pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series. The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet dataset.
- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

Part 2.Estimators + TF2-Keras Models

From TF2-Keras Models -> by Transfer Learning (TL) with ImageNet weights

Import InceptionV3 from TF2-Keras

```
import tensorflow as tf

keras_InceptionV3 = tf.keras.applications.InceptionV3(
    #input_shape=(160, 160, 3), include_top=False)
    input_shape=(75, 75, 3), include_top=False) # Input size must be at least 75x7
keras_InceptionV3.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
87916544/87910968 [==============================] - 0s 0us/step
```

Create TL-model by adding layers to InceptionV3 model

```
estimator_model = tf.keras.Sequential([
    keras_InceptionV3,
    #tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    #tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

```
keras_InceptionV3.summary()
```

```
Model: "inception_v3"
```

```
Layer (type)                    Output Shape         Param #     Connected
==================================================================================
input_2 (InputLayer)            [(None, 75, 75, 3)]  0
__________________________________________________________________________________
conv2d_94 (Conv2D)              (None, 37, 37, 32)   864         input_2[0]
__________________________________________________________________________________
batch_normalization_94 (BatchNo (None, 37, 37, 32)   96          conv2d_94[
__________________________________________________________________________________
activation_94 (Activation)      (None, 37, 37, 32)   0           batch_norm
__________________________________________________________________________________
conv2d_95 (Conv2D)              (None, 35, 35, 32)   9216        activation
__________________________________________________________________________________
batch_normalization_95 (BatchNo (None, 35, 35, 32)   96          conv2d_95[
__________________________________________________________________________________
activation_95 (Activation)      (None, 35, 35, 32)   0           batch_norm
__________________________________________________________________________________
conv2d_96 (Conv2D)              (None, 35, 35, 64)   18432       activation
__________________________________________________________________________________
batch_normalization_96 (BatchNo (None, 35, 35, 64)   192         conv2d_96[
__________________________________________________________________________________
activation_96 (Activation)      (None, 35, 35, 64)   0           batch_norm
__________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 17, 17, 64)   0           activation
__________________________________________________________________________________
conv2d_97 (Conv2D)              (None, 17, 17, 80)   5120        max_poolin
__________________________________________________________________________________
batch_normalization_97 (BatchNo (None, 17, 17, 80)   240         conv2d_97[
__________________________________________________________________________________
activation_97 (Activation)      (None, 17, 17, 80)   0           batch_norm
__________________________________________________________________________________
conv2d_98 (Conv2D)              (None, 15, 15, 192)  138240      activation
__________________________________________________________________________________
batch_normalization_98 (BatchNo (None, 15, 15, 192)  576         conv2d_98[
__________________________________________________________________________________
activation_98 (Activation)      (None, 15, 15, 192)  0           batch_norm
__________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D)  (None, 7, 7, 192)    0           activation
__________________________________________________________________________________
conv2d_102 (Conv2D)             (None, 7, 7, 64)     12288       max_poolin
__________________________________________________________________________________
batch_normalization_102 (BatchN (None, 7, 7, 64)     192         conv2d_102
__________________________________________________________________________________
activation_102 (Activation)     (None, 7, 7, 64)     0           batch_norm
__________________________________________________________________________________
conv2d_100 (Conv2D)             (None, 7, 7, 48)     9216        max_poolin
__________________________________________________________________________________
conv2d_103 (Conv2D)             (None, 7, 7, 96)     55296       activation
__________________________________________________________________________________
batch_normalization_100 (BatchN (None, 7, 7, 48)     144         conv2d_100
__________________________________________________________________________________
batch_normalization_103 (BatchN (None, 7, 7, 96)     288         conv2d_103
__________________________________________________________________________________
activation_100 (Activation)     (None, 7, 7, 48)     0           batch_norm
__________________________________________________________________________________
activation_103 (Activation)     (None, 7, 7, 96)     0           batch_norm
```

```
estimator_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 1, 1, 2048)        21802784
_________________________________________________________________
dense_2 (Dense)              (None, 1, 1, 256)         524544
_________________________________________________________________
dense_3 (Dense)              (None, 1, 1, 10)          2570
=================================================================
Total params: 22,329,898
Trainable params: 527,114
Non-trainable params: 21,802,784
_________________________________________________________________
```

Compile

```
estimator_model.compile(
  optimizer='adam',
  #optimizer=tf.keras.optimizers.SGD(lr=0.5, momentum=0.9),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  metrics=['sparse_categorical_accuracy'])
```

Create Estimator

```
est_InceptionV3 = tf.keras.estimator.model_to_estimator(keras_model = estimator_mo
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using default config.
INFO:tensorflow:Using the Keras model provided.
INFO:tensorflow:Using the Keras model provided.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Using config: {'_model_dir': 'InceptionV3_tl', '_tf_random_se
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
INFO:tensorflow:Using config: {'_model_dir': 'InceptionV3_tl', '_tf_random_se
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
```

Data preprocessing

```
#IMG_SIZE = 160 # for VGG
IMG_SIZE = 75 # for Inception
import tensorflow_datasets as tfds
import numpy as np
def preprocess(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but VGG was trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255.0)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))

# Converting Labels to one hot encoded format
#train_Y_one_hot = to_categorical(train_Y)
#test_Y_one_hot = to_categorical(test_Y)
  return image, label
```

Input function

```
def train_input_fn(batch_size):
  data = tfds.load('fashion_mnist', as_supervised=True)
  #data = tfds.load('cifar10', as_supervised=True)
  train_data = data['train']
  train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
  return train_data
```

Training

```
# For debugging only
# ! rm -r /content/InceptionV3_tl
```

```
%%time

# GPU
# Start: 18:23

est_InceptionV3.train(input_fn = lambda: train_input_fn(32), steps = 60000)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:4
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting from: InceptionV3_tl/keras/keras_model.ckpt
```

```
INFO:tensorflow:Warm-starting from: InceptionV3_tl/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-started 192 variables.
INFO:tensorflow:Warm-started 192 variables.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Saving checkpoints for 0 into InceptionV3_tl/model.ckpt.
INFO:tensorflow:Saving checkpoints for 0 into InceptionV3_tl/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:loss = 2.3001285, step = 0
INFO:tensorflow:loss = 2.3001285, step = 0
INFO:tensorflow:global_step/sec: 39.7866
INFO:tensorflow:global_step/sec: 39.7866
INFO:tensorflow:loss = 2.275893, step = 100 (2.520 sec)
INFO:tensorflow:loss = 2.275893, step = 100 (2.520 sec)
INFO:tensorflow:global_step/sec: 48.9332
INFO:tensorflow:global_step/sec: 48.9332
INFO:tensorflow:loss = 2.3070817, step = 200 (2.043 sec)
INFO:tensorflow:loss = 2.3070817, step = 200 (2.043 sec)
INFO:tensorflow:global_step/sec: 49.4098
INFO:tensorflow:global_step/sec: 49.4098
INFO:tensorflow:loss = 2.2752552, step = 300 (2.025 sec)
INFO:tensorflow:loss = 2.2752552, step = 300 (2.025 sec)
INFO:tensorflow:global_step/sec: 49.1186
INFO:tensorflow:global_step/sec: 49.1186
INFO:tensorflow:loss = 2.266654, step = 400 (2.035 sec)
INFO:tensorflow:loss = 2.266654, step = 400 (2.035 sec)
INFO:tensorflow:global_step/sec: 48.9999
INFO:tensorflow:global_step/sec: 48.9999
INFO:tensorflow:loss = 2.3189507, step = 500 (2.037 sec)
INFO:tensorflow:loss = 2.3189507, step = 500 (2.037 sec)
INFO:tensorflow:global_step/sec: 48.9776
INFO:tensorflow:global_step/sec: 48.9776
INFO:tensorflow:loss = 2.29397, step = 600 (2.046 sec)
INFO:tensorflow:loss = 2.29397, step = 600 (2.046 sec)
INFO:tensorflow:global_step/sec: 49.055
INFO:tensorflow:global_step/sec: 49.055
INFO:tensorflow:loss = 2.335431, step = 700 (2.039 sec)
```

```
est_InceptionV3.evaluate(input_fn = lambda: train_input_fn(32), steps=50)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/trainir
  warnings.warn('`Model.state_updates` will be removed in a future version. '
INFO:tensorflow:Done calling model_fn.
```

```
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-03-24T17:55:05Z
INFO:tensorflow:Starting evaluation at 2021-03-24T17:55:05Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from InceptionV3_tl/model.ckpt-1875
INFO:tensorflow:Restoring parameters from InceptionV3_tl/model.ckpt-1875
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Inference Time : 2.73915s
INFO:tensorflow:Inference Time : 2.73915s
INFO:tensorflow:Finished evaluation at 2021-03-24-17:55:08
INFO:tensorflow:Finished evaluation at 2021-03-24-17:55:08
INFO:tensorflow:Saving dict for global step 1875: global_step = 1875, loss =
INFO:tensorflow:Saving dict for global step 1875: global_step = 1875, loss =
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1875: Incept
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1875: Incept
{'global_step': 1875,
 'loss': 2.2620971,
 'sparse_categorical_accuracy': 0.11066406}
```

Analyze history and metrics

```
%reload_ext tensorboard
%tensorboard --logdir ./InceptionV3_tl
```

From TF2-Keras Models -> by Learning with Random Weights

Import InceptionV3 from TF2-Keras

```
import tensorflow as tf

keras_InceptionV3 = tf.keras.applications.InceptionV3(
    #input_shape=(160, 160, 3), include_top=False)
    input_shape=(75, 75, 3), # Input size must be at least 75x75
    include_top=True, weights=None, classes=10) # differences!
keras_InceptionV3.trainable = True
```

Create TL-model by adding layers to InceptionV3 model

```
none_estimator_model = keras_InceptionV3
```

```
keras_InceptionV3.summary()
```

```
Model: "inception_v3"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected
==================================================================================================
input_1 (InputLayer)            [(None, 75, 75, 3)]  0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 37, 37, 32)   864         input_1[0]
__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 37, 37, 32)   96          conv2d[0][
__________________________________________________________________________________________________
activation (Activation)         (None, 37, 37, 32)   0           batch_norm
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 35, 35, 32)   9216        activation
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 35, 35, 32)   96          conv2d_1[0
__________________________________________________________________________________________________
activation_1 (Activation)       (None, 35, 35, 32)   0           batch_norm
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 35, 35, 64)   18432       activation
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 35, 35, 64)   192         conv2d_2[0
__________________________________________________________________________________________________
activation_2 (Activation)       (None, 35, 35, 64)   0           batch_norm
__________________________________________________________________________________________________
max_pooling2d (MaxPooling2D)    (None, 17, 17, 64)   0           activation
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 17, 17, 80)   5120        max_poolin
__________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 17, 17, 80)   240         conv2d_3[0
__________________________________________________________________________________________________
activation_3 (Activation)       (None, 17, 17, 80)   0           batch_norm
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 15, 15, 192)  138240      activation
__________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 15, 15, 192)  576         conv2d_4[0
__________________________________________________________________________________________________
activation_4 (Activation)       (None, 15, 15, 192)  0           batch_norm
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 7, 7, 192)    0           activation
```

```
conv2d_8 (Conv2D)               (None, 7, 7, 64)     12288       max_poolin
batch_normalization_8 (BatchNor (None, 7, 7, 64)     192         conv2d_8[0
activation_8 (Activation)       (None, 7, 7, 64)     0           batch_norm
conv2d_6 (Conv2D)               (None, 7, 7, 48)     9216        max_poolin
conv2d_9 (Conv2D)               (None, 7, 7, 96)     55296       activation
batch_normalization_6 (BatchNor (None, 7, 7, 48)     144         conv2d_6[0
batch_normalization_9 (BatchNor (None, 7, 7, 96)     288         conv2d_9[0
activation_6 (Activation)       (None, 7, 7, 48)     0           batch_norm
activation_9 (Activation)       (None, 7, 7, 96)     0           batch_norm
```

```
none_estimator_model.summary()
```

```
Model: "inception_v3"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected
==================================================================================================
input_1 (InputLayer)            [(None, 75, 75, 3)]  0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 37, 37, 32)   864         input_1[0]
__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 37, 37, 32)   96          conv2d[0][
__________________________________________________________________________________________________
activation (Activation)         (None, 37, 37, 32)   0           batch_norm
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 35, 35, 32)   9216        activation
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 35, 35, 32)   96          conv2d_1[0
__________________________________________________________________________________________________
activation_1 (Activation)       (None, 35, 35, 32)   0           batch_norm
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 35, 35, 64)   18432       activation
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 35, 35, 64)   192         conv2d_2[0
__________________________________________________________________________________________________
activation_2 (Activation)       (None, 35, 35, 64)   0           batch_norm
__________________________________________________________________________________________________
max_pooling2d (MaxPooling2D)    (None, 17, 17, 64)   0           activation
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 17, 17, 80)   5120        max_poolin
__________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 17, 17, 80)   240         conv2d_3[0
__________________________________________________________________________________________________
activation_3 (Activation)       (None, 17, 17, 80)   0           batch_norm
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 15, 15, 192)  138240      activation
__________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 15, 15, 192)  576         conv2d_4[0
__________________________________________________________________________________________________
activation_4 (Activation)       (None, 15, 15, 192)  0           batch_norm
```

```
max_pooling2d_1 (MaxPooling2D)  (None, 7, 7, 192)    0           activation
conv2d_8 (Conv2D)               (None, 7, 7, 64)     12288       max_poolin
batch_normalization_8 (BatchNor (None, 7, 7, 64)     192         conv2d_8[0
activation_8 (Activation)       (None, 7, 7, 64)     0           batch_norm
conv2d_6 (Conv2D)               (None, 7, 7, 48)     9216        max_poolin
conv2d_9 (Conv2D)               (None, 7, 7, 96)     55296       activation
batch_normalization_6 (BatchNor (None, 7, 7, 48)     144         conv2d_6[0
batch_normalization_9 (BatchNor (None, 7, 7, 96)     288         conv2d_9[0
activation_6 (Activation)       (None, 7, 7, 48)     0           batch_norm
```

Compile

```
none_estimator_model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=Tru
                metrics=['sparse_categorical_accuracy'])
```

Create Estimator

```
none_est_InceptionV3 = tf.keras.estimator.model_to_estimator(keras_model = none_es
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using the Keras model provided.
INFO:tensorflow:Using config: {'_model_dir': 'InceptionV3_none', '_tf_random_
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
```

Data preprocessing

```
#IMG_SIZE = 160 # for VGG
IMG_SIZE = 75 # for Inception

import tensorflow_datasets as tfds
import numpy as np
def preprocess(image, label):
  ###
```

```
  # for fashion_mnist (they are in greyscale),
  # but VGG was trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255.0)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))

# Converting Labels to one hot encoded format
#train_Y_one_hot = to_categorical(train_Y)
#test_Y_one_hot = to_categorical(test_Y)
  return image, label
```

Input function

```
def train_input_fn(batch_size):
  #data = tfds.load('cats_vs_dogs', as_supervised=True)
  data = tfds.load('fashion_mnist', as_supervised=True)
  train_data = data['train']
  train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
  return train_data
```

Training

```
%%time

# GPU
# Start: 17:00

none_est_InceptionV3.train(input_fn = lambda: train_input_fn(32), steps = 50000)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/p
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically bot
Downloading and preparing dataset fashion_mnist/3.0.1 (download: 29.45 MiB,
```

Dl Completed...: 100%	4/4 [1:53:20<00:00, 1700.07s/ url]
Dl Size...: 100%	29/29 [1:53:20<00:00, 234.49s/ MiB]
Extraction completed...: 100%	4/4 [00:02<00:00, 1.87 file/s]
	60000/0 [00:41<00:00, 1461.15 examples/s]

```
Shuffling and writing examples to /root/tensorflow_datasets/fashion_mnist/3
```

74%	44385/60000 [00:00<00:00, 78328.53 examples/s]
	10000/0 [00:06<00:00, 1437.23 examples/s]

```
Shuffling and writing examples to /root/tensorflow_datasets/fashion_mnist/3
```

0%	0/10000 [00:00<?, ? examples/s]

```
Dataset fashion_mnist downloaded and prepared to /root/tensorflow_datasets/
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:4
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting with WarmStartSettings: WarmStartSettings(ckp
INFO:tensorflow:Warm-starting from: InceptionV3_none/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting from: InceptionV3_none/keras/keras_model.ckpt
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-starting variables only in TRAINABLE_VARIABLES.
INFO:tensorflow:Warm-started 190 variables.
INFO:tensorflow:Warm-started 190 variables.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 0...
INFO:tensorflow:Saving checkpoints for 0 into InceptionV3_none/model.ckpt.
INFO:tensorflow:Saving checkpoints for 0 into InceptionV3_none/model.ckpt.
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 0...
INFO:tensorflow:loss = 2.848578, step = 0
INFO:tensorflow:loss = 2.848578, step = 0
INFO:tensorflow:global_step/sec: 13.6576
```

```
none_est_InceptionV3.evaluate(input_fn = lambda: train_input_fn(32), steps=50)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/trainir
  warnings.warn('`Model.state_updates` will be removed in a future version. '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-03-24T17:30:07Z
INFO:tensorflow:Starting evaluation at 2021-03-24T17:30:07Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from InceptionV3_none/model.ckpt-1875
INFO:tensorflow:Restoring parameters from InceptionV3_none/model.ckpt-1875
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [5/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [10/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [15/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [20/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [25/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [30/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [35/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [40/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [45/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Evaluation [50/50]
INFO:tensorflow:Inference Time : 2.65367s
INFO:tensorflow:Inference Time : 2.65367s
INFO:tensorflow:Finished evaluation at 2021-03-24-17:30:10
INFO:tensorflow:Finished evaluation at 2021-03-24-17:30:10
INFO:tensorflow:Saving dict for global step 1875: global_step = 1875, loss =
INFO:tensorflow:Saving dict for global step 1875: global_step = 1875, loss =
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1875: Incept
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1875: Incept
{'global_step': 1875,
 'loss': 0.4322045,
 'sparse_categorical_accuracy': 0.85125}
```

Analyze history and metrics

```
%reload_ext tensorboard
%tensorboard --logdir ./InceptionV3_none
```

```

```

From TF2-Keras Models -> Full Model Re-Training with ImageNet Weights

Import InceptionV3 from TF2-Keras

```
import tensorflow as tf

keras_InceptionV3 = tf.keras.applications.InceptionV3(
    #input_shape=(160, 160, 3), include_top=False)
    input_shape=(75, 75, 3), # Input size must be at least 75x75
    include_top=False, weights='imagenet') # differences!
keras_InceptionV3.trainable = True
```

Create TL-model by adding layers to InceptionV3 model

```
imagenet_estimator_model = tf.keras.Sequential([
    keras_InceptionV3,
    #tf.keras.layers.GlobalAveragePooling2D(),
    #tf.keras.layers.Dense(256),
    #tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
imagenet_estimator_model.trainable = True
```

```
keras_InceptionV3.summary()
```

```
Model: "inception_v3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_3 (InputLayer)         [(None, 75, 75, 3)]       0         
_________________________________________________________________
```

```
conv2d_188 (Conv2D)             (None, 37, 37, 32)   864         input_3[0]
__________________________________________________________________________
batch_normalization_188 (BatchN (None, 37, 37, 32)   96          conv2d_188
__________________________________________________________________________
activation_188 (Activation)     (None, 37, 37, 32)   0           batch_norm
__________________________________________________________________________
conv2d_189 (Conv2D)             (None, 35, 35, 32)   9216        activation
__________________________________________________________________________
batch_normalization_189 (BatchN (None, 35, 35, 32)   96          conv2d_189
__________________________________________________________________________
activation_189 (Activation)     (None, 35, 35, 32)   0           batch_norm
__________________________________________________________________________
conv2d_190 (Conv2D)             (None, 35, 35, 64)   18432       activation
__________________________________________________________________________
batch_normalization_190 (BatchN (None, 35, 35, 64)   192         conv2d_190
__________________________________________________________________________
activation_190 (Activation)     (None, 35, 35, 64)   0           batch_norm
__________________________________________________________________________
max_pooling2d_8 (MaxPooling2D)  (None, 17, 17, 64)   0           activation
__________________________________________________________________________
conv2d_191 (Conv2D)             (None, 17, 17, 80)   5120        max_poolin
__________________________________________________________________________
batch_normalization_191 (BatchN (None, 17, 17, 80)   240         conv2d_191
__________________________________________________________________________
activation_191 (Activation)     (None, 17, 17, 80)   0           batch_norm
__________________________________________________________________________
conv2d_192 (Conv2D)             (None, 15, 15, 192)  138240      activation
__________________________________________________________________________
batch_normalization_192 (BatchN (None, 15, 15, 192)  576         conv2d_192
__________________________________________________________________________
activation_192 (Activation)     (None, 15, 15, 192)  0           batch_norm
__________________________________________________________________________
max_pooling2d_9 (MaxPooling2D)  (None, 7, 7, 192)    0           activation
__________________________________________________________________________
conv2d_196 (Conv2D)             (None, 7, 7, 64)     12288       max_poolin
__________________________________________________________________________
batch_normalization_196 (BatchN (None, 7, 7, 64)     192         conv2d_196
__________________________________________________________________________
activation_196 (Activation)     (None, 7, 7, 64)     0           batch_norm
__________________________________________________________________________
conv2d_194 (Conv2D)             (None, 7, 7, 48)     9216        max_poolin
__________________________________________________________________________
conv2d_197 (Conv2D)             (None, 7, 7, 96)     55296       activation
__________________________________________________________________________
batch_normalization_194 (BatchN (None, 7, 7, 48)     144         conv2d_194
__________________________________________________________________________
batch_normalization_197 (BatchN (None, 7, 7, 96)     288         conv2d_197
__________________________________________________________________________
activation_194 (Activation)     (None, 7, 7, 48)     0           batch_norm
__________________________________________________________________________
activation_197 (Activation)     (None, 7, 7, 96)     0           batch_norm
```

```
imagenet_estimator_model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 1, 1, 2048)        21802784
```

```
_________________________________________________________________
dense_4 (Dense)              (None, 1, 1, 10)          20490     
=================================================================
Total params: 21,823,274
Trainable params: 21,788,842
Non-trainable params: 34,432
_________________________________________________________________
```

Compile

```
imagenet_estimator_model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=Tru
                metrics=['sparse_categorical_accuracy'])
```

Create Estimator

```
imagenet_est_InceptionV3 = tf.keras.estimator.model_to_estimator(keras_model = ima
                                                                 model_dir = 'Ince
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using default config.
INFO:tensorflow:Using the Keras model provided.
INFO:tensorflow:Using the Keras model provided.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Using config: {'_model_dir': 'InceptionV3_imagenet', '_tf_rar
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
INFO:tensorflow:Using config: {'_model_dir': 'InceptionV3_imagenet', '_tf_rar
graph_options {
  rewrite_options {
    meta_optimizer_iterations: ONE
  }
}
, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_s
```

Data preprocessing

```
#IMG_SIZE = 160 # for VGG
IMG_SIZE = 75 # for Inception

import tensorflow_datasets as tfds
import numpy as np
def preprocess(image, label):
  ###
```

```
  # for fashion_mnist (they are in greyscale),
  # but VGG was trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255.0)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))

# Converting Labels to one hot encoded format
#train_Y_one_hot = to_categorical(train_Y)
#test_Y_one_hot = to_categorical(test_Y)
  return image, label
```

Input function

```
def train_input_fn(batch_size):
  #data = tfds.load('cats_vs_dogs', as_supervised=True)
  data = tfds.load('fashion_mnist', as_supervised=True)
  train_data = data['train']
  train_data = train_data.map(preprocess).shuffle(500).batch(batch_size)
  return train_data
```

Training

```
early_stop = tf.contrib.estimator.stop_if_no_decrease_hook(estimator, 'loss', 1000

train_spec = tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=2000, hooks

eval_spec = tf.estimator.EvalSpec(input_fn=eval_input_fn, steps=1, exporters=expor

exporter = tf.estimator.BestExporter(
        name="best_exporter",
        serving_input_receiver_fn=serving_fn,
        exports_to_keep=5
    )

tf.estimator.train_and_evaluate(
            estimator,
            train_spec=train_spec,
            eval_spec=eval_spec)
```

```
%%time

# GPU
# Start: 17:00

imagenet_est_InceptionV3.train(input_fn = lambda: train_input_fn(32), max_steps =
```

```
    INFO:tensorflow:Calling model_fn.
    INFO:tensorflow:Calling model_fn.
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from InceptionV3_imagenet/model.ckpt-375
INFO:tensorflow:Restoring parameters from InceptionV3_imagenet/model.ckpt-375
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 3750...
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 3750...
INFO:tensorflow:Saving checkpoints for 3750 into InceptionV3_imagenet/model.c
INFO:tensorflow:Saving checkpoints for 3750 into InceptionV3_imagenet/model.c
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 3750...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 3750...
INFO:tensorflow:loss = 0.23958135, step = 3750
INFO:tensorflow:loss = 0.23958135, step = 3750
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 3751...
INFO:tensorflow:Calling checkpoint listeners before saving checkpoint 3751...
INFO:tensorflow:Saving checkpoints for 3751 into InceptionV3_imagenet/model.c
INFO:tensorflow:Saving checkpoints for 3751 into InceptionV3_imagenet/model.c
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 3751...
INFO:tensorflow:Calling checkpoint listeners after saving checkpoint 3751...
INFO:tensorflow:Loss for final step: 0.23958135.
INFO:tensorflow:Loss for final step: 0.23958135.
CPU times: user 17 s, sys: 723 ms, total: 17.7 s
Wall time: 17.9 s
<tensorflow_estimator.python.estimator.estimator.EstimatorV2 at
0x7f6c300c1690>
```

```
imagenet_est_InceptionV3.evaluate(input_fn = lambda: train_input_fn(32), steps=Non
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Calling model_fn.
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/backend.py:434
  warnings.warn('`tf.keras.backend.set_learning_phase` is deprecated and '
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/trainir
  warnings.warn('`Model.state_updates` will be removed in a future version. '
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-03-24T18:48:19Z
INFO:tensorflow:Starting evaluation at 2021-03-24T18:48:19Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from InceptionV3_imagenet/model.ckpt-375
INFO:tensorflow:Restoring parameters from InceptionV3_imagenet/model.ckpt-375
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Inference Time : 38.55045s
INFO:tensorflow:Inference Time : 38.55045s
INFO:tensorflow:Finished evaluation at 2021-03-24-18:48:58
```

```
INFO:tensorflow:Finished evaluation at 2021-03-24-18:48:58
INFO:tensorflow:Saving dict for global step 3751: global_step = 3751, loss =
INFO:tensorflow:Saving dict for global step 3751: global_step = 3751, loss =
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 3751: Incept
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 3751: Incept
{'global_step': 3751,
 'loss': 0.48594028,
 'sparse_categorical_accuracy': 0.124151565}
```

Analyze history and metrics

```
%reload_ext tensorboard
%tensorboard --logdir ./InceptionV3_imagenet
```

```
Reusing TensorBoard on port 6006 (pid 282), started 0:00:20 ago. (Use '!kill
282' to kill it.)
```

Part 3.Keras - forever! :)

FSL, train->full(model), image=min(75x75), weights=None

- From Scratch Learning (FSL),
- train->full(model),
- image=min(75x75),
- weights=None.

Time: 1 epoch ---> 2 min

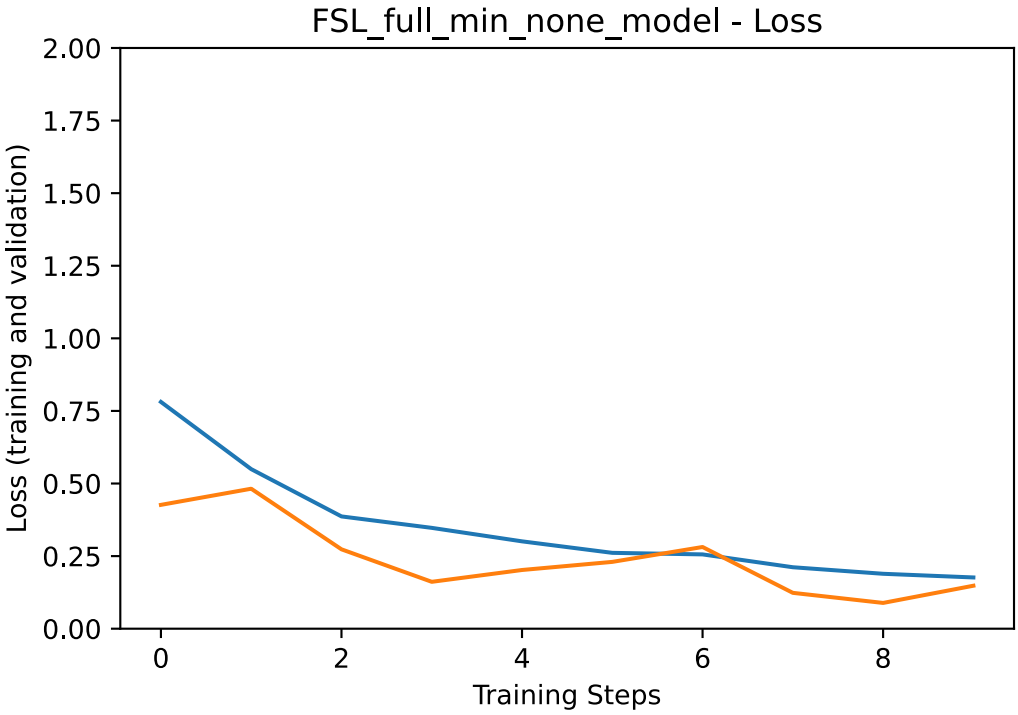
```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
model = tf.keras.applications.InceptionV3(
    input_shape=(75, 75, 3), # Input size must be at least 75x75
    include_top=True, weights=None) # differences!

FSL_full_min_none_model = model

FSL_full_min_none_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()
# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
IMG_SIZE = 75 # for Inception
```

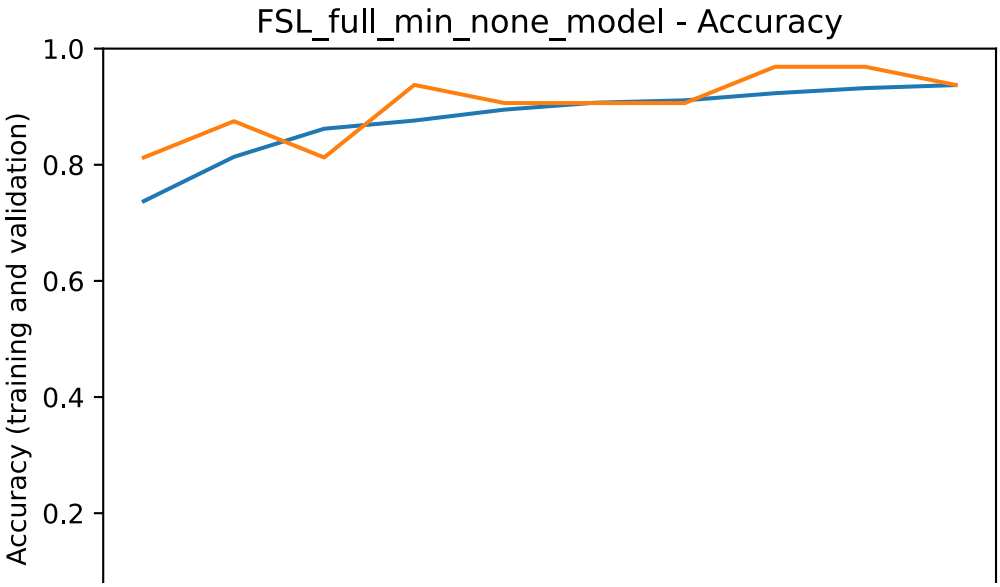
```
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tf.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
FSL_full_min_none_history = FSL_full_min_none_model.fit(train_batches, epochs=10,
                    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Epoch 1/10
1875/1875 [==============================] - 132s 68ms/step - loss: 0.9746 -
Epoch 2/10
1875/1875 [==============================] - 123s 66ms/step - loss: 0.5826 -
Epoch 3/10
1875/1875 [==============================] - 123s 66ms/step - loss: 0.4422 -
Epoch 4/10
1875/1875 [==============================] - 123s 65ms/step - loss: 0.3447 -
Epoch 5/10
1875/1875 [==============================] - 122s 65ms/step - loss: 0.3022 -
Epoch 6/10
1875/1875 [==============================] - 123s 65ms/step - loss: 0.2790 -
Epoch 7/10
1875/1875 [==============================] - 122s 65ms/step - loss: 0.2434 -
Epoch 8/10
1875/1875 [==============================] - 122s 65ms/step - loss: 0.2237 -
Epoch 9/10
1875/1875 [==============================] - 122s 65ms/step - loss: 0.1936 -
Epoch 10/10
1875/1875 [==============================] - 121s 65ms/step - loss: 0.1863 -
```

Plot History

```
plot_history(FSL_full_min_none_history.history, 'FSL_full_min_none_model')
```





TL, train->full(model), image=large(299x299), weights=ImageNet

1 epoch ---> 15 min

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 299 # for Inception: input size must be at least 75x75
model = tf.keras.applications.InceptionV3(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=True, weights='imagenet') # differences!

TL_full_large_imagenet_model = model

TL_full_large_imagenet_model.compile(optimizer='adam',
```

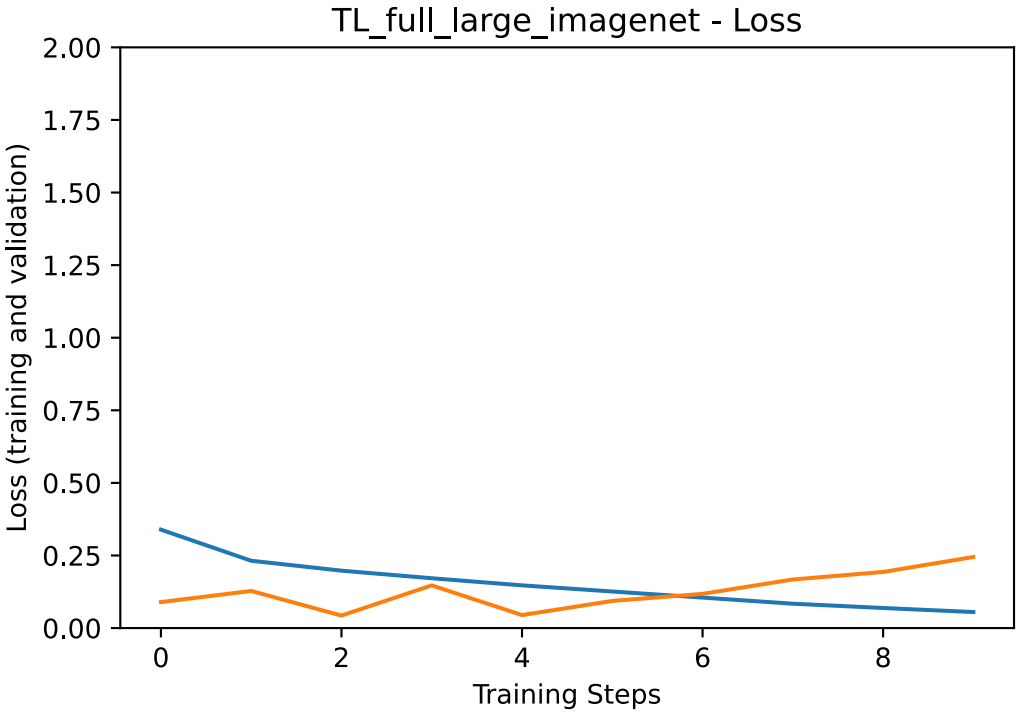
```
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tf.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_large_imagenet_history = TL_full_large_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

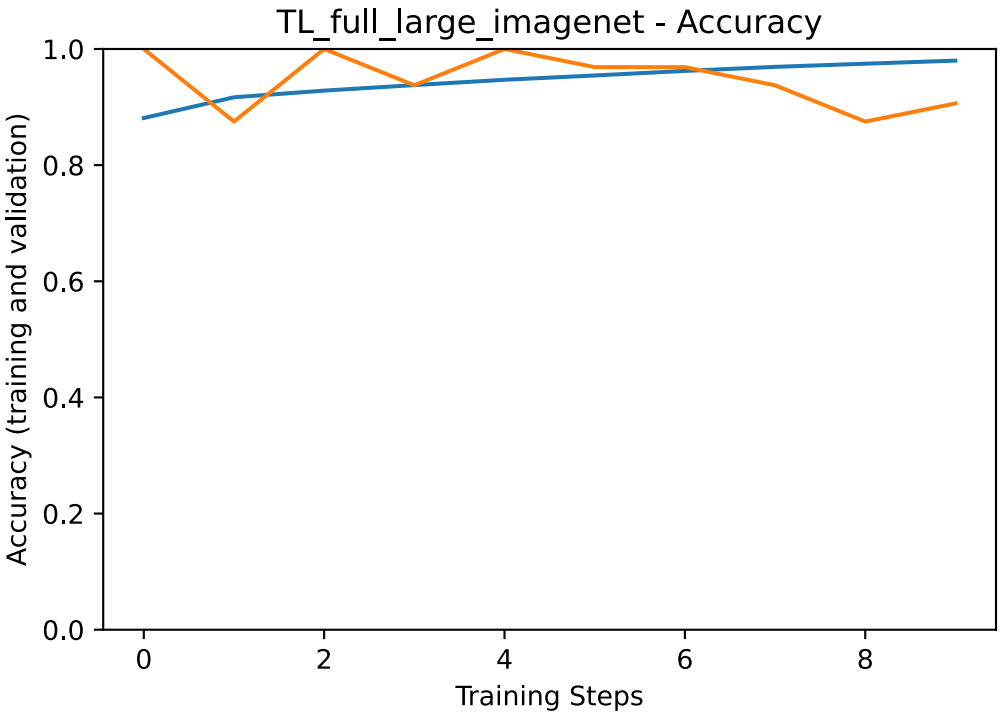
```
Epoch 1/10
1875/1875 [==============================] - 910s 482ms/step - loss: 0.4798 -
Epoch 2/10
1875/1875 [==============================] - 897s 478ms/step - loss: 0.2421 -
Epoch 3/10
1875/1875 [==============================] - 897s 478ms/step - loss: 0.2052 -
Epoch 4/10
1875/1875 [==============================] - 897s 478ms/step - loss: 0.1810 -
Epoch 5/10
1875/1875 [==============================] - 897s 478ms/step - loss: 0.1531 -
Epoch 6/10
1875/1875 [==============================] - 897s 478ms/step - loss: 0.1337 -
Epoch 7/10
1875/1875 [==============================] - 895s 478ms/step - loss: 0.1103 -
Epoch 8/10
1875/1875 [==============================] - 896s 478ms/step - loss: 0.0891 -
```

```
Epoch 9/10
1875/1875 [==============================] - 895s 477ms/step - loss: 0.0735 -
Epoch 10/10
1875/1875 [==============================] - 895s 477ms/step - loss: 0.0597 -
```

Plot History

```
plot_history(TL_full_large_imagenet_history.history, 'TL_full_large_imagenet')
```





TL, train->top(layer), image=min(75x75), weights->ImageNet

1 epoch ---> 45-38 sec

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 75 # for Inception: input size must be at least 75x75
model = tf.keras.applications.InceptionV3(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,  # train->top(layer) !
    weights='imagenet') # weights->ImageNet !

TL_top_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    #tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = False  # differences!

TL_top_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_top_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
```

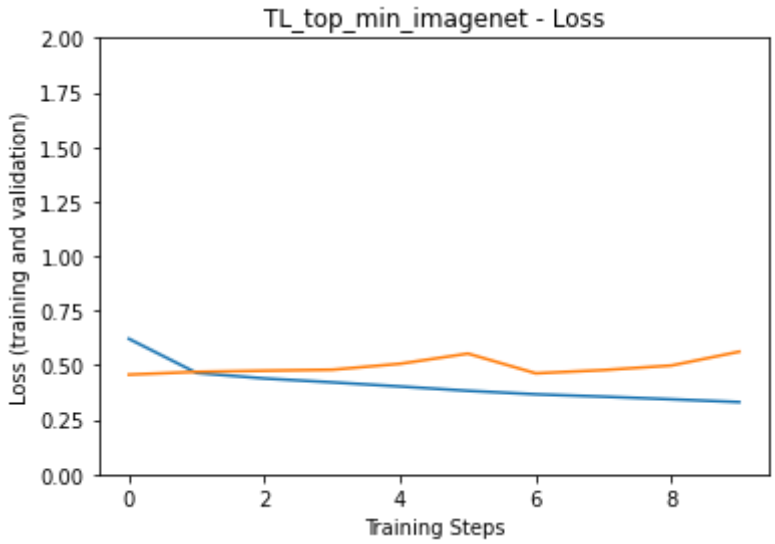
```
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_top_min_imagenet_history = TL_top_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

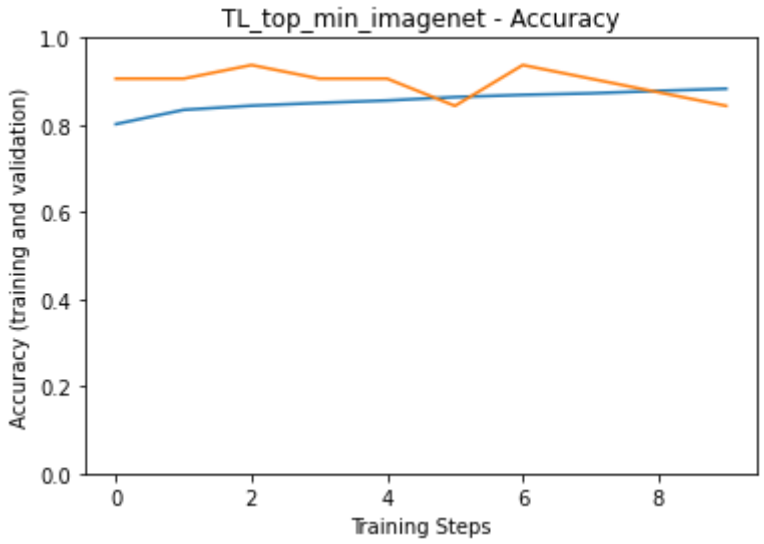
```
Model: "sequential_20"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 1, 1, 2048)        21802784
_________________________________________________________________
global_average_pooling2d_4 ( (None, 2048)              0
_________________________________________________________________
dense_28 (Dense)             (None, 256)               524544
_________________________________________________________________
dense_29 (Dense)             (None, 10)                2570
=================================================================
Total params: 22,329,898
Trainable params: 527,114
Non-trainable params: 21,802,784
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 45s 22ms/step - loss: 0.8347 - a
Epoch 2/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4633 - a
Epoch 3/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4394 - a
Epoch 4/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4275 - a
Epoch 5/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4074 - a
Epoch 6/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3879 - a
Epoch 7/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3686 - a
Epoch 8/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3544 - a
Epoch 9/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3436 - a
Epoch 10/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3329 - a
```

Plot History

```
import matplotlib.pylab as plt

# Plot Training History
def plot_history(hist, label):
  plt.figure()
```

```
  plt.ylabel("Loss (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,2])
  plt.plot(hist["loss"])
  plt.plot(hist["val_loss"])
  plt.title(label + ' - Loss')

  plt.figure()
  plt.ylabel("Accuracy (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,1])
  plt.plot(hist["accuracy"])
  plt.plot(hist["val_accuracy"])
  plt.title(label + ' - Accuracy')

plot_history(TL_top_min_imagenet_history.history, 'TL_top_min_imagenet')
```





TL, train->full(model), image=min(75x75), weights=ImageNet

1 epoch ---> 2 min

```
import tensorflow as tf
```

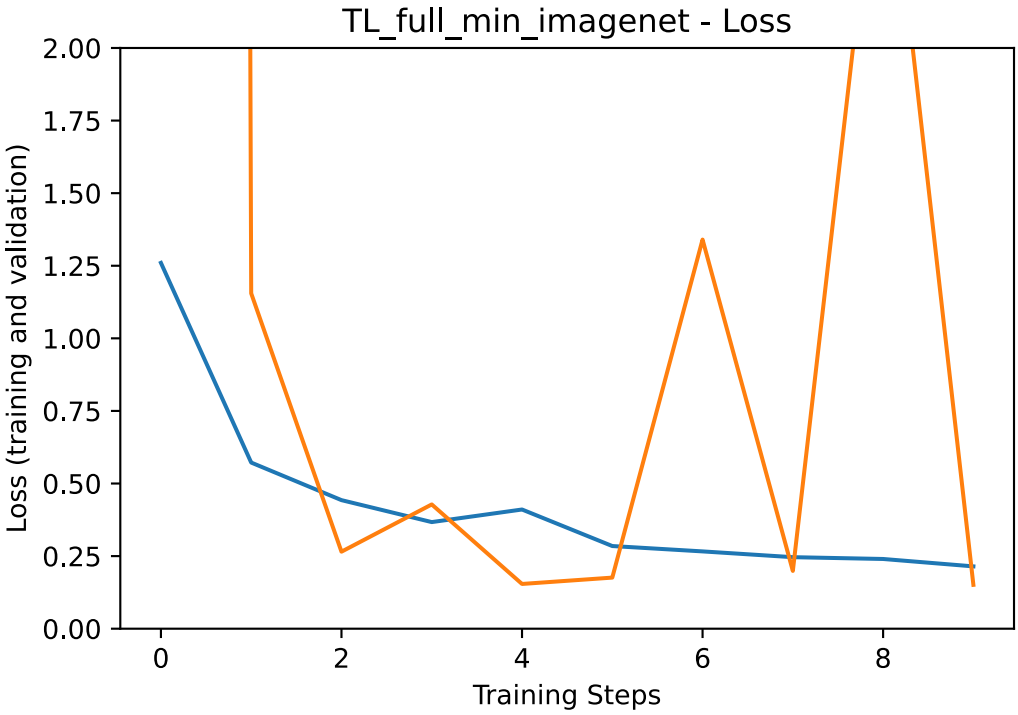
```
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 75 # for Inception: input size must be at least 75x75
model = tf.keras.applications.InceptionV3(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False, weights='imagenet') # differences!

TL_full_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = True

TL_full_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_full_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tf.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_min_imagenet_history = TL_full_min_imagenet_model.fit(
```

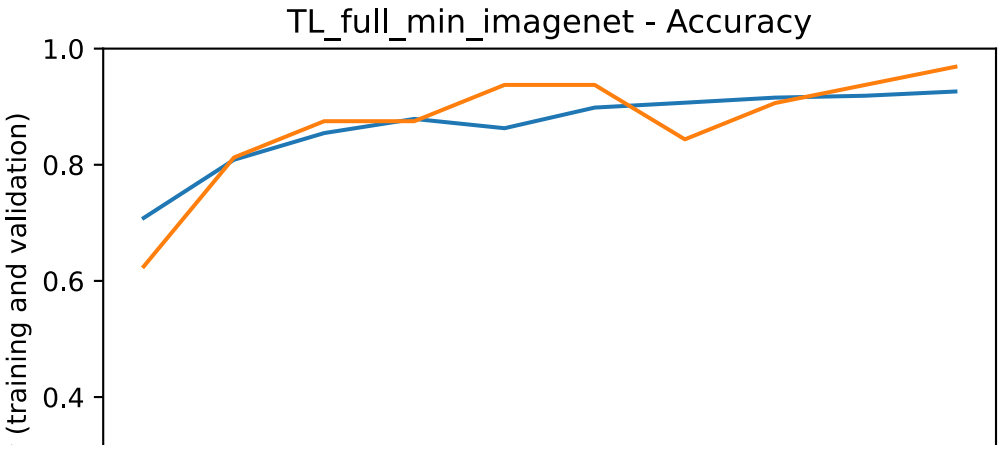
```
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Model: "sequential_34"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 1, 1, 2048)        21802784
_________________________________________________________________
global_average_pooling2d_8 ( (None, 2048)              0
_________________________________________________________________
dense_34 (Dense)             (None, 256)               524544
_________________________________________________________________
dropout_7 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_35 (Dense)             (None, 10)                2570
=================================================================
Total params: 22,329,898
Trainable params: 22,295,466
Non-trainable params: 34,432
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 128s 65ms/step - loss: 1.3384 -
Epoch 2/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.6455 -
Epoch 3/10
1875/1875 [==============================] - 120s 64ms/step - loss: 0.4653 -
Epoch 4/10
1875/1875 [==============================] - 119s 64ms/step - loss: 0.3546 -
Epoch 5/10
1875/1875 [==============================] - 119s 64ms/step - loss: 0.5048 -
Epoch 6/10
1875/1875 [==============================] - 119s 64ms/step - loss: 0.2965 -
Epoch 7/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.2692 -
Epoch 8/10
1875/1875 [==============================] - 119s 64ms/step - loss: 0.2568 -
Epoch 9/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.2336 -
Epoch 10/10
1875/1875 [==============================] - 119s 64ms/step - loss: 0.2255 -
```

Plot History

```
plot_history(TL_full_min_imagenet_history.history, 'TL_full_min_imagenet')
```





Plot ALL training histories

```
# Plot ALL Training Histories
def plot_all_histories(hist, label):
  plt.figure()
  plt.ylabel("Loss (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
    plt.plot(h["val_loss"], label=l)
  plt.title('Loss (validation)')
  plt.legend(loc=2)

  plt.figure()
  plt.ylabel("Accuracy (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1.2])
```

```
# Lecture 08. Modern CNN: DEMO 6. ResNet

(C) partially based on [7.6. Residual Ne
chapter_convolutional-modern/resnet.html
```

Lecture 08. Modern CNN: DEMO 6. ResNet

(C) partially based on [7.6. Residual Networks (ResNet)](...) from d2l Open Source book.

Preparatory Actions

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 8.3MB/s
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: ipython>=4.0.0; python_version >= "3.3" in /
Requirement already satisfied: nbformat>=4.2.0 in /usr/local/lib/python3.7/
Requirement already satisfied: traitlets>=4.3.1 in /usr/local/lib/python3.7
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: jupyter-client>=4.1 in /usr/local/lib/python
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
```

```
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
```

Part 1. Theory - Residual Networks (ResNet)

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different. To make some progress we need a bit of mathematics.

Function Classes

Consider $\mathcal{F}$, the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, for all $f \in \mathcal{F}$ there exists some set of parameters (e.g., weights and biases) that can be obtained through training on a suitable dataset. Let us assume that f^* is the "truth" function that we really would like to find. If it is in $\mathcal{F}$, we are in good shape but typically we will not be quite so lucky. Instead, we will try to find some $f^*_\mathcal{F}$ which is our best bet within $\mathcal{F}$. For instance, given a dataset with features $\mathbf{X}$ and labels $\mathbf{y}$, we might try finding it by solving the following optimization problem:

$$f^*_\mathcal{F} \stackrel{\mathrm{def}}{=} \operatorname*{argmin}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

It is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}'$ we should arrive at a better outcome. In other words, we would expect that $f^*_{\mathcal{F}'}$ is "better" than $f^*_\mathcal{F}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee that this should even happen. In fact, $f^*_{\mathcal{F}'}$ might well be worse. As illustrated by Fig. 7.6.1, for non-nested function classes, a larger function class does not always move closer to the "truth" function f^*. For instance, on the left of Fig. 7.6.1, though $\mathcal{F}_3$ is closer to f^* than $\mathcal{F}_1$, $\mathcal{F}_6$ moves away and there is no guarantee that further increasing the complexity can reduce the distance from f^*. With nested function classes where

$\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$ on the right of Fig. 7.6.1, we can avoid the aforementioned issue from the non-nested function classes.

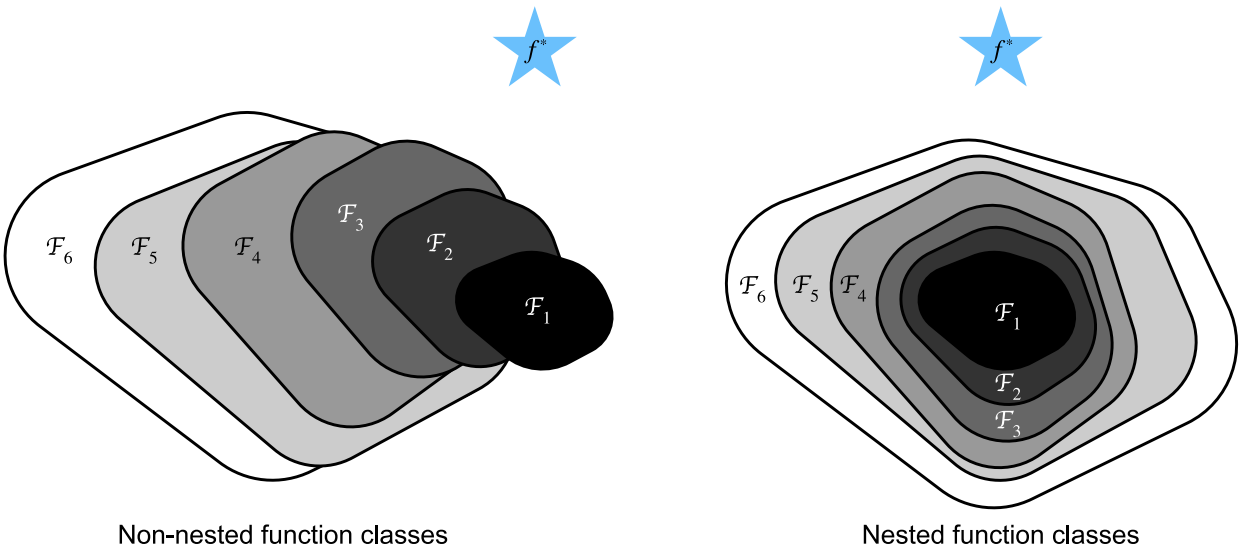


Fig. 7.6.1. For non-nested function classes, a larger (indicated by area) function class does not guarantee to get closer to the "truth" function (f^*). This does not happen in nested function classes.

Thus, only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. For deep neural networks, if we can train the newly-added layer into an identity function $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model. As the new model may get a better solution to fit the training dataset, the added layer might make it easier to reduce training errors.

This is the question that He et al. considered when working on very deep computer vision models He et al., 2016a. At the heart of their proposed *residual network* (*ResNet*) is the idea that every additional layer should more easily contain the identity function as one of its elements. These considerations are rather profound but they led to a surprisingly simple solution, a *residual block*. With it, ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural networks.

Residual Blocks

Let us focus on a local part of a neural network, as depicted in Fig. 7.6.2. Denote the input by $\mathbf{x}$. We assume that the desired underlying mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function on the top. On the left of Fig. 7.6.2, the portion within the dotted-line box must directly learn the mapping $f(\mathbf{x})$. On the right, the portion within the dotted-line box needs to learn the *residual mapping* $f(\mathbf{x}) - \mathbf{x}$, which is how the residual block derives its name. If the identity mapping $f(\mathbf{x}) = \mathbf{x}$ is the desired underlying mapping, the residual mapping is easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully-connected layer and convolutional layer) within the dotted-line box to zero. The right figure in Fig. 7.6.2 illustrates the *residual block* of ResNet, where the solid line carrying the layer input $\mathbf{x}$ to the addition operator is called a *residual connection* (or *shortcut*

connection). With residual blocks, inputs can forward propagate faster through the residual connections across layers.

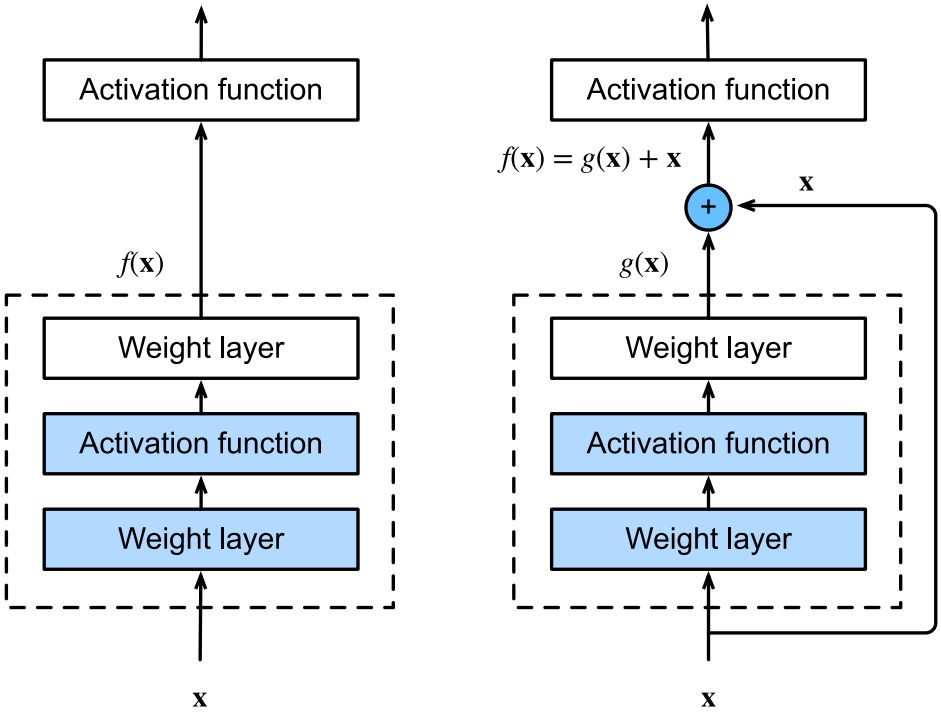


Fig. 7.6.2. A regular block (left) and a residual block (right).

ResNet follows VGG's full 3×3 convolutional layer design. The residual block has two 3×3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together. If we want to change the number of channels, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. Let us have a look at the code below.

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
import tensorflow as tf
from d2l import tensorflow as d2l

class Residual(tf.keras.Model):  #@save
    """The Residual block of ResNet."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = tf.keras.layers.Conv2D(num_channels, padding='same',
                                            kernel_size=3, strides=strides)
        self.conv2 = tf.keras.layers.Conv2D(num_channels, kernel_size=3,
                                            padding='same')
        self.conv3 = None
        if use_1x1conv:
            self.conv3 = tf.keras.layers.Conv2D(num_channels, kernel_size=1,
                                                strides=strides)
        self.bn1 = tf.keras.layers.BatchNormalization()
```

```
        self.bn2 = tf.keras.layers.BatchNormalization()

    def call(self, X):
        Y = tf.keras.activations.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3 is not None:
            X = self.conv3(X)
        Y += X
        return tf.keras.activations.relu(Y)
```

This code generates two types of networks: one where we add the input to the output before applying the ReLU nonlinearity whenever `use_1x1conv=False`, and one where we adjust channels and resolution by means of a 1×1 convolution before adding. Fig. 7.6.3 illustrates this:

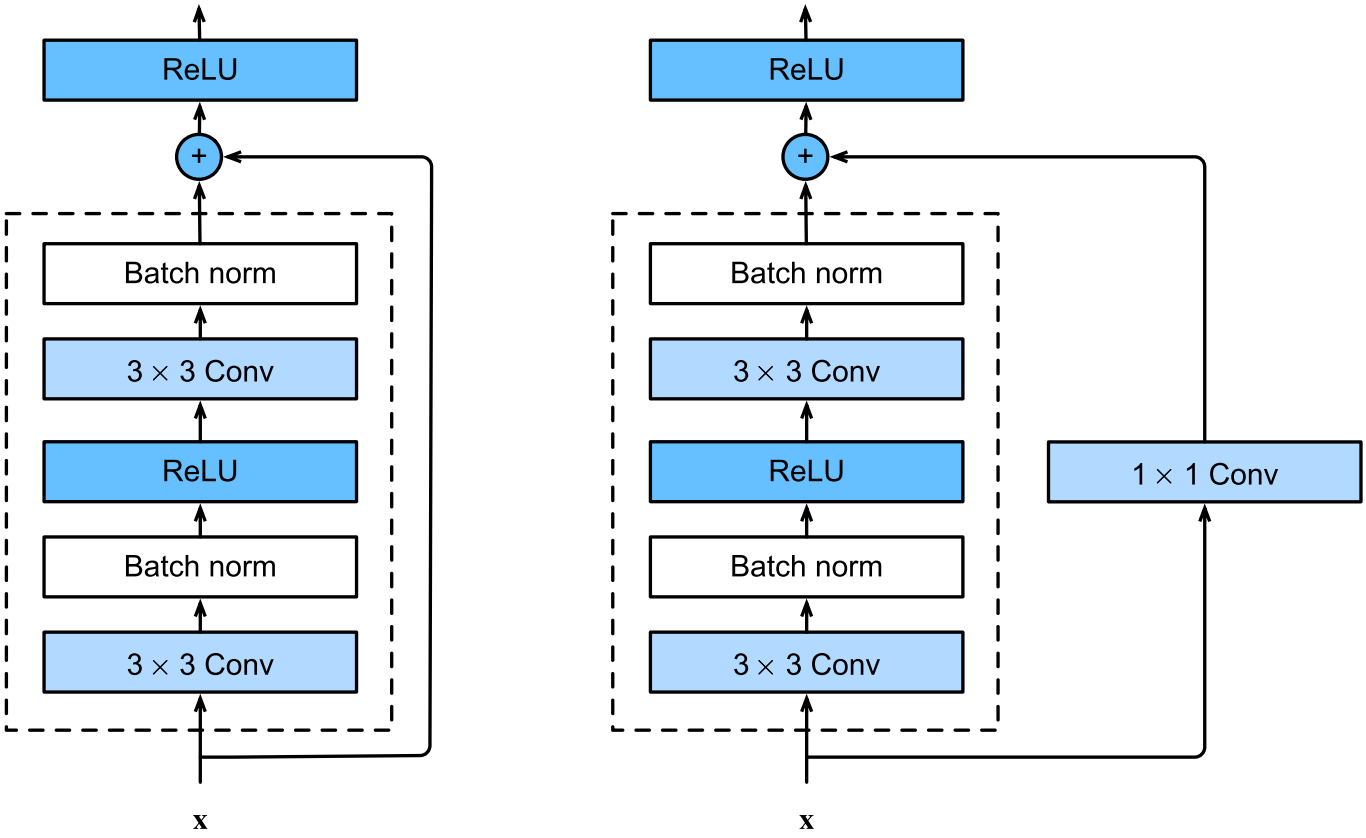


Fig. 7.6.3. ResNet block with and without 1×1 convolution.

Now let us look at a situation where the input and output are of the same shape.

```
blk = Residual(3)
X = tf.random.uniform((4, 6, 6, 3))
Y = blk(X)
Y.shape
```

```
    TensorShape([4, 6, 6, 3])
```

We also have the option to halve the output height and width while increasing the number of

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
TensorShape([4, 3, 3, 6])
```

ResNet Model

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 maximum pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.

```
b1 = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=7, strides=2, padding='same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
```

GoogLeNet uses four modules made up of Inception blocks. However, ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels. The number of channels in the first module is the same as the number of input channels. Since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width. In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

Now, we implement this module. Note that special processing has been performed on the first module.

```
class ResnetBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels, num_residuals, first_block=False,
                 **kwargs):
        super(ResnetBlock, self).__init__(**kwargs)
        self.residual_layers = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                self.residual_layers.append(
                    Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                self.residual_layers.append(Residual(num_channels))

    def call(self, X):
        for layer in self.residual_layers.layers:
            X = layer(X)
        return X
```

Then, we add all the modules to ResNet. Here, two residual blocks are used for each module.

```
b2 = ResnetBlock(64, 2, first_block=True)
b3 = ResnetBlock(128, 2)
b4 = ResnetBlock(256, 2)
b5 = ResnetBlock(512, 2)
```

Finally, just like GoogLeNet, we add a global average pooling layer, followed by the fully-connected layer output.

```
# Recall that we define this as a function so we can reuse later and run it
# within `tf.distribute.MirroredStrategy`'s scope to utilize various
# computational resources, e.g. GPUs. Also note that even though we have
# created b1, b2, b3, b4, b5 but we will recreate them inside this function's
# scope instead
def net():
    return tf.keras.Sequential([
        # The following layers are the same as b1 that we created earlier
        tf.keras.layers.Conv2D(64, kernel_size=7, strides=2, padding='same'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Activation('relu'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same'),
        # The following layers are the same as b2, b3, b4, and b5 that we
        # created earlier
        ResnetBlock(64, 2, first_block=True),
        ResnetBlock(128, 2),
        ResnetBlock(256, 2),
        ResnetBlock(512, 2),
        tf.keras.layers.GlobalAvgPool2D(),
        tf.keras.layers.Dense(units=10)])
```

There are 4 convolutional layers in each module (excluding the 1×1 convolutional layer). Together with the first 7×7 convolutional layer and the final fully-connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler and easier to modify. All these factors have resulted in the rapid and widespread use of ResNet. Fig. 7.6.4 depicts the full ResNet-18.

 The ResNet-18 architecture.

Fig. 7.6.4. The ResNet-18 architecture.

Before training ResNet, let us observe how the input shape changes across different modules in ResNet. As in all the previous architectures, the resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

```
X = tf.random.uniform(shape=(1, 224, 224, 1))
for layer in net().layers:
    X = layer(X)
    print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

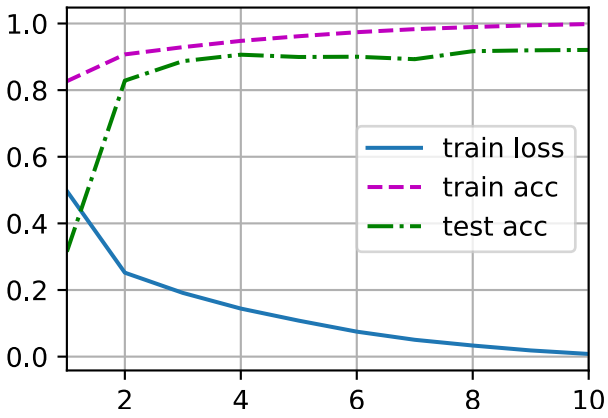
```
Conv2D output shape:     (1, 112, 112, 64)
BatchNormalization output shape:         (1, 112, 112, 64)
Activation output shape:         (1, 112, 112, 64)
MaxPooling2D output shape:       (1, 56, 56, 64)
ResnetBlock output shape:        (1, 56, 56, 64)
ResnetBlock output shape:        (1, 28, 28, 128)
ResnetBlock output shape:        (1, 14, 14, 256)
ResnetBlock output shape:        (1, 7, 7, 512)
GlobalAveragePooling2D output shape:     (1, 512)
Dense output shape:      (1, 10)
```

Training

We train ResNet on the Fashion-MNIST dataset, just like before.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.008, train acc 0.998, test acc 0.921
5263.9 examples/sec on /GPU:0
<tensorflow.python.keras.engine.sequential.Sequential at
0x7f4c1945b940>
```



Part 2.Estimators + TF2-Keras Models - NOT here :)

Part 3.Keras - forever! :)

FSL, train->full(model), image=min(75x75), weights=None

- From Scratch Learning (FSL),
- train->full(model),
- image=min(75x75),
- weights=None.

Time: 1 epoch ---> 2 min

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
model = tf.keras.applications.ResNet50(
    input_shape=(75, 75, 3), # Input size must be at least 75x75
    include_top=True, weights=None) # differences!

FSL_full_min_none_model = model

FSL_full_min_none_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()
# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
IMG_SIZE = 75 # for Inception
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tf.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END
```

```
# LOAD PHASE START #
FSL_full_min_none_history = FSL_full_min_none_model.fit(train_batches, epochs=10,
                    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Epoch 1/10
1875/1875 [==============================] - 132s 68ms/step - loss: 0.9746 -
Epoch 2/10
 911/1875 [=============>................] - ETA: 1:03 - loss: 0.6181 - accur
```

Plot History

```
plot_history(FSL_full_min_none_history.history, 'FSL_full_min_none_model')
```

TL, train->full(model), image=large(299x299), weights=ImageNet

1 epoch ---> 15 min

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 299 # for Inception: input size must be at least 75x75
model = tf.keras.applications.ResNet50(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=True, weights='imagenet') # differences!

TL_full_large_imagenet_model = model

TL_full_large_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
```

```
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tf.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_large_imagenet_history = TL_full_large_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

Plot History

```
plot_history(TL_full_large_imagenet_history.history, 'TL_full_large_imagenet')
```

TL, train->top(layer), image=min(75x75), weights->ImageNet

1 epoch ---> 45-38 sec

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 75 # for Inception: input size must be at least 75x75
model = tf.keras.applications.ResNet50(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,  # train->top(layer) !
    weights='imagenet') # weights->ImageNet !

TL_top_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    #tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = False  # differences!
```

```
TL_top_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_top_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_top_min_imagenet_history = TL_top_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

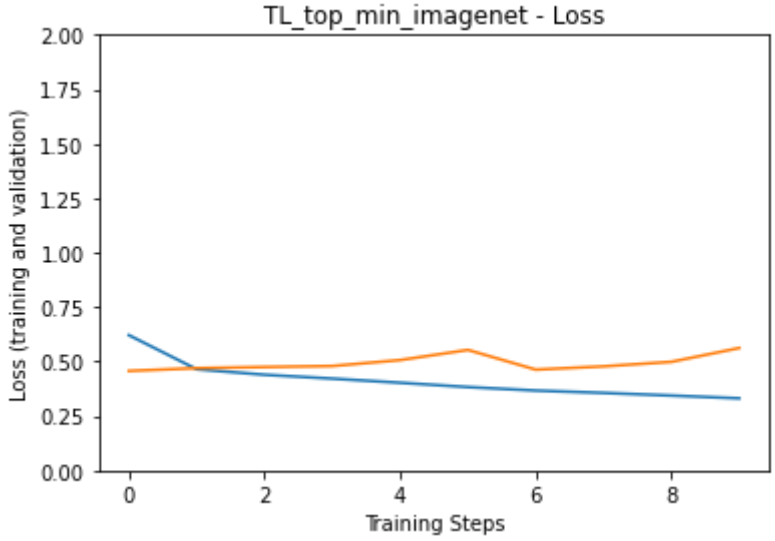
```
Model: "sequential_20"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 1, 1, 2048)        21802784
_________________________________________________________________
global_average_pooling2d_4 ( (None, 2048)              0
_________________________________________________________________
dense_28 (Dense)             (None, 256)               524544
_________________________________________________________________
dense_29 (Dense)             (None, 10)                2570
=================================================================
Total params: 22,329,898
Trainable params: 527,114
Non-trainable params: 21,802,784
```

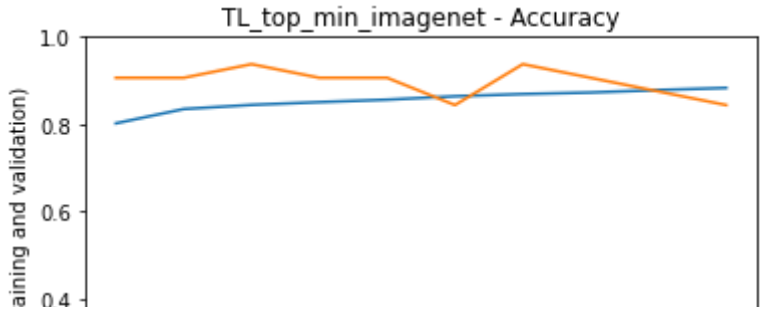
```
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 45s 22ms/step - loss: 0.8347 - a
Epoch 2/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4633 - a
Epoch 3/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4394 - a
Epoch 4/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4275 - a
Epoch 5/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.4074 - a
Epoch 6/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3879 - a
Epoch 7/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3686 - a
Epoch 8/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3544 - a
Epoch 9/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3436 - a
Epoch 10/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.3329 - a
```

Plot History

```
import matplotlib.pylab as plt

# Plot Training History
def plot_history(hist, label):
  plt.figure()
  plt.ylabel("Loss (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,2])
  plt.plot(hist["loss"])
  plt.plot(hist["val_loss"])
  plt.title(label + ' - Loss')

  plt.figure()
  plt.ylabel("Accuracy (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,1])
  plt.plot(hist["accuracy"])
  plt.plot(hist["val_accuracy"])
  plt.title(label + ' - Accuracy')
```

```
plot_history(TL_top_min_imagenet_history.history, 'TL_top_min_imagenet')
```





TL, train->full(model), image=min(75x75), weights=ImageNet

1 epoch ---> 2 min

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 75 # for Inception: input size must be at least 75x75
model = tf.keras.applications.ResNet50(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False, weights='imagenet') # differences!

TL_full_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = True

TL_full_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_full_min_imagenet_model.summary()
```

```
# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tf.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_min_imagenet_history = TL_full_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

Plot History

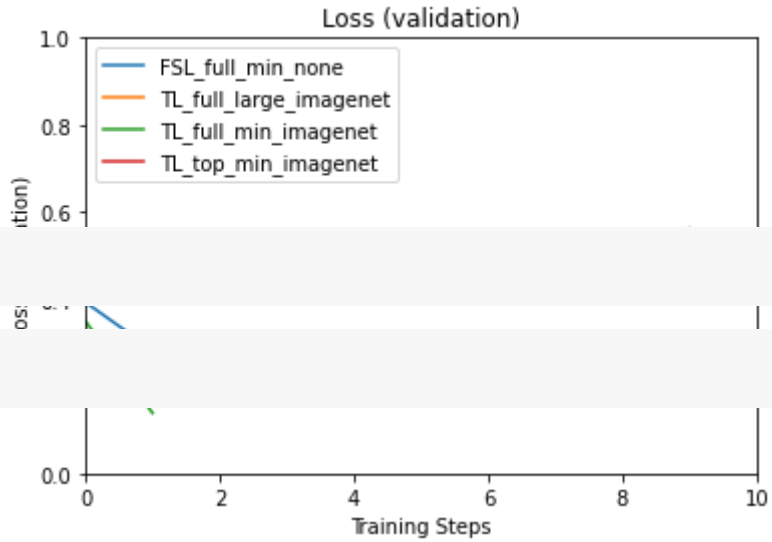
```
plot_history(TL_full_min_imagenet_history.history, 'TL_full_min_imagenet')
```

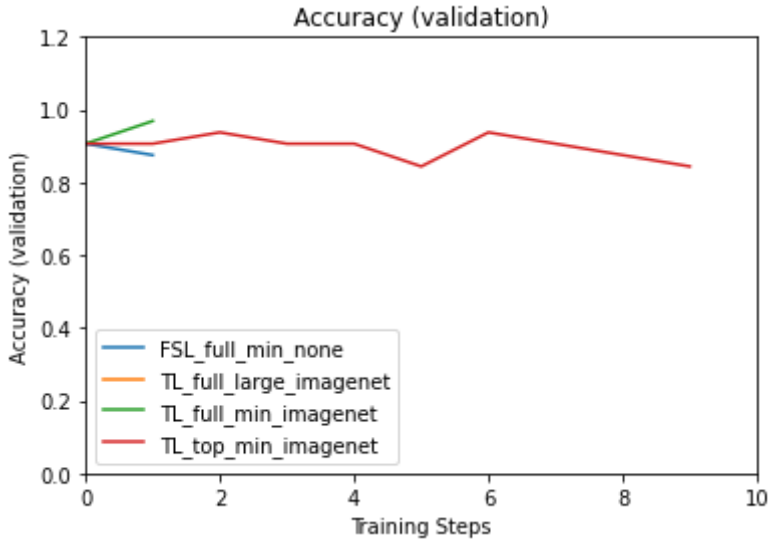
Plot ALL training histories

```
# Plot ALL Training Histories
def plot_all_histories(hist, label):
  plt.figure()
  plt.ylabel("Loss (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
```

```
    plt.plot(h["val_loss"], label=l)
  plt.title('Loss (validation)')
  plt.legend(loc=2)

  plt.figure()
  plt.ylabel("Accuracy (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1.2])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
    plt.plot(h["val_accuracy"], label=l)
  plt.title('Accuracy (validation)')
  plt.legend(loc=3)
```

```
hist = [
        FSL_full_min_none_history.history,
        TL_full_large_imagenet_history.history,
        TL_full_min_imagenet_history.history,
        TL_top_min_imagenet_history.history
        ]

label = [
        'FSL_full_min_none',
        'TL_full_large_imagenet',
        'TL_full_min_imagenet',
        'TL_top_min_imagenet'
        ]

plot_all_histories(hist, label)
```





Lecture 08. Modern CNN: DEMO 7. DenseNet

(C) partially based on 7.7. Densely Connected Networks (DenseNet) from d2l Open Source book.

Preparatory Actions

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 8.3MB/s
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: ipython>=4.0.0; python_version >= "3.3" in /
Requirement already satisfied: nbformat>=4.2.0 in /usr/local/lib/python3.7/
Requirement already satisfied: traitlets>=4.3.1 in /usr/local/lib/python3.7
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: jupyter-client>=4.1 in /usr/local/lib/python
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
```

```
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
```

```
import tensorflow as tf
```

Part 1.Theory - Densely Connected Networks (DenseNet)

ResNet significantly changed the view of how to parametrize the functions in deep networks. *DenseNet* (dense convolutional network) is to some extent the logical extension of this Huang et al., 2017. To understand how to arrive at it, let us take a small detour to mathematics.

From ResNet to DenseNet

Recall the Taylor expansion for functions. For the point $x = 0$ it can be written as

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \ldots.$$

The key point is that it decomposes a function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}).$$

That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one. What if we want to capture (not necessarily add) information beyond two terms? One solution was DenseNet Huang et al., 2017.

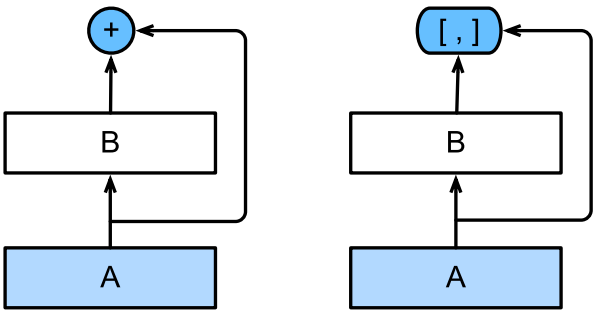


Fig. 7.7.1. The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

As shown in Fig. 7.7.1., the key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* (denoted by $[,]$) rather than added. As a result, we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions:

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots] .$$

In the end, all these functions are combined in MLP to reduce the number of features again. In terms of implementation this is quite simple: rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers. The dense connections are shown in Fig. 7.7.2.

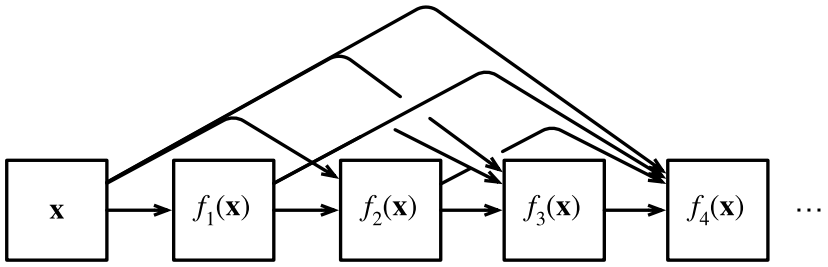


Fig. 7.7.2 Dense connections in DenseNet.

The main components that compose a DenseNet are *dense blocks* and *transition layers*. The former define how the inputs and outputs are concatenated, while the latter control the number of channels so that it is not too large.

Dense Blocks

DenseNet uses the modified "batch normalization, activation, and convolution" structure of ResNet. First, we implement this convolution block structure.

```
import tensorflow as tf
from d2l import tensorflow as d2l

class ConvBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels):
        super(ConvBlock, self).__init__()
        self.bn = tf.keras.layers.BatchNormalization()
        self.relu = tf.keras.layers.ReLU()
        self.conv = tf.keras.layers.Conv2D(filters=num_channels,
                                           kernel_size=(3, 3), padding='same')

        self.listLayers = [self.bn, self.relu, self.conv]

    def call(self, x):
        y = x
```

```
        for layer in self.listLayers.layers:
            y = layer(y)
        y = tf.keras.layers.concatenate([x, y], axis=-1)
        return y
```

A *dense block* consists of multiple convolution blocks, each using the same number of output channels. In the forward propagation, however, we concatenate the input and output of each convolution block on the channel dimension.

```
class DenseBlock(tf.keras.layers.Layer):
    def __init__(self, num_convs, num_channels):
        super(DenseBlock, self).__init__()
        self.listLayers = []
        for _ in range(num_convs):
            self.listLayers.append(ConvBlock(num_channels))

    def call(self, x):
        for layer in self.listLayers.layers:
            x = layer(x)
        return x
```

In the following example, we define a `DenseBlock` instance with 2 convolution blocks of 10 output channels. When using an input with 3 channels, we will get an output with $3 + 2 \times 10 = 23$ channels. The number of convolution block channels controls the growth in the number of output channels relative to the number of input channels. This is also referred to as the *growth rate*.

```
blk = DenseBlock(2, 10)
X = tf.random.uniform((4, 8, 8, 3))
Y = blk(X)
Y.shape
```

```
TensorShape([4, 8, 8, 23])
```

Transition Layers

Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A *transition layer* is used to control the complexity of the model. It reduces the number of channels by using the 1×1 convolutional layer and halves the height and width of the average pooling layer with a stride of 2, further reducing the complexity of the model.

```
class TransitionBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels, **kwargs):
        super(TransitionBlock, self).__init__(**kwargs)
        self.batch_norm = tf.keras.layers.BatchNormalization()
```

```
        self.relu = tf.keras.layers.ReLU()
        self.conv = tf.keras.layers.Conv2D(num_channels, kernel_size=1)
        self.avg_pool = tf.keras.layers.AvgPool2D(pool_size=2, strides=2)

    def call(self, x):
        x = self.batch_norm(x)
        x = self.relu(x)
        x = self.conv(x)
        return self.avg_pool(x)
```

Apply a transition layer with 10 channels to the output of the dense block in the previous example. This reduces the number of output channels to 10, and halves the height and width.

```
blk = TransitionBlock(10)
blk(Y).shape
```

```
TensorShape([4, 4, 4, 10])
```

DenseNet Model

Next, we will construct a DenseNet model. DenseNet first uses the same single convolutional layer and maximum pooling layer as in ResNet.

```
def block_1():
    return tf.keras.Sequential([
        tf.keras.layers.Conv2D(64, kernel_size=7, strides=2, padding='same'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.ReLU(),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
```

Then, similar to the four modules made up of residual blocks that ResNet uses, DenseNet uses four dense blocks. Similar to ResNet, we can set the number of convolutional layers used in each dense block. Here, we set it to 4, consistent with the ResNet-18 model in :numref:`sec_resnet`. Furthermore, we set the number of channels (i.e., growth rate) for the convolutional layers in the dense block to 32, so 128 channels will be added to each dense block.

In ResNet, the height and width are reduced between each module by a residual block with a stride of 2. Here, we use the transition layer to halve the height and width and halve the number of channels.

```
def block_2():
    net = block_1()
    # `num_channels`: the current number of channels
    num_channels, growth_rate = 64, 32
    num_convs_in_dense_blocks = [4, 4, 4, 4]
```

```
    for i, num_convs in enumerate(num_convs_in_dense_blocks):
        net.add(DenseBlock(num_convs, growth_rate))
        # This is the number of output channels in the previous dense block
        num_channels += num_convs * growth_rate
        # A transition layer that halves the number of channels is added
        # between the dense blocks
        if i != len(num_convs_in_dense_blocks) - 1:
            num_channels //= 2
            net.add(TransitionBlock(num_channels))
    return net
```

Similar to ResNet, a global pooling layer and a fully-connected layer are connected at the end to produce the output.

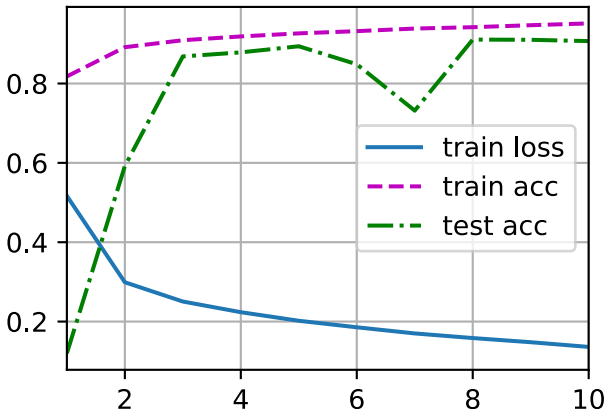
```
def net():
    net = block_2()
    net.add(tf.keras.layers.BatchNormalization())
    net.add(tf.keras.layers.ReLU())
    net.add(tf.keras.layers.GlobalAvgPool2D())
    net.add(tf.keras.layers.Flatten())
    net.add(tf.keras.layers.Dense(10))
    return net
```

Training

Since we are using a deeper network here, in this section, we will reduce the input height and width from 224 to 96 to simplify the computation.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.136, train acc 0.951, test acc 0.907
6490.0 examples/sec on /GPU:0
<tensorflow.python.keras.engine.sequential.Sequential at
0x7f679e16b910>
```



Summary

- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.
- The main components that compose DenseNet are dense blocks and transition layers.
- We need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.

Part 2.Estimators + TF2-Keras Models - NOT here :)

Part 3.Keras - forever! :)

```
import matplotlib.pylab as plt

# Plot Training History
def plot_history(hist, label):
  plt.figure()
  plt.ylabel("Loss (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,2])
  plt.plot(hist["loss"])
  plt.plot(hist["val_loss"])
  plt.title(label + ' - Loss')

  plt.figure()
  plt.ylabel("Accuracy (training and validation)")
  plt.xlabel("Training Steps")
  plt.ylim([0,1])
  plt.plot(hist["accuracy"])
  plt.plot(hist["val_accuracy"])
  plt.title(label + ' - Accuracy')
```

FSL, train->full(model), image=min(32x32), weights=None

- From Scratch Learning (FSL),
- train->full(model),
- image=min(75x75),
- weights=None.

Time: 1 epoch ---> 150 - 104 sec

```
import tensorflow as tf
import tensorflow_datasets as tfds
```

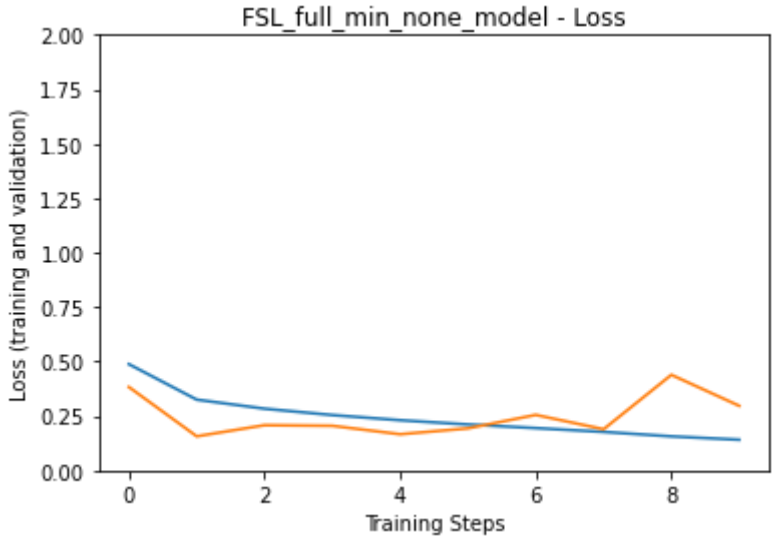
```
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 32 # for DenseNet121 # Input size must be at least 32x32
model = tf.keras.applications.DenseNet121(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=True, weights=None) # differences!

FSL_full_min_none_model = model

FSL_full_min_none_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()
# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
FSL_full_min_none_history = FSL_full_min_none_model.fit(train_batches, epochs=10,
                    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

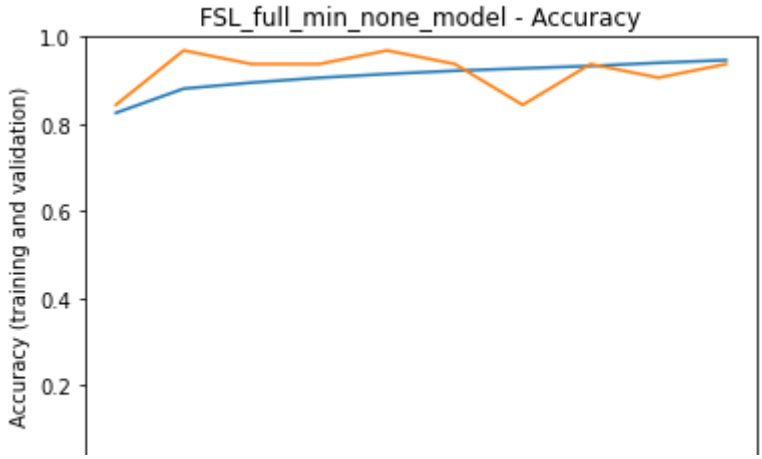
```
    Epoch 1/10
    1875/1875 [==============================] - 150s 58ms/step - loss: 0.7022 -
    Epoch 2/10
    1875/1875 [==============================] - 104s 56ms/step - loss: 0.3408 -
    Epoch 3/10
    1875/1875 [==============================] - 103s 55ms/step - loss: 0.2959 -
```

```
Epoch 4/10
1875/1875 [==============================] - 104s 56ms/step - loss: 0.2614 -
Epoch 5/10
1875/1875 [==============================] - 104s 56ms/step - loss: 0.2368 -
Epoch 6/10
1875/1875 [==============================] - 103s 55ms/step - loss: 0.2167 -
Epoch 7/10
1875/1875 [==============================] - 103s 55ms/step - loss: 0.1971 -
Epoch 8/10
1875/1875 [==============================] - 104s 56ms/step - loss: 0.1874 -
Epoch 9/10
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1616 -
Epoch 10/10
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1461 -
```

Plot History

```
plot_history(FSL_full_min_none_history.history, 'FSL_full_min_none_model')
```





TL, train->full(model), image=large(224x224), weights=ImageNet

1 epoch ---> 15 min --> VERY long NOT finished :)

```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 224 # for DenseNet121: input size must be at least 32x32
model = tf.keras.applications.DenseNet121(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=True, weights='imagenet') # differences!

TL_full_large_imagenet_model = model

TL_full_large_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_large_imagenet_history = TL_full_large_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Downloading data from https://storage.googleapis.com/tensorflow/ker
33193984/33188688 [==============================] - 0s 0us/step
Epoch 1/10
  41/1875 [..............................] - ETA: 14:39 - loss: 2.4
---------------------------------------------------------------------
--------
KeyboardInterrupt                         Traceback (most recent
call last)
<ipython-input-8-334478723f83> in <module>()
     49 TL_full_large_imagenet_history =
TL_full_large_imagenet_model.fit(
     50     train_batches, epochs=10,
---> 51     validation_data=validation_batches, validation_steps=1)
     52 # LOAD PHASE END #

                                  12 frames
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/framework/ops.py in _numpy(self)
   1035   def _numpy(self):
   1036     try:
-> 1037       return self._numpy_internal()
   1038     except core._NotOkStatusException as e:  # pylint:
```

Plot History

```
plot_history(TL_full_large_imagenet_history.history, 'TL_full_large_imagenet')
```

TL, train->top(layer), image=min(32x32), weights->ImageNet

1 epoch ---> 43-32 sec

```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 32 # for DenseNet121: input size must be at least 32x32
model = tf.keras.applications.DenseNet121(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,  # train->top(layer) !
    weights='imagenet') # weights->ImageNet !

TL_top_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    #tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

```
model.trainable = False  # differences!

TL_top_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_top_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_top_min_imagenet_history = TL_top_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
29089792/29084464 [==============================] - 1s 0us/step
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
densenet121 (Functional)     (None, 1, 1, 1024)        7037504
_________________________________________________________________
global_average_pooling2d (Gl (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 256)               262400
_________________________________________________________________
dense_1 (Dense)              (None, 10)                2570
```

```
=================================================================
Total params: 7,302,474
Trainable params: 264,970
Non-trainable params: 7,037,504
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 43s 20ms/step - loss: 0.7675 - a
Epoch 2/10
1875/1875 [==============================] - 33s 17ms/step - loss: 0.4547 - a
Epoch 3/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.4351 - a
Epoch 4/10
1875/1875 [==============================] - 33s 18ms/step - loss: 0.4202 - a
Epoch 5/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.4128 - a
Epoch 6/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.4025 - a
Epoch 7/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.3913 - a
Epoch 8/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.3857 - a
Epoch 9/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.3786 - a
Epoch 10/10
1875/1875 [==============================] - 32s 17ms/step - loss: 0.3759 - a
```

Plot History

```
plot_history(TL_top_min_imagenet_history.history, 'TL_top_min_imagenet')
```

TL, train->full(model), image=min(32x32), weights=ImageNet

1 epoch ---> 117- sec


```
import tensorflow as tf
import tensorflow_datasets as tfds
#import tensorflow_addons as tfa

# MODEL DEFINITION START #
IMG_SIZE = 32 # for DenseNet121: input size must be at least 32x32
model = tf.keras.applications.DenseNet121(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False, weights='imagenet') # differences!

TL_full_min_imagenet_model = tf.keras.Sequential([
    model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.trainable = True

TL_full_min_imagenet_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

TL_full_min_imagenet_model.summary()

# MODEL DEFINITION END #

# EXTRACT PHASE START #
#data = tfds.load('horses_or_humans', split='train', as_supervised=True)
#val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)
train_data = tfds.load('fashion_mnist', split='train', as_supervised=True)
val_data = tfds.load('fashion_mnist', split='test', as_supervised=True)
# EXTRACT PHASE END

# TRANSFORM PHASE START #
def preprocess_images(image, label):
  ###
  # for fashion_mnist (they are in greyscale),
  # but ALL TF-Keras-models were trained on RGB images from ImageNet
  image = tf.image.grayscale_to_rgb(image)
  ###
  image = tf.cast(image, tf.float32)
```

```
  image = (image/255)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
  #image = tf.image.random_flip_left_right(image)
  #image = tfa.image.rotate(image, 40, interpolation='NEAREST')
  return image, label

train_data = train_data.map(preprocess_images)
train_batches = train_data.shuffle(100).batch(32)
val_data = val_data.map(preprocess_images)
validation_batches = val_data.batch(32)
# TRANSFORM PHASE END

# LOAD PHASE START #
TL_full_min_imagenet_history = TL_full_min_imagenet_model.fit(
    train_batches, epochs=10,
    validation_data=validation_batches, validation_steps=1)
# LOAD PHASE END #
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
densenet121 (Functional)     (None, 1, 1, 1024)        7037504
_________________________________________________________________
global_average_pooling2d_1 ( (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               262400
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                2570
=================================================================
Total params: 7,302,474
Trainable params: 7,218,826
Non-trainable params: 83,648
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 117s 58ms/step - loss: 1.2779 -
Epoch 2/10
 888/1875 [=============>................] - ETA: 54s - loss: 0.5023 - accura
```

Plot History

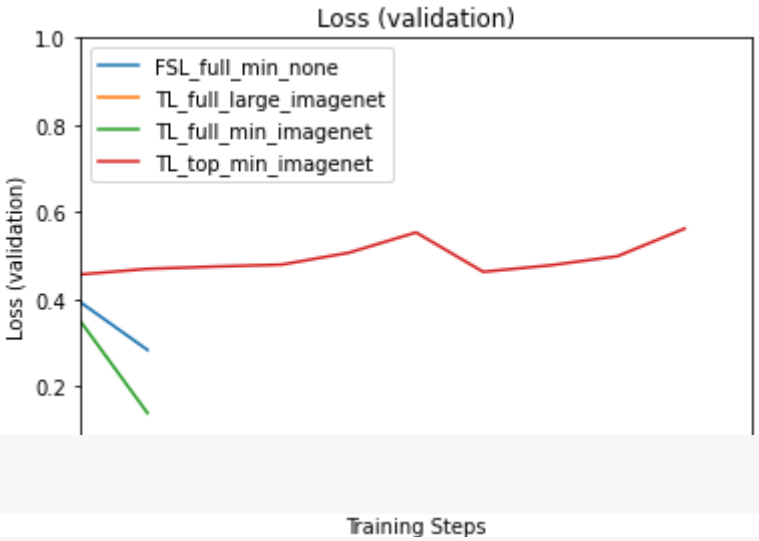
```
plot_history(TL_full_min_imagenet_history.history, 'TL_full_min_imagenet')
```

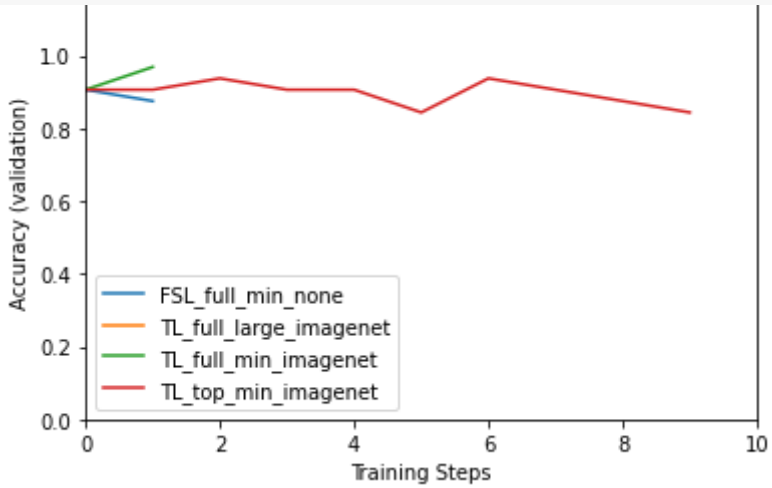
Plot ALL training histories

```
# Plot ALL Training Histories
def plot_all_histories(hist, label):
  plt.figure()
  plt.ylabel("Loss (validation)")
```

```
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
    plt.plot(h["val_loss"], label=l)
  plt.title('Loss (validation)')
  plt.legend(loc=2)

  plt.figure()
  plt.ylabel("Accuracy (validation)")
  plt.xlabel("Training Steps")
  plt.xlim([0,10])
  plt.ylim([0,1.2])
  for h,l in zip(hist,label):
    #plt.plot(hist["loss"])
    plt.plot(h["val_accuracy"], label=l)
  plt.title('Accuracy (validation)')
  plt.legend(loc=3)
```

```
hist = [
        FSL_full_min_none_history.history,
        #TL_full_large_imagenet_history.history,
        TL_full_min_imagenet_history.history,
        TL_top_min_imagenet_history.history
        ]

label = [
        'FSL_full_min_none',
        #'TL_full_large_imagenet',
        'TL_full_min_imagenet',
        'TL_top_min_imagenet'
        ]

plot_all_histories(hist, label)
```





Нейронні мережі

Лекція_09

********** **********

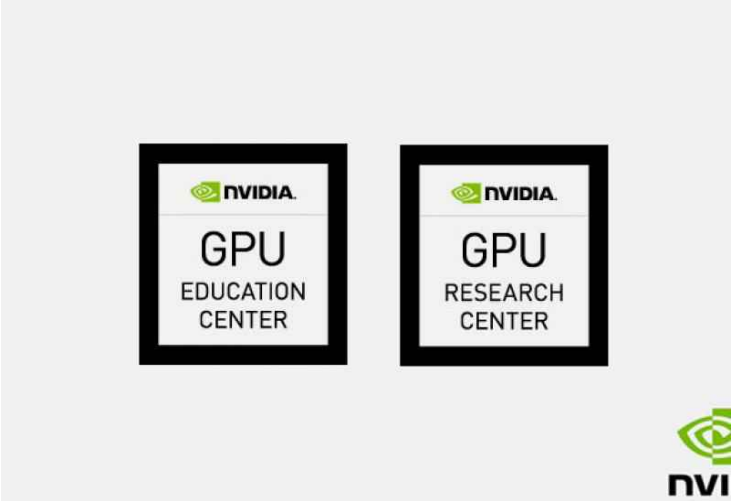
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 09- Нейронні мережі -Сучасні CNN –Мистецтво, стиль, якість

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1
Приклади застосування методів «Fancy» (Arty-Farty).
https://drive.google.com/file/d/1FkwT8cGkgbqyGdqokqw8rSVVmPI_dwve/view?usp=sharing

Lecture 09. Modern CNN: "Fancy" (Arty-Farty) Applications

(C) partially based on the works by d2l Open Source book authors, TFHub contributors, KPI look paparazzi, Planche, Martinez, Sarang, Carim, ...

To look at the **nature and influence of different components** of DNN-CNNs like:

- layers,
- filters,
- combinations of DNN-CNNs,
- ... let's consider some modern fancy examples of their applications, **except for classification**.

Part 0. Resume on the modern CNNs

Below, let's summarize the most significant contributions introduced by these researchers while further detailing their architecture.

Deep Learning Models Evolution driven by ILSVRC

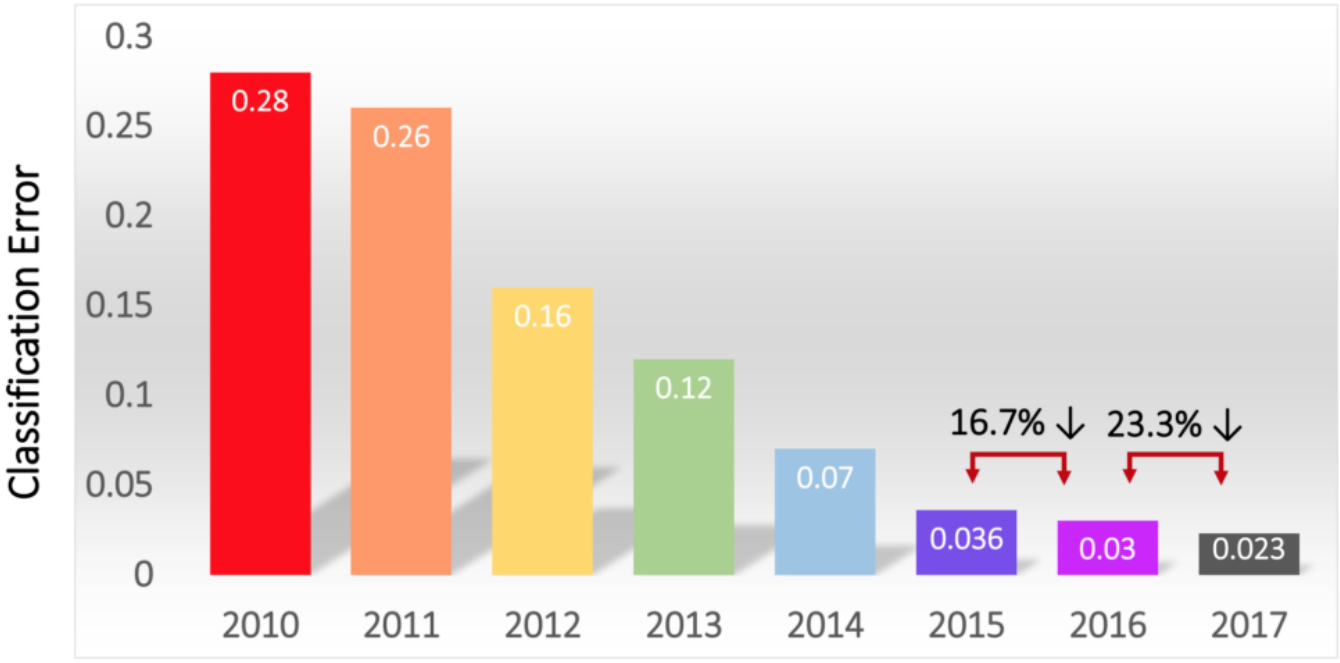
The pace of improvement in the first 5 years of the ILSVRC was dramatic, perhaps even shocking to the field of computer vision.

Success has primarily been achieved by large (deep) convolutional neural networks (CNNs) on graphical processing unit (GPU) hardware, which sparked an interest in deep learning that extended beyond the field out into the mainstream.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

The general challenge tasks for most years are as follows:

- **Image classification**: Predict the classes of objects present in an image.
- **Single-object localization**: Image classification + draw a bounding box around one example of each object present - LATER about this, NOT now.
- **Object detection**: Image classification + draw a bounding box around each object present - LATER about this, NOT now.

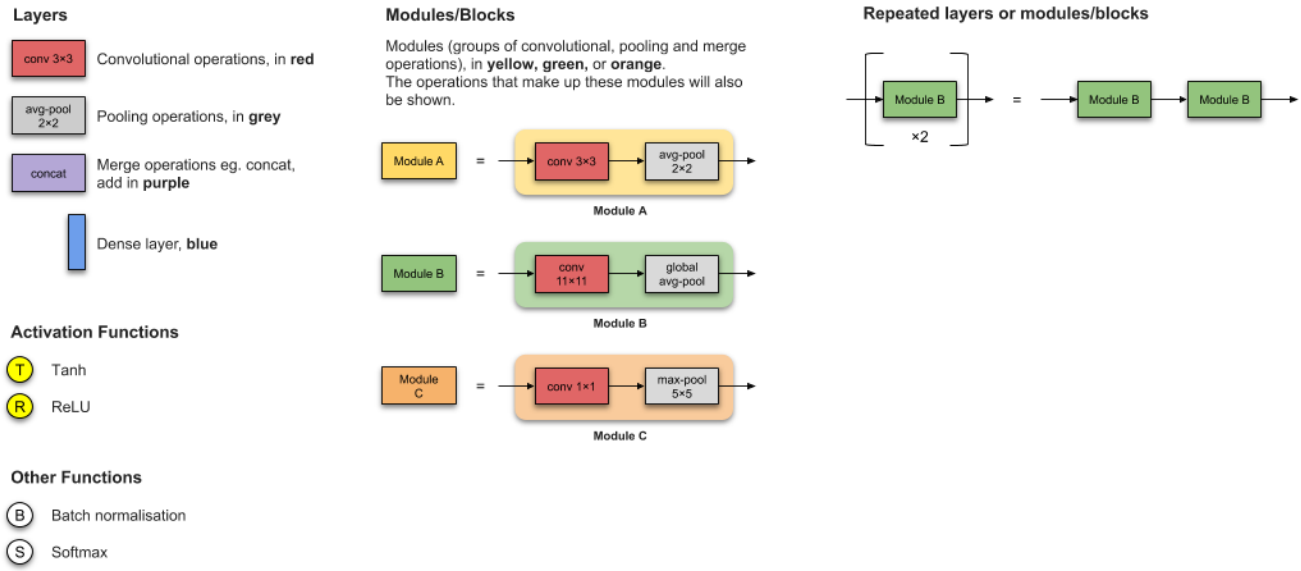


ImageNet Dataset

ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowd-sourcing tool. Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. In all, there are roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images.

Let's consider evolution of DNN-CNN through ILSVRC ... for some models.

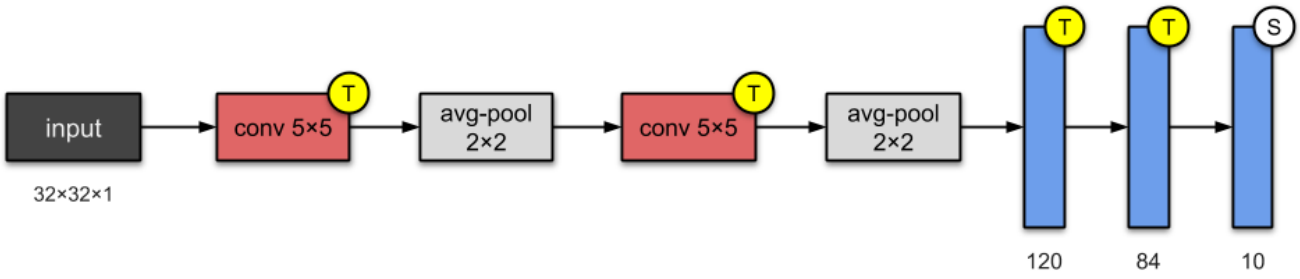
Legend for diagrams (C) Raimi Carim



LeNet-5 (1998)

LeNet-5 is one of the simplest architectures. It has 2 convolutional and 3 fully-connected layers (hence “5” — it is very common for the names of neural networks to be derived from the number of convolutional and fully connected layers that they have). The average-pooling layer as we know it now was called a sub-sampling layer and it had trainable weights (which isn’t the current practice of designing CNNs nowadays).

LeNet-5 architecture.



Parameters: 0.06M

Size: ~5MB

Novelty:

- stacking convolutions with activation function,
- pooling layers,
- one or more fully-connected layers in end of the network.

Publication:

Title: Gradient-Based Learning Applied to Document Recognition

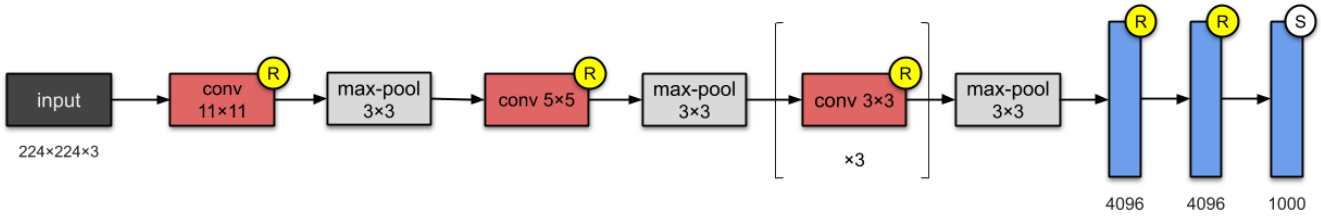
Authors: Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Published in: Proceedings of the IEEE (1998)

AlexNet (2012)

AlexNet has 8 layers — 5 convolutional and 3 fully-connected. AlexNet just stacked a few more layers onto LeNet-5. At the point of publication, the authors pointed out that their architecture was “one of the largest convolutional neural networks to date on the subsets of ImageNet.”

AlexNet architecture.



Parameters: 60M

Size: ~200MB

Novelty:

- Rectified Linear Units (ReLUs) as activation functions,
- dropout.

Publication:

Title: ImageNet Classification with Deep Convolutional Neural Networks

Authors: Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton

Published in: NeurIPS (2012)

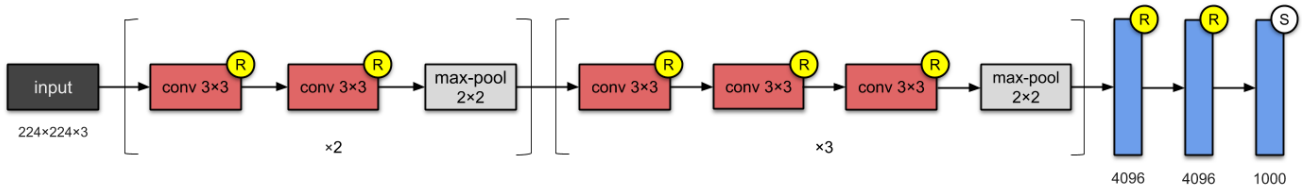
VGG (2014)

The authors of VGG actually introduced six different CNN architectures, from 11 to 25 layers deep. Each network is composed of five blocks of several consecutive convolutions followed by a max-pooling layer and three final dense layers (with dropout for training). All the convolutional and max-pooling layers have SAME for padding. The convolutions have s = 1 for stride, and are using the ReLU function for activation.

The two most performant architectures, still commonly used nowadays, are called VGG-16 and VGG-19. The numbers (16 and 19) represent the depth of these CNN architectures; that is, the number of trainable layers stacked together. For example, as shown in Figure 4.1, VGG-16 contains 13 convolutional layers and 3 dense ones, hence a depth of 16 (excluding the non-trainable operations; that is, the 5 max-pooling and 2 dropout layers). The same goes for VGG-

19, which is composed of three additional convolutions. VGG-16 has approximately 138 million parameters, and VGG-19 has 144 million.

VGG-16 architecture:



Parameters: 138M

Size: ~500MB

Novelty:

SHORT:

- Design the deeper networks (roughly twice as deep as AlexNet) by stacking uniform convolutions.

LONG:

- **Replace large convolutions with multiple smaller ones**

  The authors began with a simple observation—a stack of two convolutions with 3 × 3 kernels has the same **effective receptive field (ERF)** as a convolution with 5 × 5 kernels, because it decreases:

  - the number of parameters
  - the non-linearity

- **Increase the depth of the feature maps**

  Based on another intuition, the VGG authors doubled the depth of the feature maps for each block of convolutions (from 64 after the first convolution to 512). As each set is followed by a max-pooling layer with a 2 × 2 window size and a stride of 2, the depth doubles while the spatial dimensions are halved. This allows the encoding of spatial information into more and more complex and discriminative features for classification.

- **Augmenting data with scale jittering**

  The authors also introduced a data augmentation mechanism that they named **scale jittering**. At each training iteration, they randomly scale the batched images (from 256 pixels to 512 pixels for their smaller side) before cropping them to the proper input size (224 × 224 for their ILSVRC submission).

- **Replace fully connected layers with convolutions**

  While the classic VGG architecture ends with several fully connected (FC) layers (such as AlexNet), the authors suggest an alternative version. In this version, the dense layers are

replaced by convolutional ones.

The first set of convolutions with larger kernels (7 × 7 and 3 × 3) reduces the spatial size of the feature maps to 1 × 1 (with no padding applied beforehand) and increases their depth to 4,096. Finally, a 1 × 1 convolution is used with as many filters as classes to predict from (that is, N = 1,000 for ImageNet). The resulting 1 × 1 × N vector is normalized with the softmax function, and then flattened into the final class predictions (with each value of the vector representing the predicted class probability).

1 × 1 convolutions are commonly used to change the depth of the input volume without affecting its spatial structure. For each spatial position, the new values are interpolated from all the depth values at that position.

Such a network without any dense layers is called a **fully convolutional network (FCN)**. FCNs can be applied to images of different sizes, with no need for cropping beforehand.

- **Model averaging**

  To achieve the best accuracy for ILSVRC, the authors trained and used both versions (normal and FCN), once again averaging their results to obtain the final predictions. This technique is named **model averaging** and is frequently used in production.

Publication:

Title: Very Deep Convolutional Networks for Large-Scale Image Recognition

Authors: Karen Simonyan, Andrew Zisserman

Published in: arXiv preprint (2014)

Inception-v1 (2014)

Developed by researchers at Google, the architecture we will now present was also applied to ILSVRC 2014 and won first place for the classification task ahead of VGGNet. GoogLeNet (for Google and LeNet, as an homage to this pioneering network) is structurally very different from its linear challenger, introducing the notion of inception blocks (the network is also commonly called an inception network).

This 22-layer architecture with 5M parameters is called the Inception-v1. Here, the Network In Network (see previous lecture) approach is heavily used, as mentioned in the paper. This is done by means of 'Inception modules'. The design of the architecture of an Inception module is a product of research on approximating sparse structures (read paper for more!).

Each module presents 3 ideas:

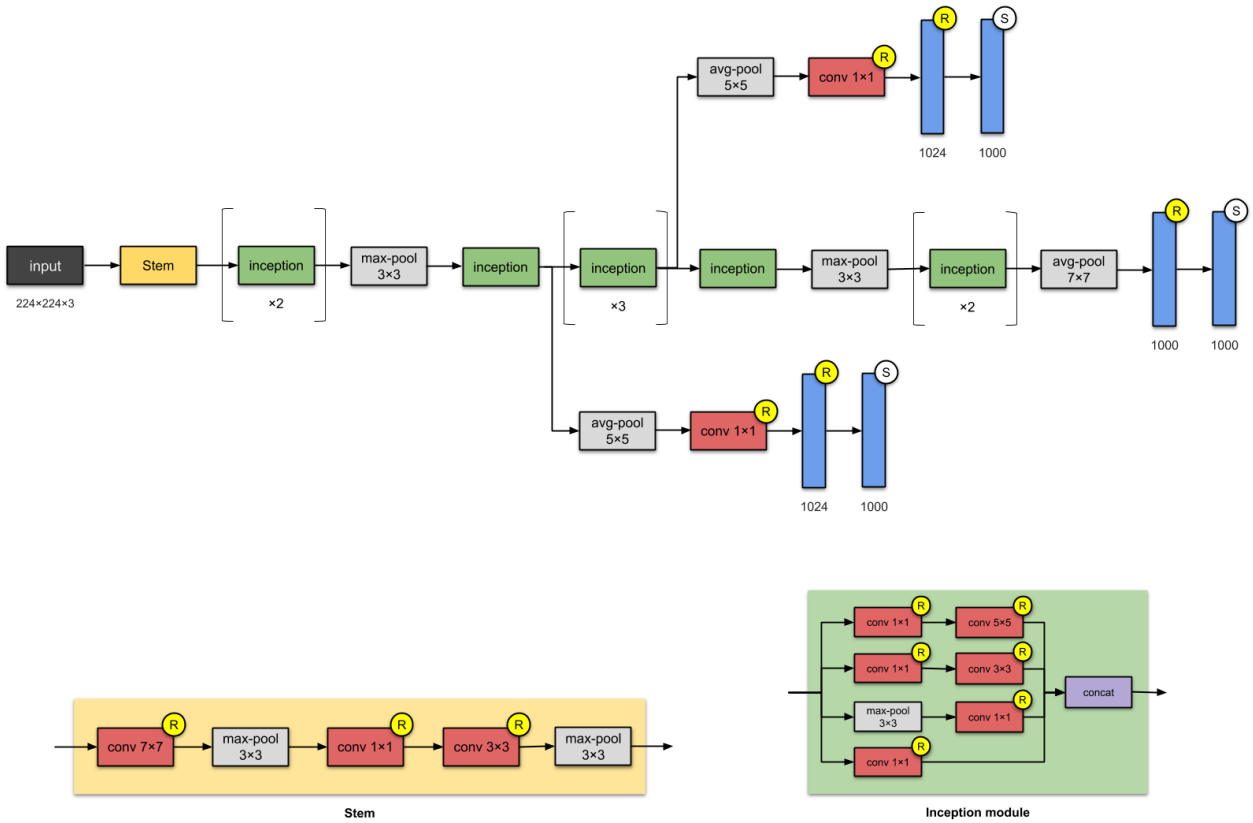
- Having parallel towers of convolutions with different filters, followed by concatenation, captures different features at 1×1, 3×3 and 5×5, thereby 'clustering' them. This idea is motivated by Arora et al. in the paper Provable bounds for learning some deep

representations, suggesting a layer-by layer construction in which one should analyse the correlation statistics of the last layer and cluster them into groups of units with high correlation.

- 1×1 convolutions are used for dimensionality reduction to remove computational bottlenecks.
- Due to the activation function from 1×1 convolution, its addition also adds nonlinearity. This idea is based on the Network In Network paper. See appendix here.
- The authors also introduced two auxiliary classifiers to encourage discrimination in the lower stages of the classifier, to increase the gradient signal that gets propagated back, and to provide additional regularisation. The auxiliary networks (the branches that are connected to the auxiliary classifier) are discarded at inference time.

It is worth noting that “the main hallmark of this architecture is the improved utilisation of the computing resources inside the network.”

Inception architecture:



Note: The names of the modules (Stem and Inception) were not used for this version of Inception until its later versions i.e. Inception-v4 and Inception-ResNets. They are added here for easy comparison.

Parameters: ~20-50M

Size: ~100MB

Novelty:

- Building networks using modules/blocks.

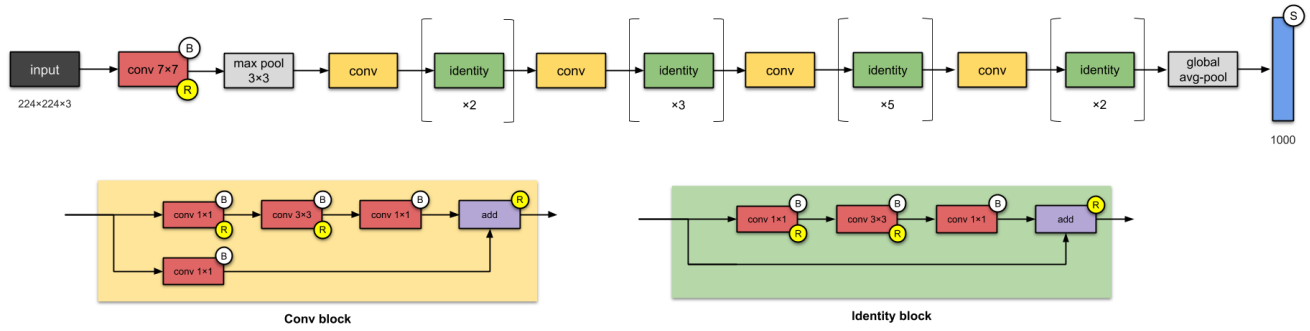
  Instead of stacking convolutional layers, we stack modules or blocks, within which are convolutional layers. Hence the name Inception (with reference to the 2010 sci-fi movie Inception starring Leonardo DiCaprio).

Publication:

Paper: Going Deeper with Convolutions *Authors:* Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Google, University of Michigan, University of North Carolina *Published in:* 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

ResNet-50 (2015)

From the past few CNNs, we have seen nothing but an increasing number of layers in the design, and achieving better performance. But "with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly." The folks from Microsoft Research addressed this problem with ResNet — using skip connections (a.k.a. shortcut connections, residuals), while building deeper models.



ResNet is one of the early adopters of batch normalisation (the batch norm paper authored by Ioffe and Szegedy was submitted to ICML in 2015). Shown above is ResNet-50, with 26M parameters.

The basic building block for ResNets are the conv and identity blocks. Because they look alike, you might simplify ResNet-50 like this (don't quote me for this!):

Novelty:

- Popularised skip connections (they weren't the first to use skip connections).
- Designing even deeper CNNs (up to 152 layers) without compromising model's generalisation power
- Among the first to use batch normalisation.

Publication:

Paper: Deep Residual Learning for Image Recognition

Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Microsoft

Published in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Part 1. Neural Style Transfer (NST)

Again ... to look at the **nature and influence of different components** of DNN-CNNs like:

- layers,
- filters,
- combinations of DNN-CNNs,
- ... let's consider some modern fancy examples of their applications, **except for classification**.

(C) partially based on 13.12. Neural Style Transfer from d2l Open Source book.

If you use social sharing apps or happen to be an amateur photographer, you are familiar with filters. Filters can alter the color styles of photos to make the background sharper or people's faces whiter. However, a filter generally can only change one aspect of a photo. To create the ideal photo, you often need to try many different filter combinations. This process is as complex as tuning the hyperparameters of a model.

In this section, we will discuss how we can use convolution neural networks (CNNs) to automatically apply the style of one image to another image, an operation known as style transfer Gatys et al., 2016. Here, we need two input images:

- one **content** image,
- one **style** image.

We use a neural network to alter the content image so that its style mirrors that of the style image. In Fig. 13.12.1, the content image is a landscape photo the author took in Mount Rainier National Part near Seattle. The style image is an oil painting of oak trees in autumn. The output composite image retains the overall shapes of the objects in the content image, but applies the oil painting brushwork of the style image and makes the overall color more vivid.

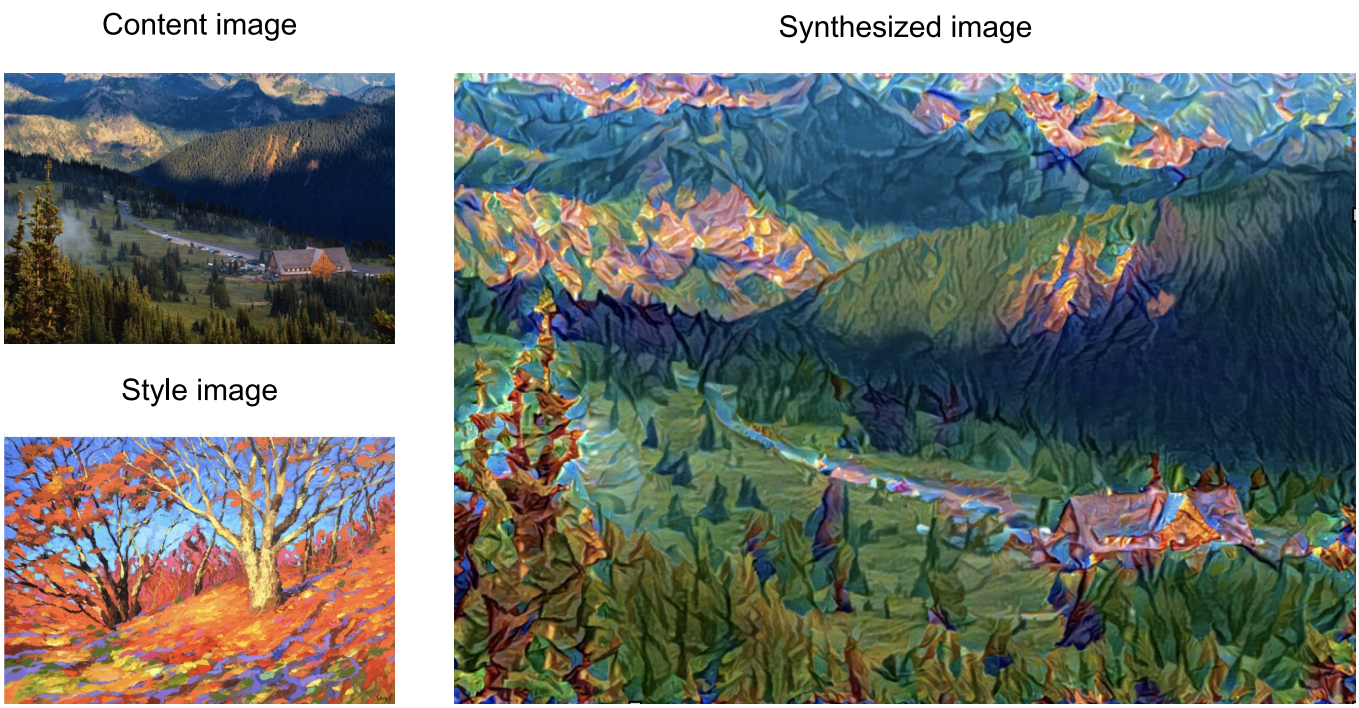


Fig. 13.12.1. Content and style input images and composite image produced by style transfer

Technique

The CNN-based style transfer model is shown in `Fig. 13.12.2`.

- First, we initialize the composite image. For example, we can initialize it as the content image. This composite image is the only variable that needs to be updated in the style transfer process, i.e., the model parameter to be updated in style transfer.
- Then, we select a pre-trained CNN to extract image features. These model parameters do not need to be updated during training. The deep CNN uses multiple neural layers that successively extract image features. We can select the output of certain layers to use as content features or style features. If we use the structure in `Fig. 13.12.2`, the pre-trained neural network contains three convolutional layers. The second layer outputs the image content features, while the outputs of the first and third layers are used as style features. Next, we use forward propagation (in the direction of the solid lines) to compute the style transfer loss function and backward propagation (in the direction of the dotted lines) to update the model parameter, constantly updating the composite image.

The **loss** functions used in style transfer generally have three parts:

1. **Content** loss is used to make the composite image approximate the content image as regards content features.
2. **Style** loss is used to make the composite image approximate the style image in terms of style features.
3. **Total variation (TV)** loss helps reduce the noise in the composite image.

Finally, after we finish training the model, we output the style transfer model parameters to obtain the final composite image.

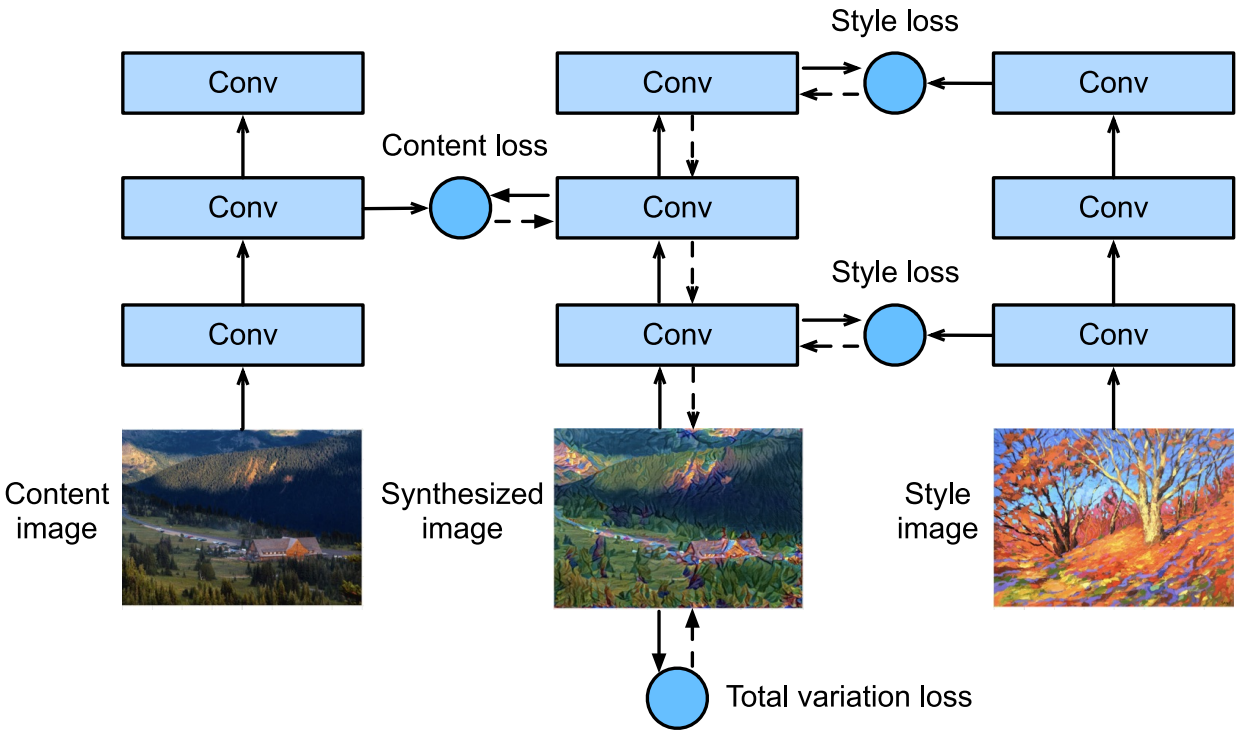


Fig. 13.12.2. CNN-based style transfer process.

Solid lines show the direction of forward propagation and dotted lines show backward propagation.

Next, we will perform two experiments to help us better understand the technical details of style transfer based on:

- PyTorch (from `d2l.ai`),
- TensorFlow.

Preparatory Actions

[ ] ↳ *6 cells hidden*

NST - PyTorch Example

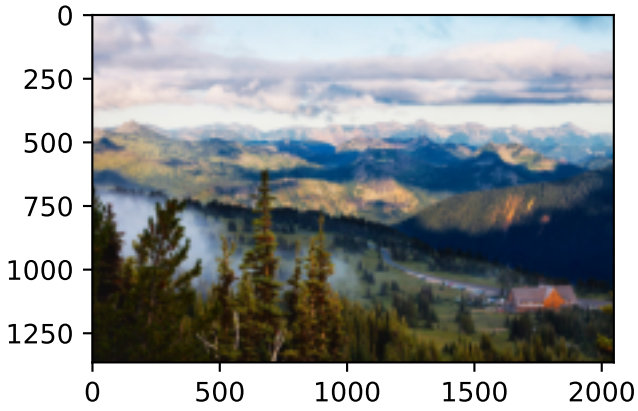
```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

Content and Style Images

First, we read the content and style images. By printing out the image coordinate axes, we can see that they have different dimensions.

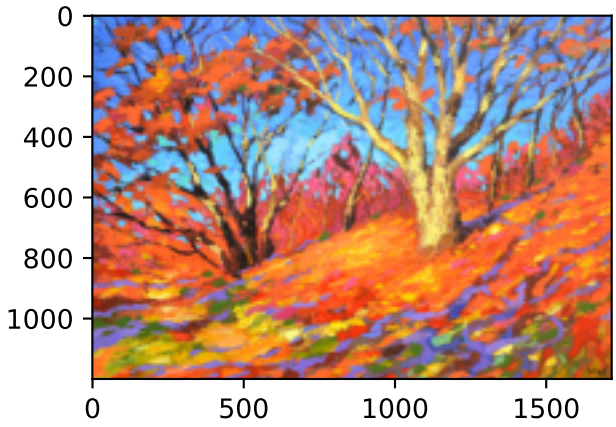
Content Image

```
d2l.set_figsize()
content_img = d2l.Image.open('./figures/rainier.jpg')
d2l.plt.imshow(content_img);
```



Style Image

```
style_img = d2l.Image.open('./figures/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```



Preprocessing and Postprocessing

Below, we define the functions for image preprocessing and postprocessing. The `preprocess` function normalizes each of the three RGB channels of the input images and transforms the results to a format that can be input to the CNN. The `postprocess` function restores the pixel

values in the output image to their original values before normalization. Because the image printing function requires that each pixel has a floating point value from 0 to 1, we use the `clip` function to replace values smaller than 0 or greater than 1 with 0 or 1, respectively.

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

Extracting Features (layers) for Content and Style

We use the VGG-19 model pre-trained on the ImageNet dataset to extract image features described in the classic paper:

Simonyan, K., & Zisserman, A., Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556 (2014).

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

To extract image content and style features, we can select the outputs of certain layers in the VGG network. In general, the closer an output is to the input layer, the easier it is to extract image detail information. The farther away an output is, the easier it is to extract global information. To prevent the composite image from retaining too many details from the content image, we select a VGG network layer near the output layer to output the image content features. This layer is called the content layer. We also select the outputs of different layers from the VGG network for matching local and global styles. These are called the style layers. As we mentioned in :numref: `sec_vgg` , VGG networks have five convolutional blocks. In this experiment, we select the last convolutional layer of the fourth convolutional block as the content layer and the first layer of each block as style layers. We can obtain the indexes for these layers by printing the `pretrained_net` instance.

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

During feature extraction, we only need to use all the VGG layers from the input layer to the content or style layer nearest the output layer. Below, we build a new network, `net` , which only

retains the layers in the VGG network we need to use. We then use `net` to extract features.

Build Model

```
net = nn.Sequential(*[
    pretrained_net.features[i]
    for i in range(max(content_layers + style_layers) + 1)])
```

Given input `X`, if we simply call the forward computation `net(X)`, we can only obtain the output of the last layer. Because we also need the outputs of the intermediate layers, we need to perform layer-by-layer computation and retain the content and style layer outputs.

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

Next, we define two functions: The `get_contents` function obtains the content features extracted from the content image, while the `get_styles` function obtains the style features extracted from the style image. Because we do not need to change the parameters of the pre-trained VGG model during training, we can extract the content features from the content image and style features from the style image before the start of training. As the composite image is the model parameter that must be updated during style transfer, we can only call the `extract_features` function during training to extract the content and style features of the composite image.

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

Loss Function

Next, we will look at the loss function used for style transfer.

The loss function includes:

- content loss,
- style loss,
- total variation (TV) loss.

Content Loss

Similar to the loss function used in linear regression, content loss uses a square error function to measure the difference in content features between the composite image and content image. The two inputs of the square error function are both content layer outputs obtained from the `extract_features` function.

```
def content_loss(Y_hat, Y):
    # we 'detach' the target content from the tree used
    # to dynamically compute the gradient: this is a stated value,
    # not a variable. Otherwise the loss will throw an error.
    return torch.square(Y_hat - Y.detach()).mean()
```

Style Loss

Style loss, similar to content loss, uses a square error function to measure the difference in style between the composite image and style image. To express the styles output by the style layers, we first use the `extract_features` function to compute the style layer output. Assuming that the output has 1 example, c channels, and a height and width of h and w, we can transform the output into the matrix $\mathbf{X}$, which has c rows and $h \cdot w$ columns. You can think of matrix $\mathbf{X}$ as the combination of the c vectors $\mathbf{x}_1, \ldots, \mathbf{x}_c$, which have a length of hw. Here, the vector $\mathbf{x}_i$ represents the style feature of channel i. In the Gram matrix of these vectors $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$, element x_{ij} in row i column j is the inner product of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. It represents the correlation of the style features of channels i and j. We use this type of Gram matrix to represent the style output by the style layers. You must note that, when the $h \cdot w$ value is large, this often leads to large values in the Gram matrix. In addition, the height and width of the Gram matrix are both the number of channels c. To ensure that the style loss is not affected by the size of these values, we define the `gram` function below to divide the Gram matrix by the number of its elements, i.e., $c \cdot h \cdot w$.

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

Naturally, the two Gram matrix inputs of the square error function for style loss are taken from the composite image and style image style layer outputs. Here, we assume that the Gram matrix of the style image, `gram_Y`, has been computed in advance.

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

Total Variance (TV) Loss

Sometimes, the composite images we learn have a lot of high-frequency noise, particularly bright or dark pixels. One common noise reduction method is total variation denoising. We assume that $x_{i,j}$ represents the pixel value at the coordinate (i, j), so the total variance loss is:

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|.$$

We try to make the values of neighboring pixels as similar as possible.

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

Loss Function

The loss function for style transfer is the weighted sum of the content loss, style loss, and total variance loss. By adjusting these weight hyperparameters, we can balance the retained content, transferred style, and noise reduction in the composite image according to their relative importance.

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [
        content_loss(Y_hat, Y) * content_weight
        for Y_hat, Y in zip(contents_Y_hat, contents_Y)]
    styles_l = [
        style_loss(Y_hat, Y) * style_weight
        for Y_hat, Y in zip(styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

Creating and Initializing the Composite Image

In style transfer, the composite image is the only variable that needs to be updated. Therefore, we can define a simple model, `GeneratedImage`, and treat the composite image as a model parameter. In the model, forward computation only returns the model parameter.

```
class GeneratedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(GeneratedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

Next, we define the `get_inits` function. This function creates a composite image model instance and initializes it to the image `X`. The Gram matrix for the various style layers of the style image, `styles_Y_gram`, is computed prior to training.

```
def get_inits(X, device, lr, styles_Y):
    gen_img = GeneratedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

Training - Generating Composite (Stylized) image

During model training, we constantly extract the content and style features of the composite image and calculate the loss function. Recall our discussion of how synchronization functions force the front end to wait for computation results in :numref:`sec_async`. Because we only call the `asnumpy` synchronization function every 10 epochs, the process may occupy a great deal of memory. Therefore, we call the `waitall` synchronization function during every epoch.

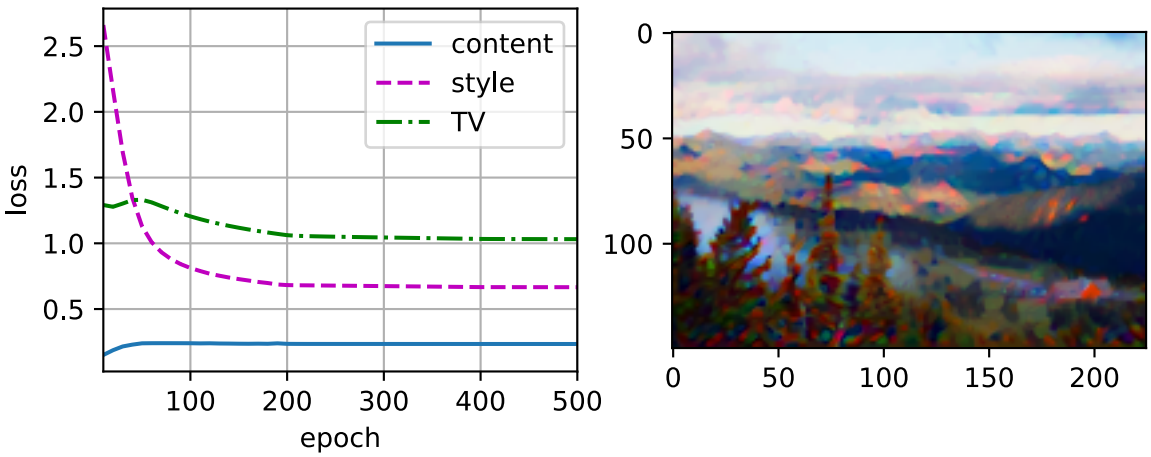
```
from tqdm import tqdm

def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style',
                                    'TV'], ncols=2, figsize=(7, 2.5))
    #for epoch in tqdm(range(num_epochs)):
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(X, contents_Y_hat,
```

```
                                                styles_Y_hat, contents_Y,
                                                styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(
                epoch + 1,
                [float(sum(contents_l)),
                 float(sum(styles_l)),
                 float(tv_l)])
    return X
```

Next, we start to train the model. First, we set the height and width of the content and style images to 150 by 225 pixels. We use the content image to initialize the composite image.

```
%%time
# Wall time (CPU): 12 min 48 sec.
# Wall time (GPU): 36 sec.

device, image_shape = d2l.try_gpu(), (150, 225)  # PIL Image (h, w)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.01, 500, 200)
```

```
CPU times: user 29.7 s, sys: 5.84 s, total: 35.6 s
Wall time: 36.1 s
```

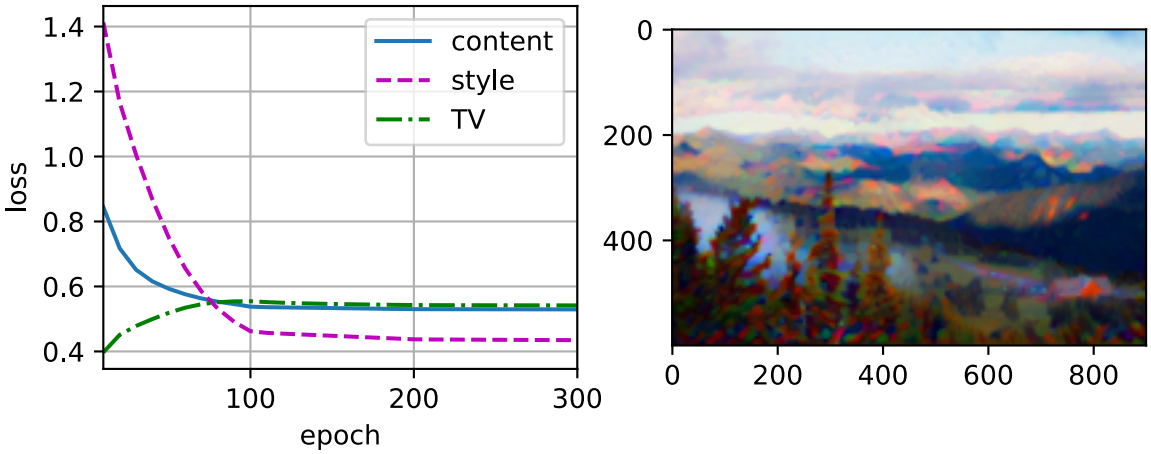


As you can see, the composite image retains the scenery and objects of the content image, while introducing the color of the style image. Because the image is relatively small, the details are a bit fuzzy.

To obtain a clearer composite image, we train the model using a larger image size: 900×600. We increase the height and width of the image used before by a factor of four and initialize a larger composite image.

```
%%time
# Wall time CPU (approx.): 2 h 30 min - forget about this and try GPU :)
# Wall time GPU: 1 min 52 sec.

image_shape = (600, 900)  # PIL Image (h, w)
_, content_Y = get_contents(image_shape, device)
_, style_Y = get_styles(image_shape, device)
X = preprocess(postprocess(output), image_shape).to(device)
output = train(X, content_Y, style_Y, device, 0.01, 300, 100)
d2l.plt.imsave('./neural-style.jpg', postprocess(output))
```

```
CPU times: user 1min 9s, sys: 41.3 s, total: 1min 51s
Wall time: 1min 52s
```



As you can see, each epoch takes more time due to the larger image size. As shown in `Fig. 13.12.3`, the composite image produced retains more detail due to its larger size. The composite image not only has large blocks of color like the style image, but these blocks even have the subtle texture of brush strokes.

@@0@@ composite image.

Fig. 13.12.3. The final 900×600 composite image (after successful completion of the previous script).

NST - TensorFlow Example

Restart Runtime!

Again, use VGG16 model to extract features and define your own network for style transfer.

Import Libraries

```
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
import re
```

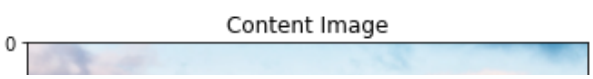
```
import urllib
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from matplotlib import pyplot as plt
from IPython import display
from PIL import Image
import numpy as np
from tensorflow.keras.applications import vgg16
#from tensorflow.keras import backend as K
from keras import backend as K
from scipy.optimize import fmin_l_bfgs_b
```

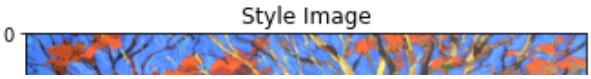
Content and Style Images

From d2l Textbook

```
# Dimensions for the generated picture.
target_path = './figures/rainier.jpg'
style_path = './figures/autumn-oak.jpg'

width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)
```

```
content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```

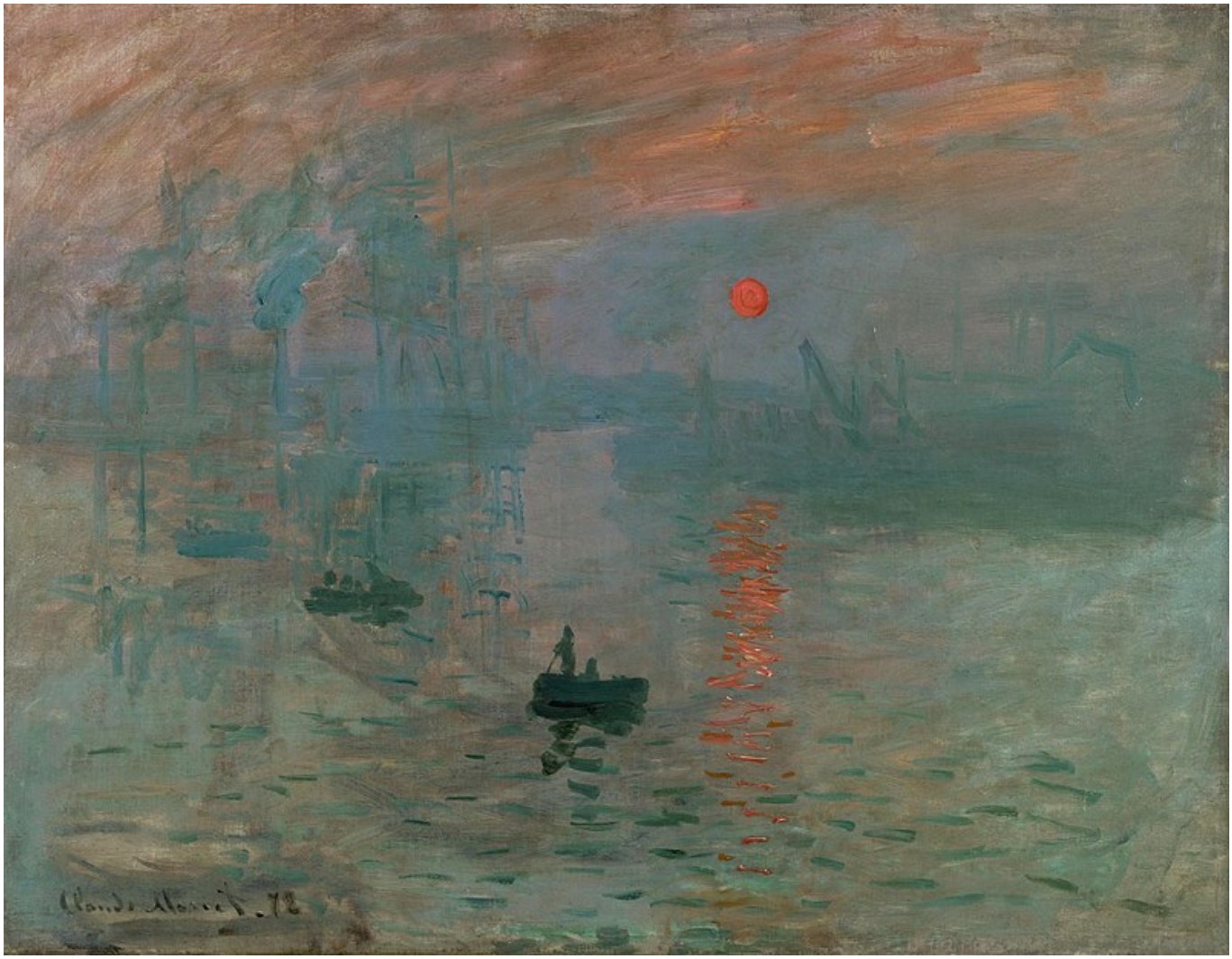




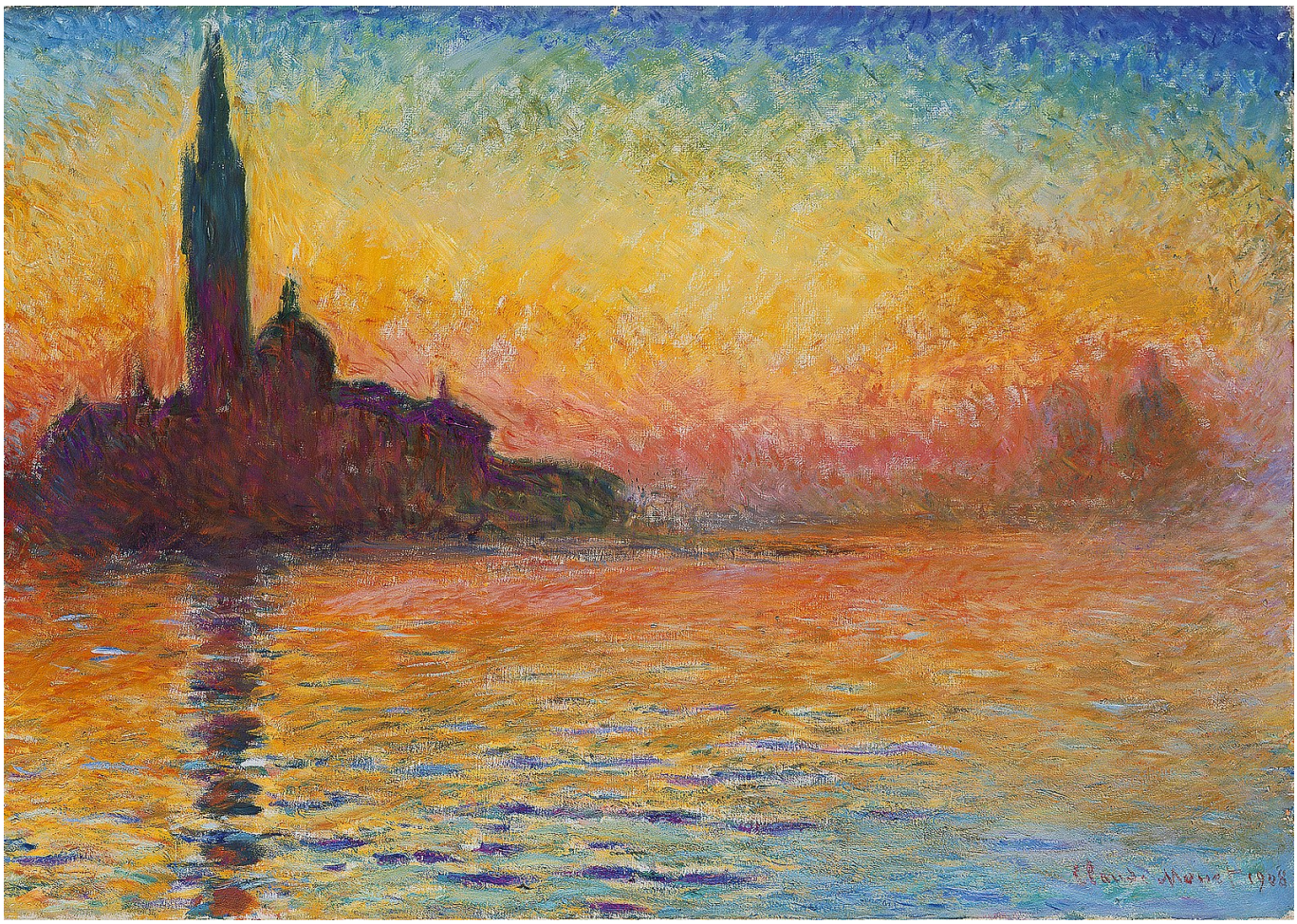
KPI look 1 + Monet

Oscar-Claude Monet (UK: /ˈmɒneɪ/, US: /moʊˈneɪ/, French: [klod mɔnɛ]; 14 November 1840 – 5 December 1926) was a French painter, a founder of French Impressionist painting and the most consistent and prolific practitioner of the movement's philosophy of expressing one's perceptions before nature, especially as applied to plein air landscape painting.

The term "Impressionism" is derived from the title of his painting Impression, soleil levant (Impression, Sunrise), which was exhibited in 1874 in the first Salon des Refusés (exhibition of rejects) mounted by Monet and his associates as an alternative to the Salon de Paris.



Between 1883 and 1908, Monet traveled to the Mediterranean, where he painted landmarks, landscapes, and seascapes, including a series of paintings in Venice.



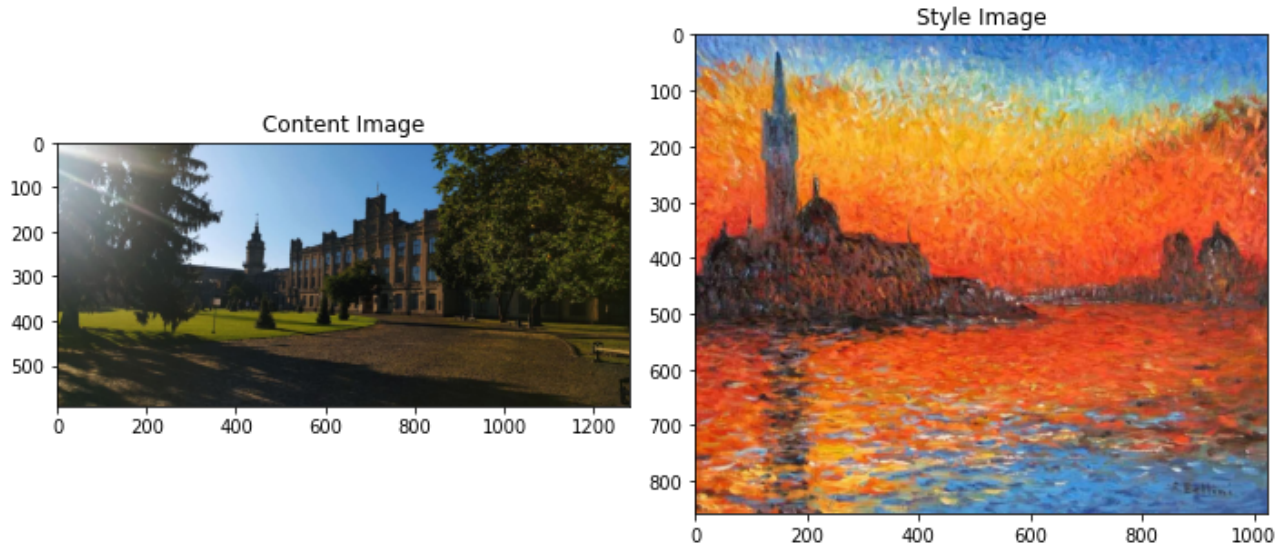
San Giorgio Maggiore at Dusk

In London he painted four series: the Houses of Parliament, London, Charing Cross Bridge, Waterloo Bridge, and Views of Westminster Bridge.

```
# Dimensions for the generated picture.

target_path = './figures/KPI_look_1_sunny.jpg'
style_path = './figures/monet.jpg'

width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)
```

```
content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')
```

```
plt.tight_layout()
plt.show()
```



KPI look UNKNOWN hero + Pollock-style

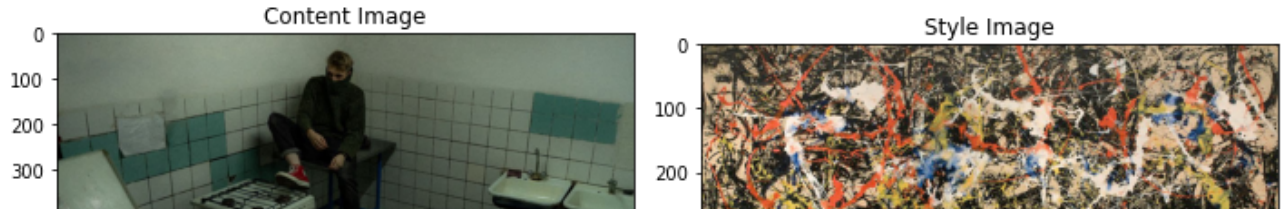
Paul Jackson Pollock /ˈpɒlək/ (January 28, 1912 – August 11, 1956) was an American painter and a major figure in the abstract expressionist movement.

He was widely noticed for his technique of pouring or splashing liquid household paint onto a horizontal surface ("drip technique"), enabling him to view and paint his canvases from all angles. It was also called all-over painting and "action painting", since he covered the entire canvas and used the force of his whole body to paint, often in a frenetic dancing style. This extreme form of abstraction divided the critics: some praised the immediacy of the creation, while others derided the random effects.

In 2015, it was sold by the David Geffen Foundation to Kenneth C. Griffin for $200 million.

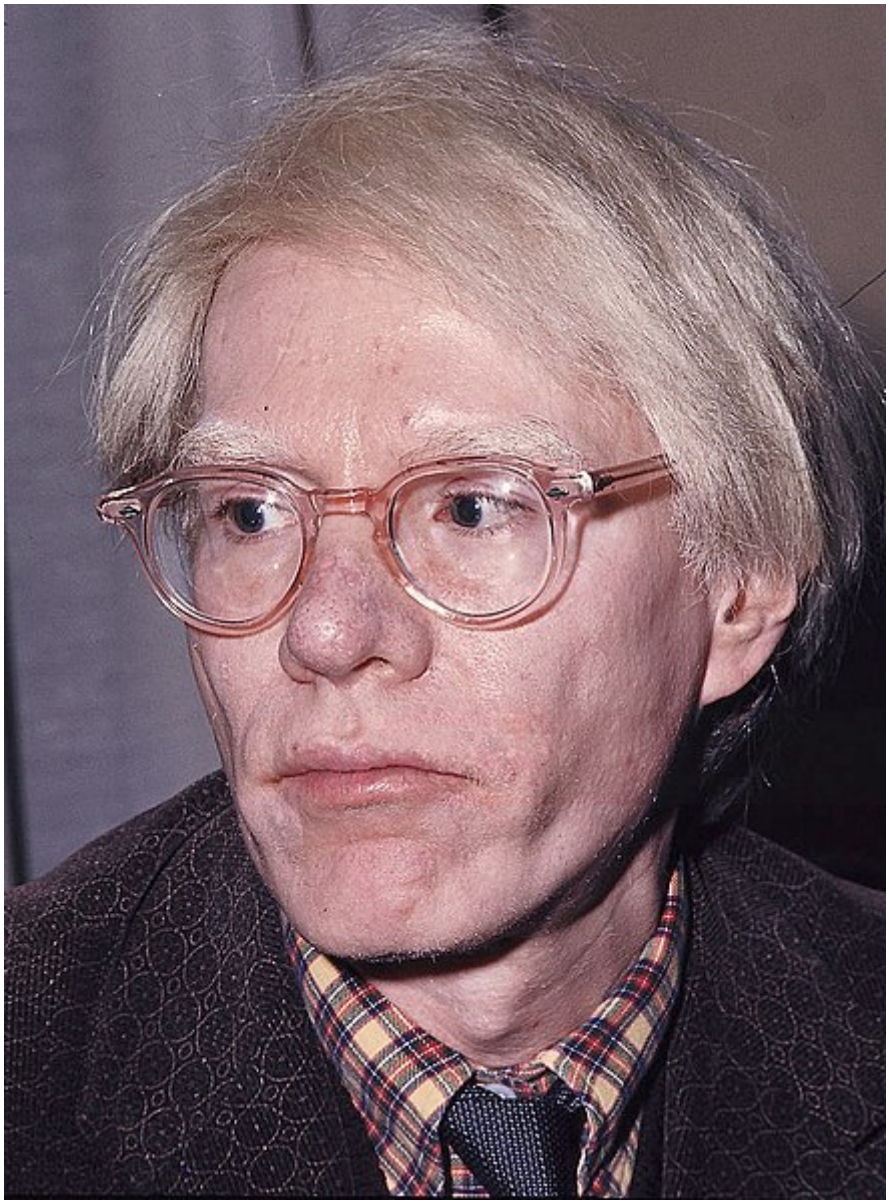
```
# Dimensions for the generated picture.

target_path = './figures/KPI_look_UNK.jpg'
style_path = './figures/pollock.jpg'

width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)
```

```
##Displaying Content and Style images

content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```



KPI look 18 wild + Warhol-style

Andy Warhol (/ˈwɔːrhɒl/;[1] born Andrew Warhola; August 6, 1928 – February 22, 1987) was an American artist, film director, and producer who was a leading figure in the visual art movement known as pop art. His works explore the relationship between artistic expression, advertising, and celebrity culture that flourished by the 1960s, and span a variety of media, including painting, silkscreening, photography, film, and sculpture.



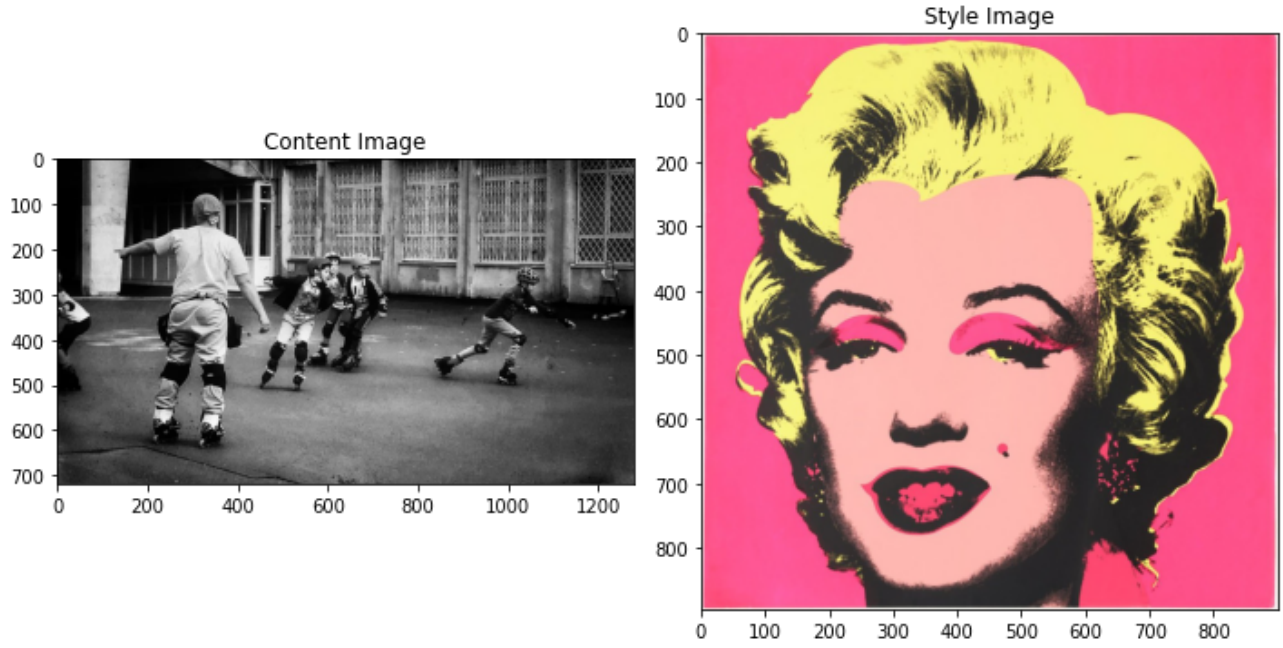
Some of his best known works include the silkscreen paintings Campbell's Soup Cans (1962) and Marilyn Diptych (1962), the experimental films Empire (1964) and Chelsea Girls (1966), and the multimedia events known as the Exploding Plastic Inevitable (1966–67).

A pivotal event was the 1964 exhibit The American Supermarket, a show held in Paul Bianchini's Upper East Side gallery. The show was presented as a typical U.S. small supermarket environment, except that everything in it—from the produce, canned goods, meat, posters on the wall, etc.—was created by six prominent pop artists of the time, among them the controversial (and like-minded) Billy Apple, Mary Inman, and Robert Watts. Warhol's painting of a can of Campbell's soup cost $1500 while each autographed can sold for $6. The exhibit was one of the first mass events that directly confronted the general public with both pop art and the perennial question of what art is.

```
# Dimensions for the generated picture.

target_path = './figures/KPI_look_18_wild.jpg'
style_path = './figures/warhol.jpg'

width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)
```

```
##Displaying Content and Style images

content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')
```

```
plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```



KPI look 18 + Kusama-style

Yayoi Kusama (草間 彌生, Kusama Yayoi, born 22 March, 1929) is a Japanese contemporary artist who works primarily in sculpture and installation, but is also active in painting, performance, film, fashion, poetry, fiction, and other arts.

.

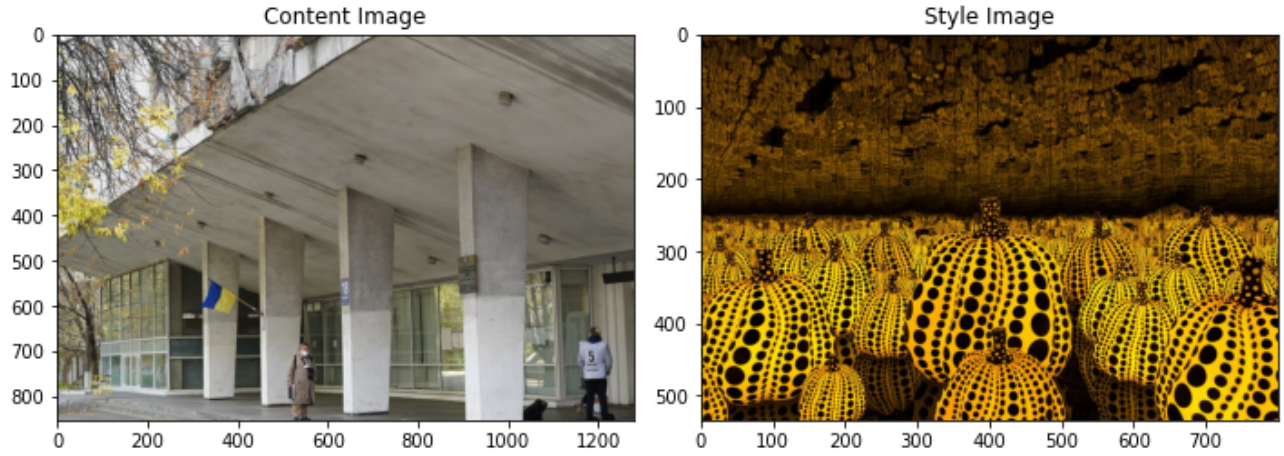
Her work is based in conceptual art and shows some attributes of feminism, minimalism, surrealism, Art Brut, pop art, and abstract expressionism, and is infused with autobiographical, psychological, and sexual content. She has been acknowledged as one of the most important living artists to come out of Japan.

On 25 February 2017, Kusama's All the Eternal Love I Have for the Pumpkins exhibit, one of the six components to her Infinity Mirror rooms at the Hirshhorn Museum, was temporarily closed for three days following damage to one of the exhibit's glowing pumpkin sculptures. The room, which measures 13 square feet (1.2 m2) and was filled with over 60 pumpkin sculptures, was one of the museum's most popular attractions ever. Allison Peck, a spokeswoman for the Hirshhorn, said in an interview that the museum "has never had a show with that kind of visitor demand", with the room averaging over 8,000 visitors between its opening and the date of its temporary closing.

Museum visitors shared 34,000 images of the exhibition to their Instagram accounts, and social media posts using the hashtag #InfiniteKusama garnered 330 million impressions, as reported by the Smithsonian the day after the exhibit's closing.[50] The works provided the perfect setting for Instagram-able selfies which inadvertently added to the performative nature of the works.

```
# Dimensions for the generated picture.

target_path = './figures/KPI_look_18.jpg'
style_path = './figures/kusama.jpg'

width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)
```

```
##Displaying Content and Style images

content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```



Data preparation

```
# Preprocess the data as per VGG16 requirements
def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = tf.keras.applications.vgg16.preprocess_input(img)
    return img
```

Inverse processing

VGG networks are trained on image with each channel normalized by mean = [103.939, 116.779, 123.68] and having channels BGR.

Furthermore, since our optimized image may take values anywhere between $-\infty$ and ∞ , we must clip those to maintain them in the 0-255 range.

```
def deprocess_image(x):
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Create inputs for the VGG16 model

We need to organize input for all three images:

- content image,
- style image,
- composite (combination) image.

```
target = K.constant(preprocess_image(target_path))
style = K.constant(preprocess_image(style_path))

# This placeholder will contain our generated image
combination_image = K.placeholder((1, img_height, img_width, 3))

# We combine the 3 images into a single batch
input_tensor = K.concatenate([target,
                              style,
                              combination_image], axis=0)
```

Build the Model

The model is loaded with pre-trained ImageNet weights. The input is a batch of 3 images. The model extracts the feature maps for content and style representations. The generated image is improved upon on every iteration.

Note that by setting include_top=False in the code below, we don't include any of the fully connected layers.

```
# Build the VGG16 network with our batch of 3 images as input.
model = vgg16.VGG16(input_tensor=input_tensor,
                    weights='imagenet',
                    include_top=False)
```

```
model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(3, 400, 600, 3)]        0
_________________________________________________________________
block1_conv1 (Conv2D)        (3, 400, 600, 64)         1792
_________________________________________________________________
block1_conv2 (Conv2D)        (3, 400, 600, 64)         36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (3, 200, 300, 64)         0
_________________________________________________________________
block2_conv1 (Conv2D)        (3, 200, 300, 128)        73856
_________________________________________________________________
block2_conv2 (Conv2D)        (3, 200, 300, 128)        147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (3, 100, 150, 128)        0
_________________________________________________________________
block3_conv1 (Conv2D)        (3, 100, 150, 256)        295168
_________________________________________________________________
block3_conv2 (Conv2D)        (3, 100, 150, 256)        590080
_________________________________________________________________
block3_conv3 (Conv2D)        (3, 100, 150, 256)        590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (3, 50, 75, 256)          0
_________________________________________________________________
block4_conv1 (Conv2D)        (3, 50, 75, 512)          1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (3, 50, 75, 512)          2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (3, 50, 75, 512)          2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (3, 25, 37, 512)          0
_________________________________________________________________
block5_conv1 (Conv2D)        (3, 25, 37, 512)          2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (3, 25, 37, 512)          2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (3, 25, 37, 512)          2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (3, 12, 18, 512)          0
```

```
==================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
__________________________________________________________________
```

Loss Function

Next, we will look at the loss function used for style transfer.

The loss function includes:

- content loss,
- style loss,
- total variation (TV) loss.

Content Loss

Compute content loss for the generated image

```
# compute content loss for the generated image
def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

Style Loss

Use gram matrix to get the correlation between channels, which ultimately act a measure of the style itself.

```
def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

Total Variation (TV) loss

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

```
    return K.sum(K.pow(a + b, 1.25))
```

Extracting Features (layers) for Content and Style

```
# Dict mapping layer names to activation tensors
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])

# Name of layer used for content loss
content_layer = 'block5_conv2'

# Name of layers used for style loss;
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']
```

Computing Loss

```
# Weights in the weighted average of the loss components
total_variation_weight = 1e-4
style_weight = 10.
content_weight = 0.025

# Define the loss by adding all components to a `loss` variable
loss = K.variable(0.)
layer_features = outputs_dict[content_layer]
target_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss = loss + content_weight * content_loss(target_features,
                                            combination_features)
for layer_name in style_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    loss += (style_weight / len(style_layers)) * sl
loss += total_variation_weight * total_variation_loss(combination_image)
```

Function to compute loss and gradients in one pass

```
grads = K.gradients(loss, combination_image)[0]

# Function to fetch the values of the current loss and the current gradients
fetch_loss_and_grads = K.function([combination_image], [loss, grads])

class Evaluator(object):
```

```
    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height, img_width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator()
```

Training - Generating Composite (Stylized) image

Below we use **Limited Memory Broyden–Fletcher–Goldfarb–Shanno (L-BGFS)** optimizer described in the paper:

Byrd, R. H.; Lu, P.; Nocedal, J.; Zhu, C. (1995). A Limited Memory Algorithm for Bound Constrained Optimization. SIAM J. Sci. Comput. 16 (5): 1190–1208.

It is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) using a limited amount of computer memory. It is a popular algorithm for parameter estimation in machine learning.

```
%%time
# Wall time (GPU): 4-5 min

from tqdm import tqdm

iterations = 50

x = preprocess_image(target_path)
x = x.flatten()
for i in tqdm(range(1, iterations)):
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss,
                                     x,
                                     fprime=evaluator.grads,
                                     maxfun=10)
    print('Iteration %0d, loss: %0.02f' %(i, min_val))
```

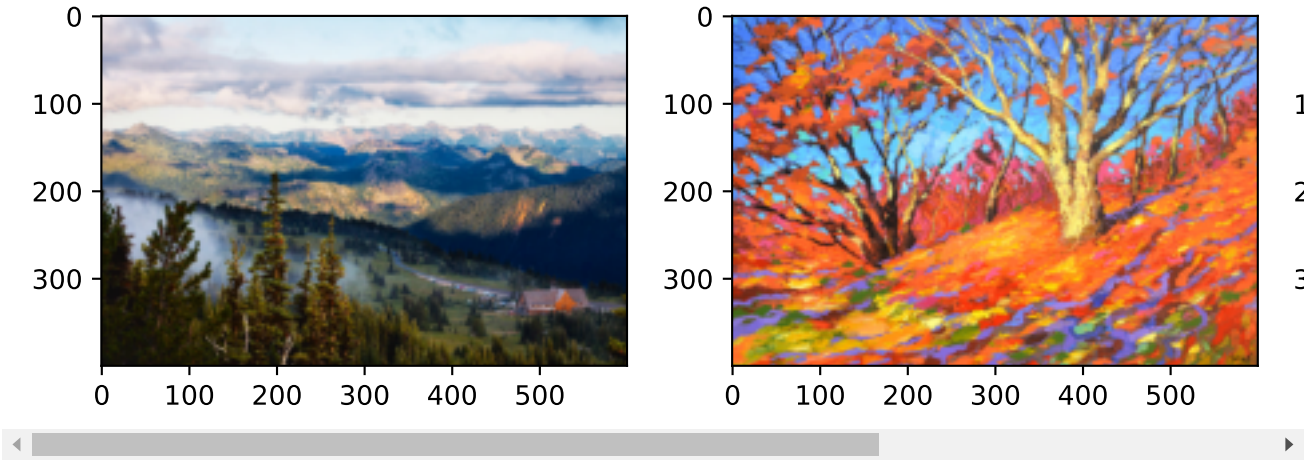
```
img = x.copy().reshape((img_height, img_width, 3))
img = deprocess_image(img)
```

```
  2%|▏          | 1/49 [00:08<06:33,  8.20s/it]Iteration 1, loss: 134052000563
  4%|▎          | 2/49 [00:11<05:22,  6.86s/it]Iteration 2, loss: 339576619008
  6%|▍          | 3/49 [00:15<04:32,  5.92s/it]Iteration 3, loss: 126946148352
  8%|▌          | 4/49 [00:19<03:57,  5.29s/it]Iteration 4, loss: 52008210432.
 10%|▋          | 5/49 [00:23<03:33,  4.85s/it]Iteration 5, loss: 27920457728.
 12%|▊          | 6/49 [00:27<03:15,  4.55s/it]Iteration 6, loss: 18009055232.
 14%|▉          | 7/49 [00:31<03:02,  4.34s/it]Iteration 7, loss: 13247670272.
 16%|█          | 8/49 [00:34<02:52,  4.21s/it]Iteration 8, loss: 10825725952.
 18%|█▏         | 9/49 [00:38<02:44,  4.12s/it]Iteration 9, loss: 9305966592.0
 20%|█▎         | 10/49 [00:42<02:38,  4.06s/it]Iteration 10, loss: 8111252480
 22%|█▍         | 11/49 [00:46<02:32,  4.00s/it]Iteration 11, loss: 6677910016
 24%|█▌         | 12/49 [00:50<02:25,  3.94s/it]Iteration 12, loss: 5937475072
 27%|█▋         | 13/49 [00:54<02:21,  3.94s/it]Iteration 13, loss: 5162137088
 29%|█▊         | 14/49 [00:58<02:17,  3.94s/it]Iteration 14, loss: 4687155712
 31%|█▉         | 15/49 [01:02<02:14,  3.94s/it]Iteration 15, loss: 4173940992
 33%|██         | 16/49 [01:06<02:09,  3.92s/it]Iteration 16, loss: 3865184256
 35%|██▏        | 17/49 [01:10<02:05,  3.93s/it]Iteration 17, loss: 3575309824
 37%|██▎        | 18/49 [01:14<02:02,  3.94s/it]Iteration 18, loss: 3332487936
 39%|██▍        | 19/49 [01:17<01:58,  3.95s/it]Iteration 19, loss: 3059362048
 41%|██▌        | 20/49 [01:21<01:54,  3.95s/it]Iteration 20, loss: 2869979136
 43%|██▋        | 21/49 [01:25<01:51,  3.96s/it]Iteration 21, loss: 2689353728
 45%|██▊        | 22/49 [01:29<01:47,  3.97s/it]Iteration 22, loss: 2536455936
 47%|██▉        | 23/49 [01:33<01:43,  3.97s/it]Iteration 23, loss: 2434625024
 49%|███        | 24/49 [01:37<01:39,  3.97s/it]Iteration 24, loss: 2337246464
 51%|███▏       | 25/49 [01:41<01:35,  3.99s/it]Iteration 25, loss: 2231579136
 53%|███▎       | 26/49 [01:45<01:32,  4.00s/it]Iteration 26, loss: 2101045120
 55%|███▍       | 27/49 [01:49<01:27,  3.98s/it]Iteration 27, loss: 2011997056
 57%|███▌       | 28/49 [01:53<01:24,  4.02s/it]Iteration 28, loss: 1903082880
 59%|███▋       | 29/49 [01:57<01:20,  4.01s/it]Iteration 29, loss: 1826006272
 61%|███▊       | 30/49 [02:01<01:16,  4.01s/it]Iteration 30, loss: 1772489984
 63%|███▉       | 31/49 [02:05<01:12,  4.00s/it]Iteration 31, loss: 1715293184
 65%|████       | 32/49 [02:10<01:08,  4.03s/it]Iteration 32, loss: 1644223360
 67%|████▏      | 33/49 [02:13<01:04,  4.01s/it]Iteration 33, loss: 1580539776
 69%|████▎      | 34/49 [02:18<01:00,  4.03s/it]Iteration 34, loss: 1487360384
 71%|████▍      | 35/49 [02:22<00:56,  4.01s/it]Iteration 35, loss: 1429213568
 73%|████▌      | 36/49 [02:26<00:52,  4.04s/it]Iteration 36, loss: 1387316736
 76%|████▋      | 37/49 [02:30<00:48,  4.04s/it]Iteration 37, loss: 1337258240
 78%|████▊      | 38/49 [02:34<00:44,  4.06s/it]Iteration 38, loss: 1296556032
 80%|████▉      | 39/49 [02:38<00:40,  4.05s/it]Iteration 39, loss: 1253664000
 82%|█████      | 40/49 [02:42<00:36,  4.05s/it]Iteration 40, loss: 1214142592
 84%|█████▏     | 41/49 [02:46<00:32,  4.05s/it]Iteration 41, loss: 1183071872
 86%|█████▎     | 42/49 [02:50<00:28,  4.05s/it]Iteration 42, loss: 1136907648
 88%|█████▍     | 43/49 [02:54<00:24,  4.06s/it]Iteration 43, loss: 1105679360
 90%|█████▌     | 44/49 [02:58<00:20,  4.06s/it]Iteration 44, loss: 1075033728
 92%|█████▋     | 45/49 [03:02<00:16,  4.06s/it]Iteration 45, loss: 1039957568
 94%|█████▊     | 46/49 [03:06<00:12,  4.08s/it]Iteration 46, loss: 999286208.
 96%|█████▉     | 47/49 [03:10<00:08,  4.09s/it]Iteration 47, loss: 974230592.
 98%|██████     | 48/49 [03:14<00:04,  4.09s/it]Iteration 48, loss: 948763968.
100%|██████████| 49/49 [03:19<00:00,  4.06s/it]Iteration 49, loss: 929080384.
CPU times: user 2min 23s, sys: 2min 10s, total: 4min 33s
Wall time: 3min 19s
```

Display All Images: Content, Style, and Composite

From Textbook

```
plt.figure(figsize=(12, 12))

plt.subplot(3,3,1)
plt.imshow(load_img(target_path, target_size=(img_height,
                                              img_width)))

plt.subplot(3,3,2)
plt.imshow(load_img(style_path, target_size=(img_height,
                                             img_width)))

plt.subplot(3,3,3)
plt.imshow(img)

plt.show()
```

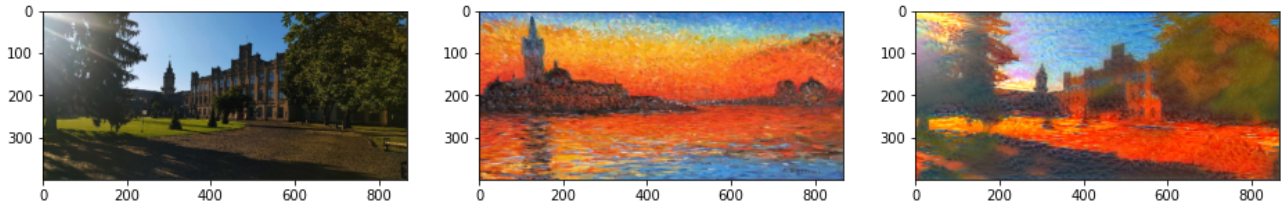



```
plt.imsave('./neural-style.jpg', postprocess(output))
```

KPI look + Monet-style

```
plt.figure(figsize=(16, 16))

plt.subplot(3,3,1)
plt.imshow(load_img(target_path, target_size=(img_height,
                                              img_width)))

plt.subplot(3,3,2)
plt.imshow(load_img(style_path, target_size=(img_height,
                                             img_width)))

plt.subplot(3,3,3)
```

```
plt.imshow(img)

plt.show()
```




```
plt.imsave('./KPI_look_1_sunny_monet-style.jpg', img)
```

```
result = Image.open('./KPI_look_1_sunny_monet-style.jpg')
plt.figure(figsize=(32, 32))
plt.imshow(result)
plt.show()
```

▸ KPI look 18 + Kusama-style

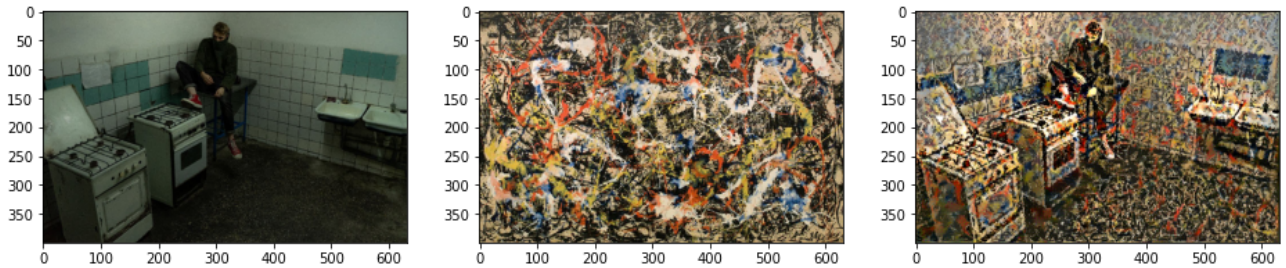
[ ] ↳ *4 cells hidden*

▸ KPI look 18 wild + Warhol-style

[ ] ↳ *4 cells hidden*

▾ KPI look UNKNOWN hero + Pollock-style

```
plt.figure(figsize=(16, 16))

plt.subplot(3,3,1)
plt.imshow(load_img(target_path, target_size=(img_height,
                                              img_width)))

plt.subplot(3,3,2)
plt.imshow(load_img(style_path, target_size=(img_height,
                                             img_width)))

plt.subplot(3,3,3)
plt.imshow(img)

plt.show()
```



```
plt.imsave('./KPI_look_UNK_pollock-style.jpg', img)
```

```
result = Image.open('./KPI_look_UNK_pollock-style.jpg')
plt.figure(figsize=(32, 32))
plt.imshow(result)
plt.show()
```

NST by TensorFlow Hub

Restart Runtime!

Neural Style Transfer (NST) from scratch is a demanding task.

But, fortunately, we can use out-of-the-box solutions that can be found in **TensorFlow Hub (TFHub)** in section image-style-transfer.

Below, we'll style our own images in just a few lines of code by harnessing the utility and convenience that TFHub provides.

Model: arbitrary-image-stylization-v1-256

The original work for artistic style transfer with neural networks proposed a slow optimization algorithm that works on any arbitrary painting. Subsequent work developed a method for fast artistic style transfer that may operate in real time, but was limited to one or a limited set of styles.

Paper:

This module performs fast artistic style transfer that may work on arbitrary painting styles as described in the following work:

Golnaz Ghiasi, Honglak Lee, Manjunath Kudlur, Vincent Dumoulin, Jonathon Shlens, Exploring the structure of a real-time, arbitrary neural artistic stylization network, Proceedings of the British Machine Vision Conference (BMVC), 2017.

Abstract:

In this paper, we present a method which combines the flexibility of the neural algorithm of artistic style with the speed of fast style transfer networks to allow real-time stylization using any content/style image pair. We build upon recent work leveraging conditional instance normalization for multi-style transfer networks by learning to predict the conditional instance normalization parameters directly from a style image. The model is successfully trained on a corpus of roughly 80,000 paintings and is able to generalize to paintings previously unobserved. We demonstrate that the learned embedding space is smooth and contains a rich structure and organizes semantic information associated with paintings in an entirely unsupervised manner.

Import Libraries

```
import matplotlib.pyplot as plt
```

```
import numpy as np
import tensorflow as tf
from tensorflow_hub import load
from PIL import Image
from tensorflow.keras.preprocessing.image import load_img, img_to_array
```

KPI look + Monet-style

Content and Style Images

```
target_path = './figures/KPI_look_1_sunny.jpg'
style_path = './figures/monet.jpg'
```

```
content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```

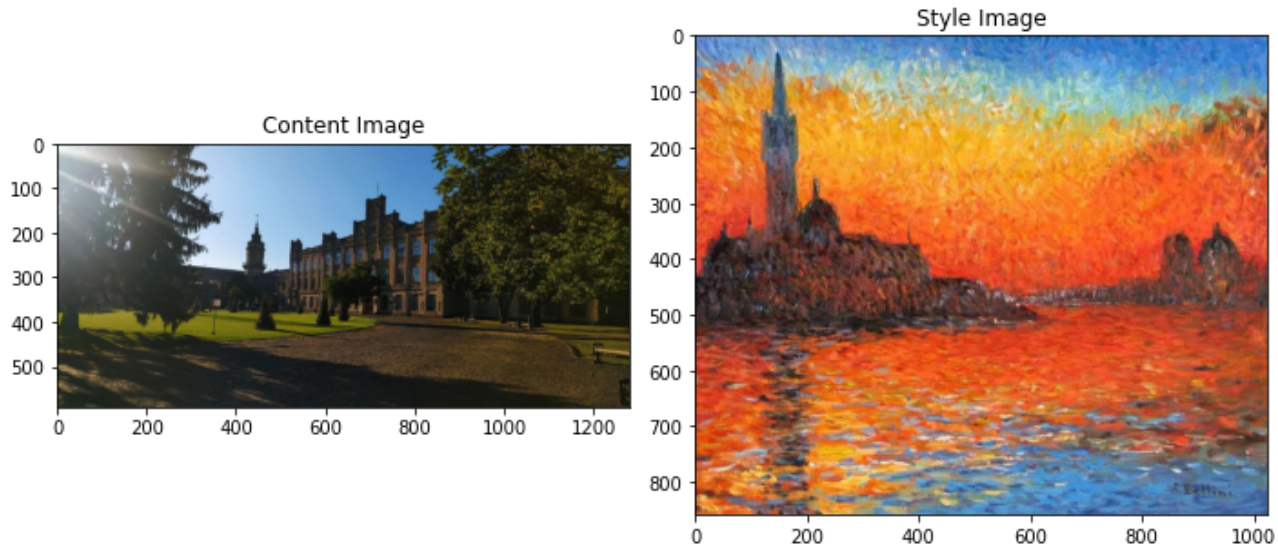


Image Pre-processing

```
# Load content and style images (see example in the attached colab).
```

```
content_image = plt.imread(target_path)
style_image = plt.imread(style_path)
# Convert to float32 numpy array, add batch dimension, and normalize to range [0,
content_image = content_image.astype(np.float32)[np.newaxis, ...] / 255.
style_image = style_image.astype(np.float32)[np.newaxis, ...] / 255.
# Optionally resize the images. It is recommended that the style image is about
# 256 pixels (this size was used when training the style transfer network).
# The content image can be any size.
style_image = tf.image.resize(style_image, (256, 256))
```

Build (actually load) Model

```
from tensorflow_hub import load

# Load image stylization module.
hub_module = load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1
```

Training - Generating Composite (Stylized) image

```
%%time

# Stylize image
outputs = hub_module(tf.constant(content_image), tf.constant(style_image))
#outputs = hub_module(content_image, style_image)
stylized_image = outputs[0]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-0b88f54ef7ca> in <module>()
----> 1 get_ipython().run_cell_magic('time', '', '\n# Stylize image\noutputs
= hub_module(tf.constant(content_image), tf.constant(style_image))\n#outputs
= hub_module(content_image, style_image)\nstylized_image = outputs[0]')

                                2 frames
<decorator-gen-53> in time(self, line, cell, local_ns)

/usr/local/lib/python3.7/dist-packages/IPython/core/magics/execution.py in
time(self, line, cell, local_ns)
   1191         else:
   1192             st = clock2()
-> 1193             exec(code, glob, local_ns)
   1194             end = clock2()
   1195             out = None

<timed exec> in <module>()

NameError: name 'hub_module' is not defined

SEARCH STACK OVERFLOW
```

Display All Images: Content, Style, and Composite

```
if len(stylized_image.shape) > 3:
  stylized_image = tf.squeeze(stylized_image, axis=0)

plt.figure(figsize=(32, 32))
plt.imshow(stylized_image)
plt.title('Composite Image')
plt.show()
```

▾ KPI look UNKNOWN hero + Pollock-style

Content and Style Images

```
target_path = './figures/KPI_look_UNK.jpg'
style_path = './figures/pollock.jpg'
```

```
content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```

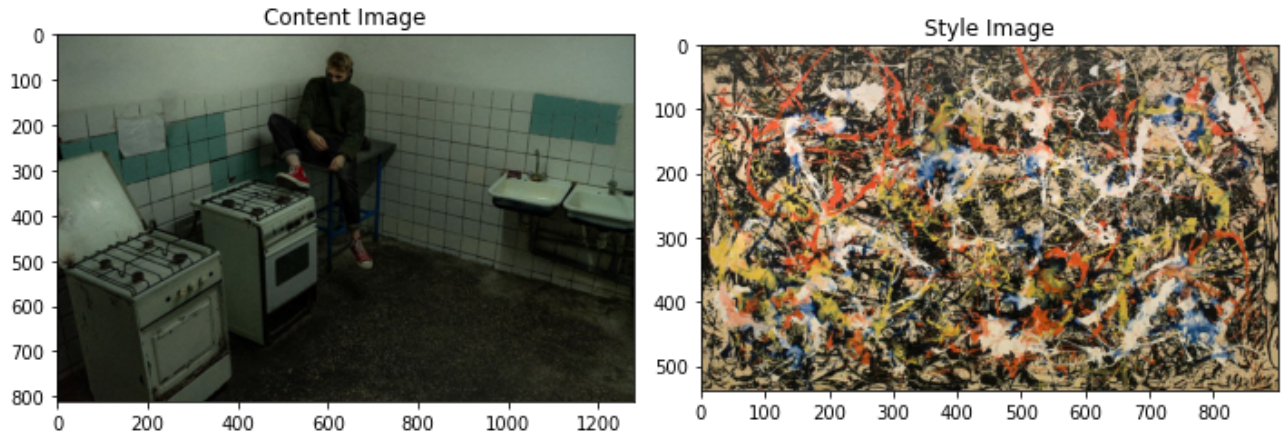


Image Pre-processing

```
# Load content and style images (see example in the attached colab).
content_image = plt.imread(target_path)
style_image = plt.imread(style_path)
# Convert to float32 numpy array, add batch dimension, and normalize to range [0,
content_image = content_image.astype(np.float32)[np.newaxis, ...] / 255.
style_image = style_image.astype(np.float32)[np.newaxis, ...] / 255.
# Optionally resize the images. It is recommended that the style image is about
# 256 pixels (this size was used when training the style transfer network).
# The content image can be any size.
style_image = tf.image.resize(style_image, (256, 256))
```

Build (actually load) Model

```
from tensorflow_hub import load

# Load image stylization module.
hub_module = load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1
```

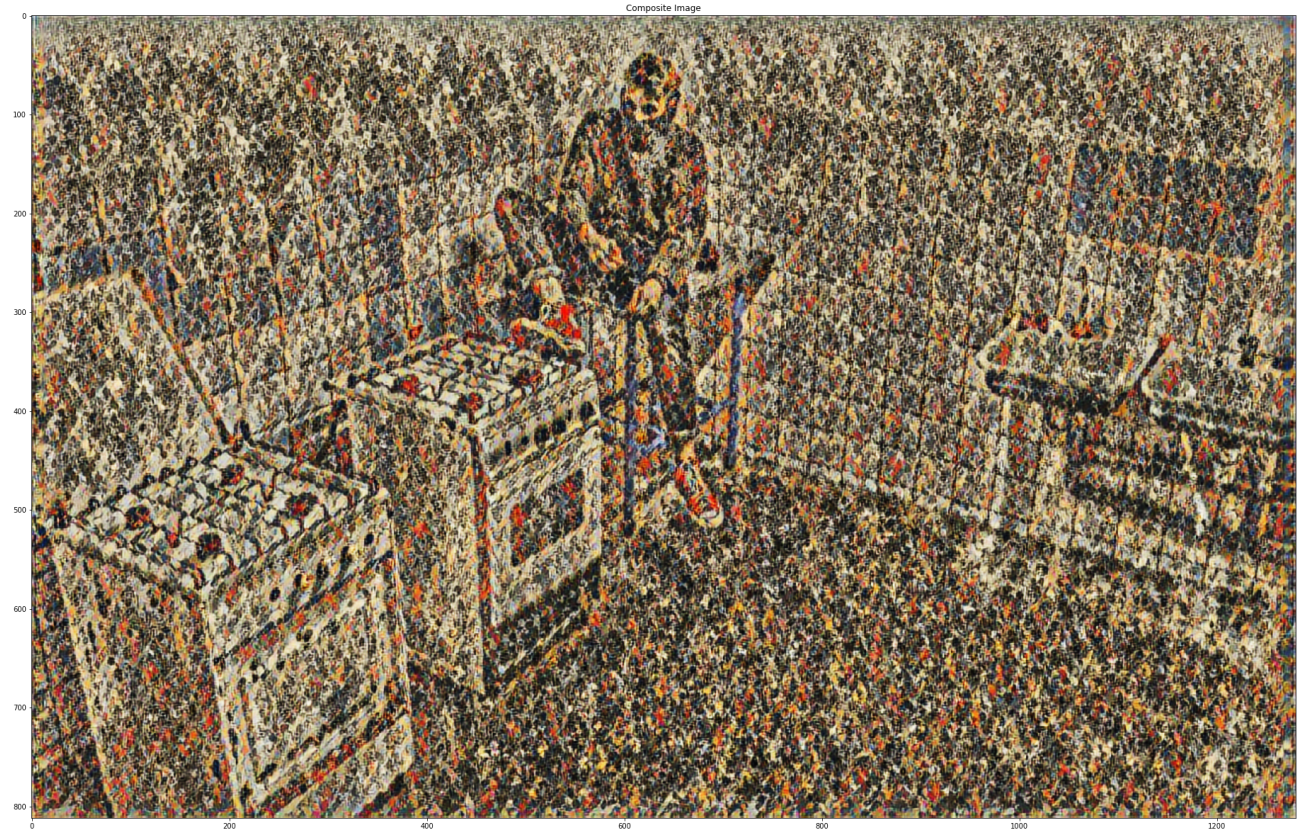
Training - Generating Composite (Stylized) image

```
%%time

# Stylize image
outputs = hub_module(tf.constant(content_image), tf.constant(style_image))
#outputs = hub_module(content_image, style_image)
stylized_image = outputs[0]
```

```
CPU times: user 1.91 s, sys: 756 ms, total: 2.67 s
Wall time: 2.64 s
```

Display All Images: Content, Style, and Composite

```
if len(stylized_image.shape) > 3:
  stylized_image = tf.squeeze(stylized_image, axis=0)

plt.figure(figsize=(32, 32))
plt.imshow(stylized_image)
plt.title('Composite Image')
plt.show()
```



Summary

In this part, you learned fancy, but important technique in CNNs, namely, **neural style transfer (NST)**.

The technique allows you to transform the contents of an image of your choice into a style of another image. You learned how to do this style transfer in different ways:

- to build your own network to do the style transfers at implemented by frameworks:
    - PyTorch,
    - TensorFlow
- to use a pre-trained model provided in TFHub to perform a fast artistic style transfer, and the style transfers using this method are quick and do a very good job.

You learned to create a network which learned how to apply the style to the given contents over several iterations. This method allows you to perform your own experiments on style transfer.

- The **loss** functions used in style transfer generally have three parts:

1. **Content** loss is used to make the composite image approximate the content image as regards content features.
2. **Style** loss is used to make the composite image approximate the style image in terms of style features.
3. **Total variation (TV)** loss helps reduce the noise in the composite image.

- We use a pre-trained CNN to extract image features and minimize the loss function to continuously update the composite image.
- We use a Gram matrix to represent the style output by the style layers.

Part 2. Image Super Resolution (ISR)

Convolutional Neural Networks (CNNs) can also be used to improve the resolution of low-quality images.

Before CNNs, we could achieve this by using:

- interpolation techniques,
- example-based approaches, or
- low- to high-resolution mappings that must be learned.

As we can see below, we can obtain better results faster by using an end-to-end DL-based approach.

The demo below is aimed to improve the resolution of low-quality images containing **dogs and cats ONLY**!

In simple words, we would like to us ISR and get the better quality of dogs/cats from any low resolutions images with dogs/cats.

Import Libraries

```
import pathlib
from glob import glob

import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from tensorflow.keras import Model
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import *
```

Functions for ISR-model and Image Pre- and Post-processing

```
def build_srcnn(height, width, depth):
    input = Input(shape=(height, width, depth))

    x = Conv2D(filters=64, kernel_size=(9, 9),
               kernel_initializer='he_normal')(input)
    x = ReLU()(x)
    x = Conv2D(filters=32, kernel_size=(1, 1),
               kernel_initializer='he_normal')(x)
    x = ReLU()(x)
    output = Conv2D(filters=depth, kernel_size=(5, 5),
                    kernel_initializer='he_normal')(x)

    return Model(input, output)


def resize_image(image_array, factor):
    original_image = Image.fromarray(image_array)
    new_size = np.array(original_image.size) * factor
    new_size = new_size.astype(np.int32)
    new_size = tuple(new_size)

    resized = original_image.resize(new_size)
    resized = img_to_array(resized)
    resized = resized.astype(np.uint8)

    return resized


def tight_crop_image(image):
    height, width = image.shape[:2]
    width -= int(width % SCALE)
    height -= int(height % SCALE)

    return image[:height, :width]


def downsize_upsize_image(image):
    scaled = resize_image(image, 1.0 / SCALE)
    scaled = resize_image(scaled, SCALE / 1.0)

    return scaled


def crop_input(image, x, y):
    y_slice = slice(y, y + INPUT_DIM)
    x_slice = slice(x, x + INPUT_DIM)

    return image[y_slice, x_slice]


def crop_output(image, x, y):
```

```
    y_slice = slice(y + PAD, y + PAD + LABEL_SIZE)
    x_slice = slice(x + PAD, x + PAD + LABEL_SIZE)

    return image[y_slice, x_slice]
```

Get Open Source Dataset with Cats and Dogs

```
!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \
    -O /tmp/cats_and_dogs_filtered.zip
```

```
--2021-03-31 08:39:31--  https://storage.googleapis.com/mledu-datasets/cats_a
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.65.80, 1
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.65.80|:
HTTP request sent, awaiting response... 200 OK
Length: 68606236 (65M) [application/zip]
Saving to: '/tmp/cats_and_dogs_filtered.zip'

/tmp/cats_and_dogs_ 100%[===================>]  65.43M   191MB/s    in 0.3s

2021-03-31 08:39:32 (191 MB/s) - '/tmp/cats_and_dogs_filtered.zip' saved [686
```

Extract and Check Data

```
import os
import zipfile

local_zip = '/tmp/cats_and_dogs_filtered.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('./tmp')
zip_ref.close()
```

```
! ls ./tmp/cats_and_dogs_filtered/train
```

```
cats  dogs
```

```
! mkdir ./tmp/cats_and_dogs_filtered/dogscats
! cp ./tmp/cats_and_dogs_filtered/train/dogs/* ./tmp/cats_and_dogs_filtered/dogsca
! cp ./tmp/cats_and_dogs_filtered/train/cats/* ./tmp/cats_and_dogs_filtered/dogsca
#! ls ./tmp/cats_and_dogs_filtered/dogscats
```

```
mkdir: cannot create directory './tmp/cats_and_dogs_filtered/dogscats': File
```

```
from tqdm import tqdm
```

```
SEED = 42
np.random.seed(SEED)
SUBSET_SIZE = 500

dataset_paths = [*glob('./tmp/cats_and_dogs_filtered/dogscats/*.jpg')]
print('Full dataset: len(dataset_paths)=',len(dataset_paths))
#print(dataset_paths)
#dataset_paths = np.random.choice(dataset_paths, SUBSET_SIZE)
#print('Subset of dataset: len(dataset_paths)=',len(dataset_paths))
```

```
Full dataset: len(dataset_paths)= 2000
```

Re-format Dataset

The inputs will be low-resolution patches that have been extracted from the images after being downsized and upsized.

```
%%time

SCALE = 4.0
INPUT_DIM = 33
LABEL_SIZE = 21
PAD = int((INPUT_DIM - LABEL_SIZE) / 2.0)
STRIDE = 14

data = []
labels = []
for image_path in tqdm(dataset_paths):
    image = load_img(image_path)
    image = img_to_array(image)
    image = image.astype(np.uint8)

    image = tight_crop_image(image)
    scaled = downsize_upsize_image(image)

    height, width = image.shape[:2]

    for y in range(0, height - INPUT_DIM + 1, STRIDE):
        for x in range(0, width - INPUT_DIM + 1, STRIDE):
            crop = crop_input(scaled, x, y)
            target = crop_output(image, x, y)

            data.append(crop)
            labels.append(target)

data = np.array(data)
labels = np.array(labels)
```

```
100%|██████████| 2000/2000 [00:21<00:00, 93.16it/s]
CPU times: user 21.5 s, sys: 2.43 s, total: 24 s
Wall time: 24.9 s
```

```
labels.shape
```

```
(1341578, 21, 21, 3)
```

Train

```
%%time

EPOCHS = 10

optimizer = Adam(lr=1e-3, decay=1e-3 / EPOCHS)
model = build_srcnn(INPUT_DIM, INPUT_DIM, 3)
model.compile(loss='mse', optimizer=optimizer)

BATCH_SIZE = 64
model.fit(data, labels, verbose=1, batch_size=BATCH_SIZE, epochs=EPOCHS)
```

```
Epoch 1/10
20963/20963 [==============================] - 160s 6ms/step - loss: 169.7474
Epoch 2/10
20963/20963 [==============================] - 129s 6ms/step - loss: 108.8311
Epoch 3/10
20963/20963 [==============================] - 129s 6ms/step - loss: 107.1874
Epoch 4/10
20963/20963 [==============================] - 129s 6ms/step - loss: 106.2954
Epoch 5/10
20963/20963 [==============================] - 129s 6ms/step - loss: 106.2813
Epoch 6/10
20963/20963 [==============================] - 129s 6ms/step - loss: 105.7666
Epoch 7/10
20963/20963 [==============================] - 129s 6ms/step - loss: 105.5526
Epoch 8/10
20963/20963 [==============================] - 130s 6ms/step - loss: 105.1966
Epoch 9/10
20963/20963 [==============================] - 130s 6ms/step - loss: 105.0190
Epoch 10/10
20963/20963 [==============================] - 130s 6ms/step - loss: 105.3183
<tensorflow.python.keras.callbacks.History at 0x7efe64758350>
```

Test

Get Any High-Quality Image with Dogs/Cats

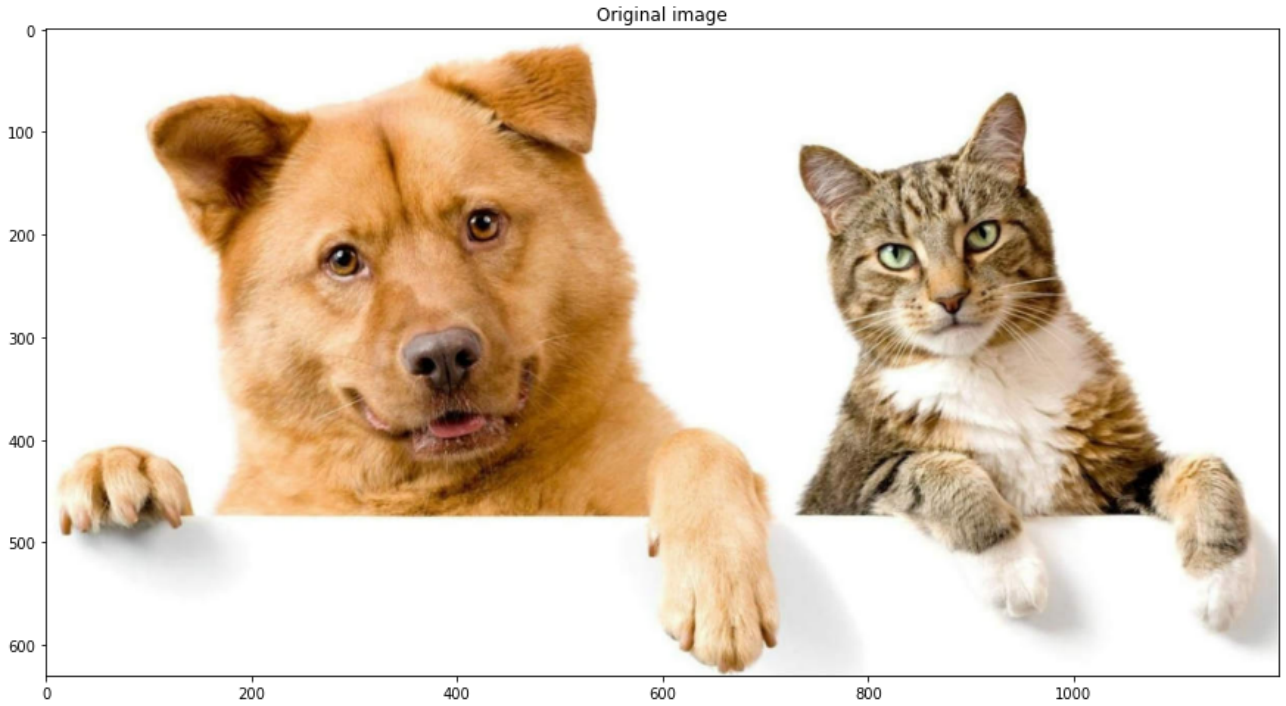
```
! wget https://dog-cat.com.ua/wp-content/uploads/2021/01/dog-and-cat-cover.jpg
! ls *.jpg
```

```
--2021-03-31 08:17:36--  https://dog-cat.com.ua/wp-content/uploads/2021/01/dc
Resolving dog-cat.com.ua (dog-cat.com.ua)... 185.104.45.88, 2a06:6440:0:2d58:
```

```
Connecting to dog-cat.com.ua (dog-cat.com.ua)|185.104.45.88|:443... connected
HTTP request sent, awaiting response... 200 OK
Length: 72452 (71K) [image/jpeg]
Saving to: ‘dog-and-cat-cover.jpg’

dog-and-cat-cover.j 100%[===================>]  70.75K   315KB/s    in 0.2s

2021-03-31 08:17:37 (315 KB/s) - ‘dog-and-cat-cover.jpg’ saved [72452/72452]

dog-and-cat-cover.jpg
```

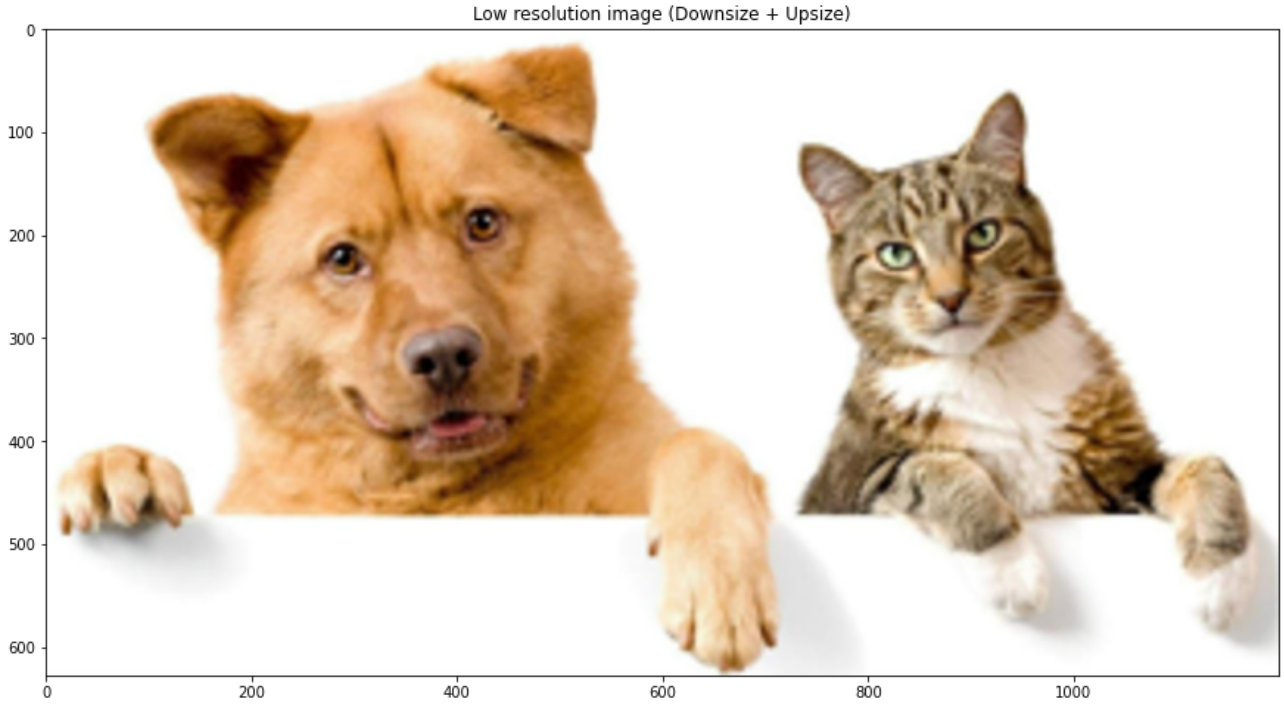
Original

```
# Test on image
image = load_img('./dog-and-cat-cover.jpg')
image = img_to_array(image)
image = image.astype(np.uint8)

plt.figure(figsize=(15, 15))
plt.title('Original image')
plt.imshow(image)
plt.show()
```



Low Resolution Version

```
scaled = downsize_upsize_image(image)

plt.figure(figsize=(15, 15))
plt.title('Low resolution image (Downsize + Upsize)')
plt.imshow(scaled)
plt.show()
```



"Predict" the Better Quality Image

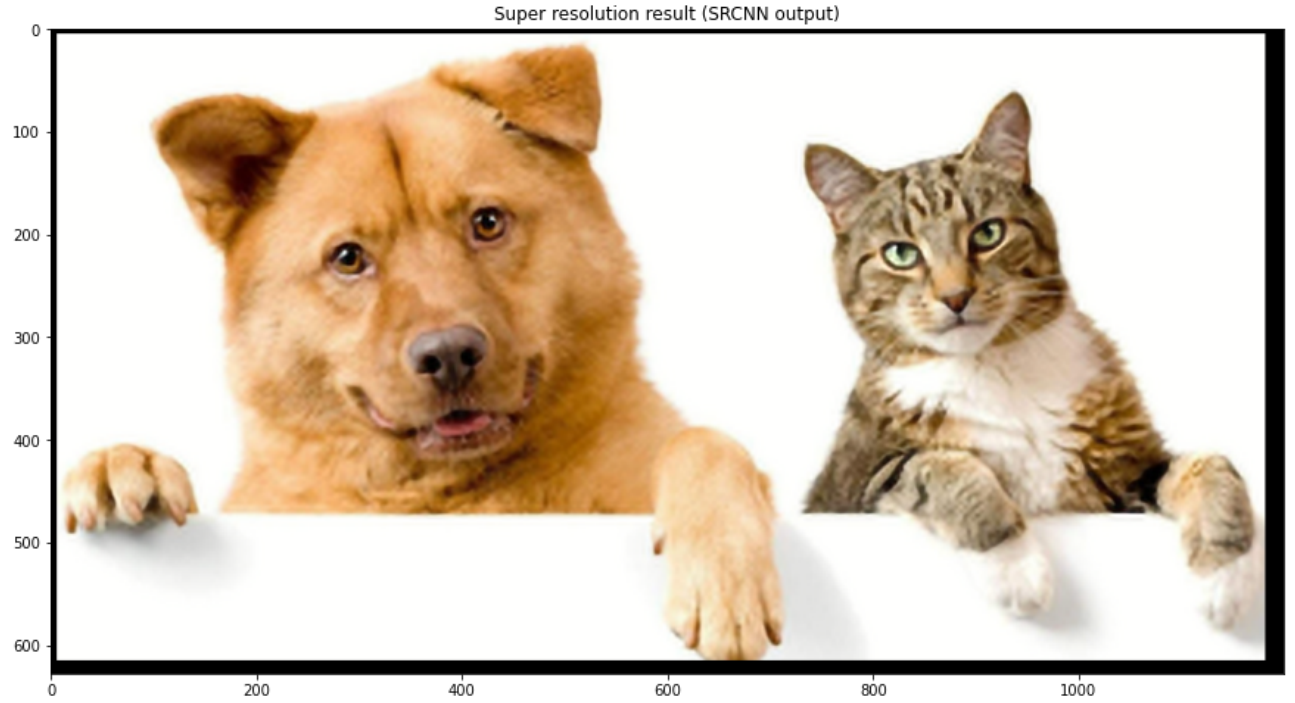
```
output = np.zeros(scaled.shape)
height, width = output.shape[:2]

for y in tqdm(range(0, height - INPUT_DIM + 1, LABEL_SIZE)):
    for x in range(0, width - INPUT_DIM + 1, LABEL_SIZE):
        crop = crop_input(scaled, x, y)

        image_batch = np.expand_dims(crop, axis=0)
        prediction = model.predict(image_batch)
        new_shape = (LABEL_SIZE, LABEL_SIZE, 3)
        prediction = prediction.reshape(new_shape)

        output_y_slice = slice(y + PAD, y + PAD + LABEL_SIZE)
        output_x_slice = slice(x + PAD, x + PAD + LABEL_SIZE)
        output[output_y_slice, output_x_slice] = prediction
```

```
100%|██████████| 29/29 [00:50<00:00,  1.72s/it]
```

"Predicted" ISR-result: Better Quality Image

```
plt.figure(figsize=(15, 15))
plt.title('Super resolution result (SRCNN output)')
plt.imshow(output / 255)
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
```



Part 3. Creative Abstract Art

DeepDream is an algorithm used to make neural networks produce dream-like images.

DeepDream is the result of an experiment that aimed to visualize the internal patterns that are learned by a neural network.

In order to achieve this goal, we can

- pass an image through the network,
- compute its gradient with respect to the activations of a specific layer, and
- then modify the image to increase the magnitude of such activations to, in turn, magnify the patterns.

The result? Psychedelic, surreal photos in late-1960->earlier-1970 fashion style! :)

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.applications.inception_v3 import *


class DeepDreamer(object):
    def __init__(self,
                 octave_scale=1.30,
                 octave_power_factors=None,
                 layers=None):

        self.octave_scale = octave_scale

        if octave_power_factors is None:
            self.octave_power_factors = [*range(-2, 3)]
        else:
            self.octave_power_factors = octave_power_factors

        if layers is None:
            self.layers = ['mixed3', 'mixed5']
        else:
            self.layers = layers

        self.base_model = InceptionV3(weights='imagenet',
                                      include_top=False)
        outputs = [self.base_model.get_layer(name).output
                   for name in self.layers]
        self.dreamer_model = Model(self.base_model.input,
                                   outputs)

    def _calculate_loss(self, image):
        image_batch = tf.expand_dims(image, axis=0)
        activations = self.dreamer_model(image_batch)

        if len(activations) == 1:
            activations = [activations]

        losses = []
        for activation in activations:
            loss = tf.math.reduce_mean(activation)
            losses.append(loss)

        total_loss = tf.reduce_sum(losses)
```

```
        return total_loss

    @tf.function
    def _gradient_ascent(self, image, steps, step_size):
        loss = tf.constant(0.0)

        for _ in range(steps):
            with tf.GradientTape() as tape:
                tape.watch(image)
                loss = self._calculate_loss(image)

            gradients = tape.gradient(loss, image)
            gradients /= tf.math.reduce_std(gradients) + 1e-8

            image = image + gradients * step_size
            image = tf.clip_by_value(image, -1, 1)

        return loss, image

    def _deprocess(self, image):
        image = 255 * (image + 1.0) / 2.0
        image = tf.cast(image, tf.uint8)
        image = np.array(image)

        return image

    def _dream(self, image, steps, step_size):
        image = preprocess_input(image)
        image = tf.convert_to_tensor(image)
        step_size = tf.convert_to_tensor(step_size)
        step_size = tf.constant(step_size)
        steps_remaining = steps

        current_step = 0
        while steps_remaining > 0:
            if steps_remaining > 100:
                run_steps = tf.constant(100)
            else:
                run_steps = tf.constant(steps_remaining)

            steps_remaining -= run_steps
            current_step += run_steps

            loss, image = self._gradient_ascent(image,
                                                run_steps,
                                                step_size)

        result = self._deprocess(image)
        return result

    def dream(self, image, steps=100, step_size=0.01):
        image = tf.constant(np.array(image))
        base_shape = tf.shape(image)[:-1]
        base_shape = tf.cast(base_shape, tf.float32)
```

```
        for factor in self.octave_power_factors:
            new_shape = tf.cast(
                base_shape * (self.octave_scale ** factor),
                tf.int32)
            image = tf.image.resize(image, new_shape).numpy()
            image = self._dream(image, steps=steps,
                                step_size=step_size)

        base_shape = tf.cast(base_shape, tf.int32)
        image = tf.image.resize(image, base_shape)
        image = tf.image.convert_image_dtype(image / 255.0,
                                             dtype=tf.uint8)
        image = np.array(image)

        return np.array(image)
```

```
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import *

#from ch4.recipe1.deepdream import DeepDreamer


def load_image(image_path):
    image = load_img(image_path)
    image = img_to_array(image)

    return image


def show_image(image, title):
    plt.figure(figsize=(15, 15))
    plt.title(title)
    plt.imshow(image)
    plt.show()
```

Original

```
target_path = './figures/KPI_look_1_sunny.jpg'
original_image = load_image(target_path)
show_image(original_image / 255.0, 'Original')
```

DeepDream with the default parameters

```
%%time
# Wall time (GPU): 2 min 4 sec

dreamy_image_DEFAULT = DeepDreamer().dream(original_image)
show_image(dreamy_image_DEFAULT, 'DEFAULT - DeepDreamer().dream')
```

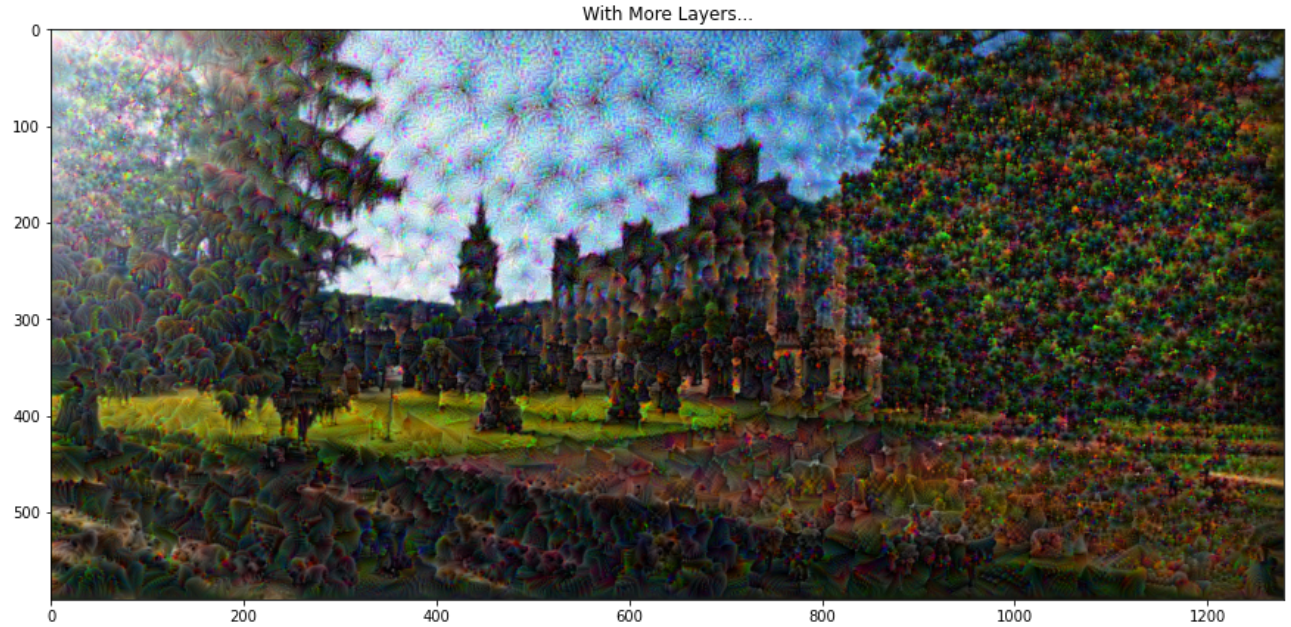
```
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
```


```
CPU times: user 30.3 s, sys: 4.55 s, total: 34.8 s
Wall time: 2min 4s
```

DeepDream with more, higher-level layers

```
%%time
# Wall time (GPU): 2 min 34 sec

dreamy_image_LAYERS = (DeepDreamer(layers=['mixed2',
                                           'mixed5',
                                           'mixed7'])
                       .dream(original_image))
show_image(dreamy_image_LAYERS, 'With More Layers...')
```

```
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
```



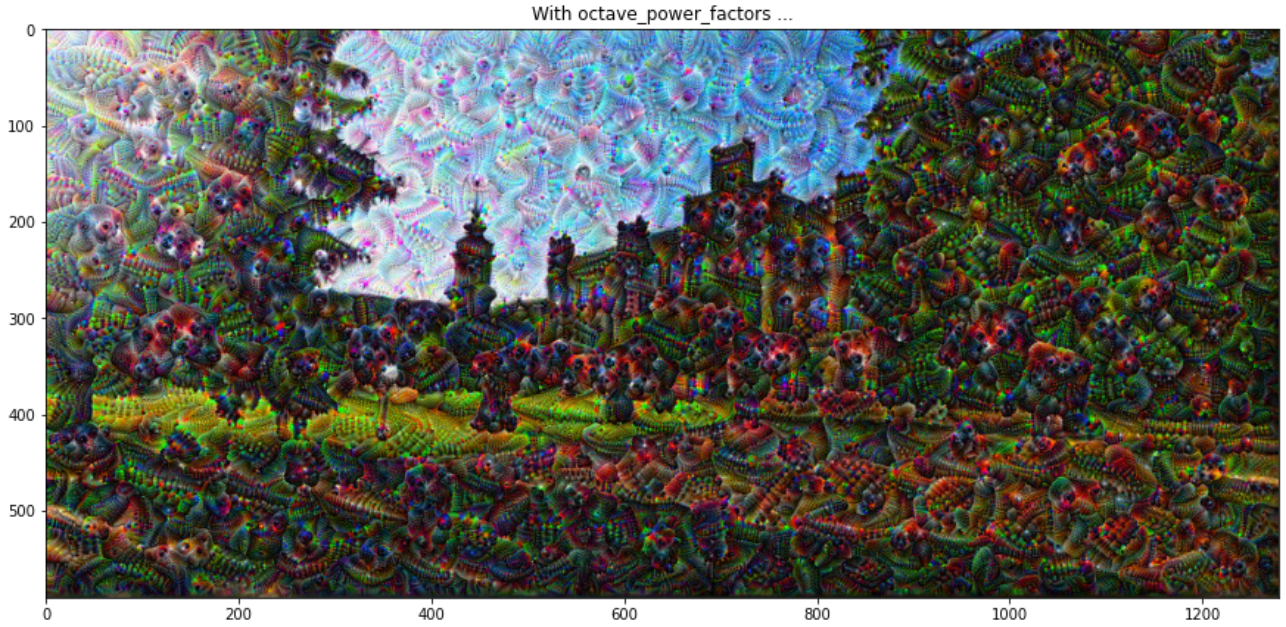

```
CPU times: user 55.1 s, sys: 26.4 s, total: 1min 21s
Wall time: 2min 34s
```

DeepDream with more octaves

```
%%time
# Wall time (GPU): 2 min 5 sec

dreamy_image_OCTAVES = (DeepDreamer(octave_power_factors=[-3, -1,
                                                          0, 3])
```

```
                  .dream(original_image))
show_image(dreamy_image_OCTAVES, 'With octave_power_factors ...')
```

```
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
WARNING:tensorflow:11 out of the last 11 calls to <function DeepDreamer._grad
```




```
CPU times: user 26.6 s, sys: 4.87 s, total: 31.4 s
Wall time: 2min 5s
```

Нейронні мережі

Лекція_10

********** **********

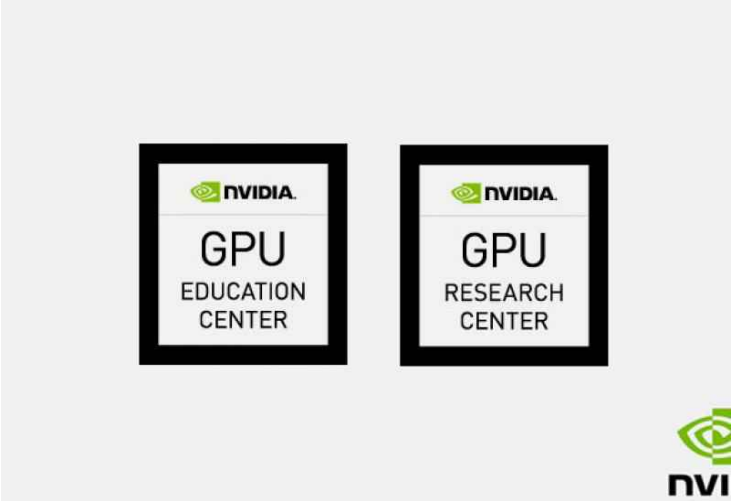
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 10 - Нейронні мережі -Сучасні CNN – Пошук найкращої моделі

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1
Пошук найкращої попередньо підготовленої моделі для класифікації зображень
https://drive.google.com/file/d/1oHJxoAZaDDIUlsAKSixGtORjDXNPz3gl/view?usp=sharing

Lecture 10 - Search for the Best Pre-trained Model for Image Classification

(C) partially based on the works by d2l Open Source book authors, TFHub contributors, Planche, Martinez, Asadi, ...

Deciding **which pre-trained model** - for example, from the 28 currently avaialble models in Keras TF2 framework - will be **the best model** for your DL problem ... is **a very big and resource-intensive task**.

This DEMO will use the so-called **data-driven method** to find out the best Keras TF2 model **pre-trained on ImageNet** for the following classic datasets:

- `cats_vs_dogs`,
- `CIFAR10`,
- `CIFAR100`,
- ...,
- `ImageNet` - **it is NOT considered here** - very BIG for validation part (6 GB) even + it is fast to download to Colab (>24 hours to download to Colab).

In the future it will also help you easily:

- input **any dataset**,
- configure **transfer learning (TL) models** provided by Keras in TF2,
- test all these TL models in **brute force** manner,
- **analyze the metrics** obtained,
- **choose the best model** for your problem's dataset.

Best Model - How to Find It among Many Models?

As it was already shown in the previous lectures, Transfer Learning (TL) is a technique in machine learning where we can take a model developed on one task and use it as a starting point in some other similar but different tasks.

TL is extremely popular in DL since the "transfer" of knowledge from one "parent" model to a "child" model means that the "child" model can be trained to high accuracies with a much smaller dataset compared to the "parent" model.

High-level DL frameworks like TensorFlow and Pytorch have made it incredibly easy **to leverage the power of TL** by including several **pre-trained models** within the package itself.

For example, the TensorFlow2 Keras API includes 28 highly **advanced model architectures** pre-trained on the "ImageNet" dataset.

During recent years **ImageNet** is estimated as the **State Of The Art (SOTA)** of image classification datasets.

Most Image Classification tasks in DL today will start by:

- **downloading models** (one or more from these 28 pre-trained models),
- **modify the models slightly** to suit the task on hand,
- **train only the custom modifications** while **freezing** the layers in the pre-trained model.

This approach gives very high accuracy on real-world image classification tasks since **ImageNet** collects **very many** real-world images.

However, choosing which of the 28 pre-trained models to use is not always an exact science.

Many developers select **the models they are familiar** ... :) and me too ... and gave them good results in the past - now, this is a naive not-effective, and stupid way. :(

As a Data Scientist, you should use the classic AutoML-like approach, namely, **the data-driven AutoDL approach** to select the pre-trained model free from personal biases that can be implemented in a **brute force** manner. **It is stupid also**, but ... effective sometimes.

Best Model - Selection Criteria

In general, there are several competing criteria while doing any DL task in the industry:

1. **Accuracy**: the Higher the Better
2. **Prediction (Inference) Time** (sometimes Training): the Smaller the Better
3. **Training Time**: the Smaller the Better
4. **Size**: the Smaller the Better (especially when you need to port it to Edge Computing layer)

These criteria are very straightforward.

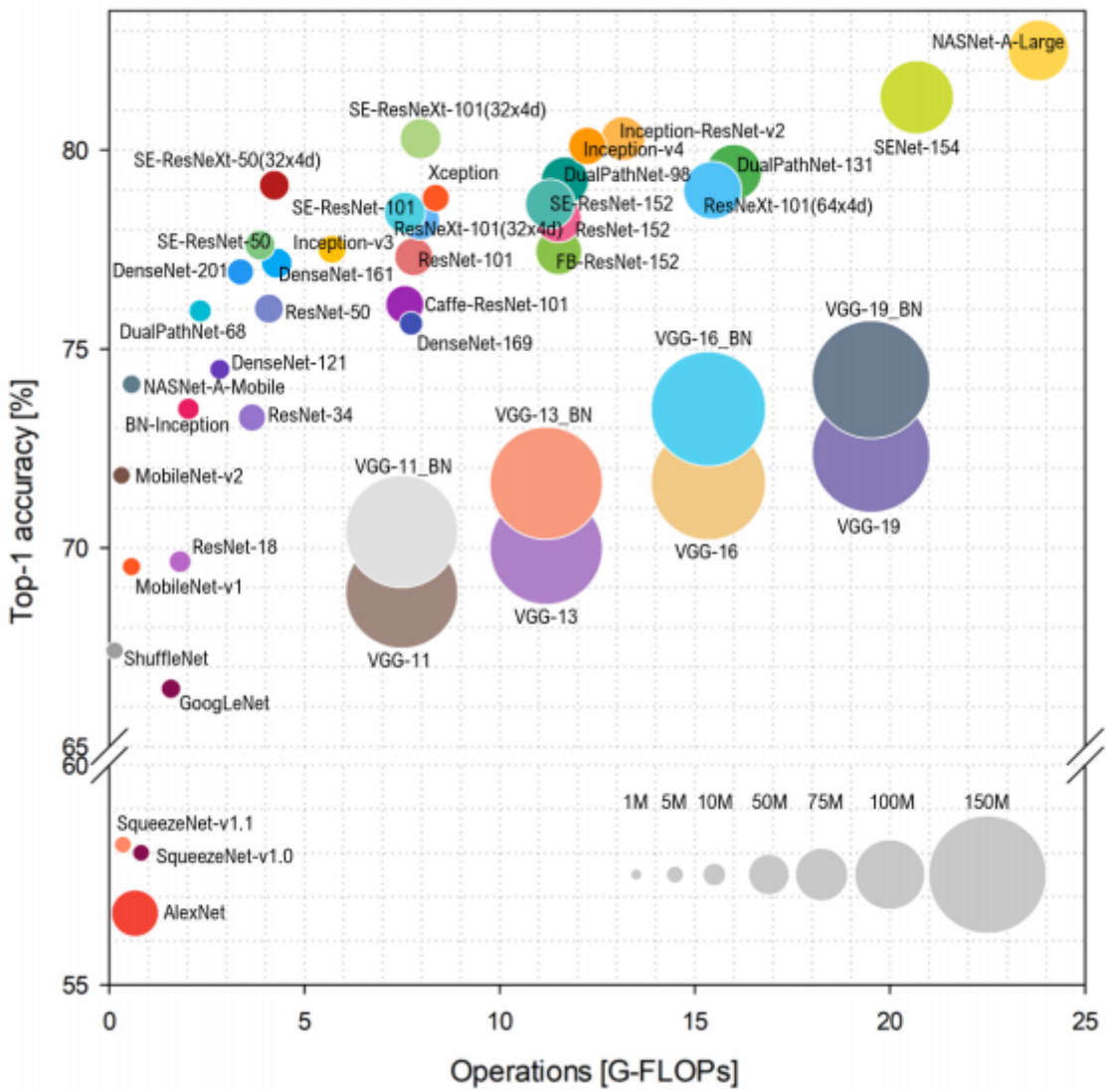
We want the model that gives us the highest accuracy on validation data since it can make useful predictions.

We also want the model to train and predict as fast as possible because, in production, we might need to serve hundreds or thousands of predictions every second.

We the model that could work on Edge Computing devices with the limited computing resources.

It is not simple!

In general, to get higher accuracy, Data Scientist needs to use a "deeper" or larger model. But a larger model has many more parameters that make it slower to execute. We can see the tradeoff between the accuracy and size/number of operations of the model from the below graph Reference.



Data Scientist should find some **balance**: to choose

- the **smallest model**
- that provides the **good enough accuracy** and
- **smallest prediction time**.

That is why, we need to experiment among many available models to pick the one that meets these criteria.

NOTE:

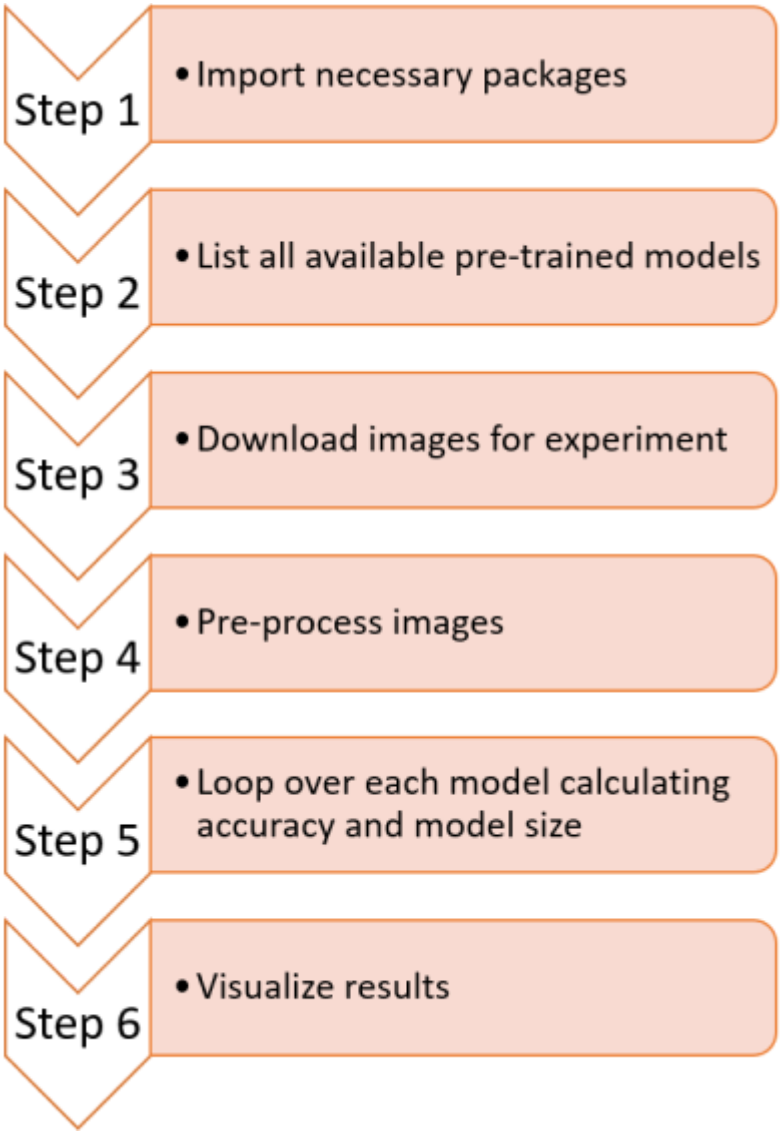
If Data Scientist only care about achieving the highest accuracy no matter the speed, then he/she could combine all these models using ... **ensembling** techniques (like in XGB, ... AutoML techniques)!

In fact, ML/DL ensembling is the most popular approach in the current scientific applications and ML/DL competitions like Kaggle, ILSVRC, and others.

Workflow - Brute Force Experiment

The workflow is shown in the image below:

1. Import libraries
2. Find all the pre-trained models in Keras TF2.
3. Download the standard dataset from TF2: cats_vs_dogs, CIFAR10, CIFAR100, ... or other. Later, you can replace it with any other dataset, including your own custom dataset.
4. Pre-process images to satisfy demands for input data from the pre-trained models.
5. Train all the models on the `cats_vs_dogs` dataset in a loop.
6. Present and analyze the criteria for the best model.



1.Import Libraries

```
# Imports
import tensorflow as tf
import tensorflow_datasets as tfds

import pandas as pd
import matplotlib.pyplot as plt
```

```
import inspect
from tqdm import tqdm

# Set batch size for training and validation
batch_size = 32
```

```
! nvidia-smi
```

```
Thu Apr  8 06:22:24 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.67       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   58C    P8    12W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

2.Pre-trained Models in Keras TF2

Automatically get a list of all available pre-trained models from Keras by listing all the functions inside `tf.keras.applications`.

Since each model is instantiated by calling a function from `tf.keras.applications`, when we list all the functions within this module using `inspect.isfunction`, we get a list of all the models.

In TF2.2, there are a total of 28 models that we can use.

You can add any other custom pre-trained models into the experiments by manually adding elements into the `model_dictionary` as well by following the syntax `model_dictionary["new_model_name"] = new_model_function()`, where `new_model_function()` should return the custom pre-trained model without the final output `Dense` layer.

NOTE:

The model's input shape should be (224,224,3) if you don't want to change any other code below.

```
# List all available models
model_dictionary = {m[0]:m[1] for m in inspect.getmembers(tf.keras.applications, i
```

```
model_dictionary
```

```
{'DenseNet121': <function
tensorflow.python.keras.applications.densenet.DenseNet121>,
 'DenseNet169': <function
tensorflow.python.keras.applications.densenet.DenseNet169>,
 'DenseNet201': <function
tensorflow.python.keras.applications.densenet.DenseNet201>,
 'EfficientNetB0': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB0>,
 'EfficientNetB1': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB1>,
 'EfficientNetB2': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB2>,
 'EfficientNetB3': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB3>,
 'EfficientNetB4': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB4>,
 'EfficientNetB5': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB5>,
 'EfficientNetB6': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB6>,
 'EfficientNetB7': <function
tensorflow.python.keras.applications.efficientnet.EfficientNetB7>,
 'InceptionResNetV2': <function
tensorflow.python.keras.applications.inception_resnet_v2.InceptionResNetV2>,
 'InceptionV3': <function
tensorflow.python.keras.applications.inception_v3.InceptionV3>,
 'MobileNet': <function
tensorflow.python.keras.applications.mobilenet.MobileNet>,
 'MobileNetV2': <function
tensorflow.python.keras.applications.mobilenet_v2.MobileNetV2>,
 'MobileNetV3Large': <function
tensorflow.python.keras.applications.mobilenet_v3.MobileNetV3Large>,
 'MobileNetV3Small': <function
tensorflow.python.keras.applications.mobilenet_v3.MobileNetV3Small>,
 'NASNetLarge': <function
tensorflow.python.keras.applications.nasnet.NASNetLarge>,
 'NASNetMobile': <function
tensorflow.python.keras.applications.nasnet.NASNetMobile>,
 'ResNet101': <function
tensorflow.python.keras.applications.resnet.ResNet101>,
 'ResNet101V2': <function
tensorflow.python.keras.applications.resnet_v2.ResNet101V2>,
 'ResNet152': <function
tensorflow.python.keras.applications.resnet.ResNet152>,
 'ResNet152V2': <function
tensorflow.python.keras.applications.resnet_v2.ResNet152V2>,
 'ResNet50': <function
tensorflow.python.keras.applications.resnet.ResNet50>,
 'ResNet50V2': <function
tensorflow.python.keras.applications.resnet_v2.ResNet50V2>,
 'VGG16': <function tensorflow.python.keras.applications.vgg16.VGG16>,
 'VGG19': <function tensorflow.python.keras.applications.vgg19.VGG19>,
```

```
 'Xception': <function
tensorflow.python.keras.applications.xception.Xception>}
```

Part 1.Train + Validation on Simple Dataset - `cats_vs_dogs`

3.Dataset - `cats_vs_dogs`

A large set of images of cats and dogs. There are 1738 corrupted images that are dropped.

Download some images to run the experiment.

NOTE:

You could modify this step to load your data when you run the experiment for your use case.

```
# Download the training and validation data
(train, validation), metadata = tfds.load('cats_vs_dogs', split=['train[:70%]', 't
                                          with_info=True, as_supervised=True)

# Number of training examples and labels
num_train = len(list(train))
num_validation = len(list(validation))
num_classes = len(metadata.features['label'].names)
num_iterations = int(num_train/batch_size)

# Print important info
print(f'Num train images: {num_train} \
        \nNum validation images: {num_validation} \
        \nNum classes: {num_classes} \
        \nNum iterations per epoch: {num_iterations}')
```

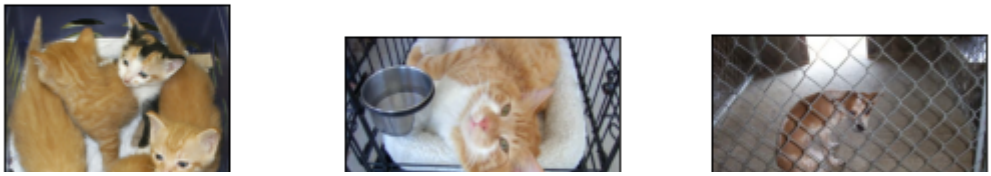
```
Num train images: 16283
Num validation images: 6979
Num classes: 2
Num iterations per epoch: 508
```

The example below creates a plot of the first nine images in the training dataset.

This function - `tfds.show_examples` - is for interactive use (Colab, Jupyter) **ONLY!**

It displays and return a plot of (rows*columns) images from a tf.data.Dataset.

```
fig = tfds.show_examples(train, metadata)

# show the figure
fig.show()
```



4.Image Pre-processing

cat (0)

- Images are **resized**, because some pre-trained models require images to be of size (224,224,3) while some require (331,331,3).
- Images are **normalized** them by dividing each pixel by 255.
- Labels are **one-hot encoded** so that we can use `categorical_crossentropy` loss during training.

```
def normalize_img(image, label, img_size):
    # Resize image to the desired img_size and normalize it
    # One hot encode the label
    image = tf.image.resize(image, img_size)
    image = tf.cast(image, tf.float32) / 255.
    label = tf.one_hot(label, depth=num_classes)
    return image, label

def preprocess_data(train, validation, batch_size, img_size):
    # Apply the normalize_img function on all train and validation data and create
    train_processed = train.map(lambda image, label: normalize_img(image, label, i
    train_processed = train_processed.batch(batch_size).repeat()

    validation_processed = validation.map(lambda image, label: normalize_img(image
    validation_processed = validation_processed.batch(batch_size)

    return train_processed, validation_processed
```

```
# Run preprocessing
train_processed_224, validation_processed_224 = preprocess_data(train, validation,
train_processed_331, validation_processed_331 = preprocess_data(train, validation,
```

5.Train All Models

Loop over each model by:

- download the pre-trained model without the output layers and freeze the weights.
- construct an empty `Sequential` model and first add the pre-trained model to it,
- add a single output Dense layer with `softmax` activation and compile it with `categorical_crossentropy` loss,
- train the model by calling `model.fit` for `3 epochs`,
- log the number of parameters (size of the model) and each model's final accuracy into a dictionary to visualize the results.

IMPORTANT:

The whole process takes ~2 hours on the standard `NVIDIA Tesla T4` GPU on Google Colaboratory Platform for ... **1 (ONE!)** epoch only. :)

```
EPOCHS = 1

# Loop over each model available in Keras
model_benchmarks = {'model_name': [], 'num_model_params': [], 'validation_accuracy
for model_name, model in tqdm(model_dictionary.items()):
    # Special handling for "NASNetLarge" since it requires input images with size
    print('\n**************************************')
    print(model_name)
    if 'NASNetLarge' in model_name:
        input_shape=(331,331,3)
        train_processed = train_processed_331
        validation_processed = validation_processed_331
    else:
        input_shape=(224,224,3)
        train_processed = train_processed_224
        validation_processed = validation_processed_224

    # load the pre-trained model with global average pooling as the last layer and
    pre_trained_model = model(include_top=False, pooling='avg', input_shape=input_
    pre_trained_model.trainable = False

    # custom modifications on top of pre-trained model
    clf_model = tf.keras.models.Sequential()
    clf_model.add(pre_trained_model)
    clf_model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
    clf_model.compile(loss='categorical_crossentropy', metrics=['accuracy'])
    history = clf_model.fit(train_processed, epochs=EPOCHS, validation_data=valida
                            steps_per_epoch=num_iterations)
```

```
# Calculate all relevant metrics
model_benchmarks['model_name'].append(model_name)
model_benchmarks['num_model_params'].append(pre_trained_model.count_params())
model_benchmarks['validation_accuracy'].append(history.history['val_accuracy'])
```

```
  0%|          | 0/28 [00:00<?, ?it/s]
****************************************
DenseNet121
508/508 [==============================] - 162s 304ms/step - loss: 0.1430 -

  4%|▍         | 1/28 [02:45<1:14:37, 165.83s/it]
****************************************
DenseNet169
508/508 [==============================] - 199s 371ms/step - loss: 0.1250 -

  7%|▋         | 2/28 [06:10<1:16:58, 177.63s/it]
****************************************
DenseNet201
508/508 [==============================] - 250s 468ms/step - loss: 0.0856 -

 11%|█         | 3/28 [10:28<1:24:00, 201.63s/it]
****************************************
EfficientNetB0
508/508 [==============================] - 93s 171ms/step - loss: 0.6975 -

 14%|█▍        | 4/28 [12:04<1:07:56, 169.85s/it]
****************************************
EfficientNetB1
Downloading data from https://storage.googleapis.com/keras-applications/eff
27025408/27018416 [==============================] - 1s 0us/step
508/508 [==============================] - 130s 237ms/step - loss: 0.6997 -

 18%|█▊        | 5/28 [14:22<1:01:25, 160.22s/it]
****************************************
EfficientNetB2
Downloading data from https://storage.googleapis.com/keras-applications/eff
31793152/31790344 [==============================] - 1s 0us/step
508/508 [==============================] - 137s 250ms/step - loss: 0.7014 -

 21%|██        | 6/28 [16:45<56:50, 155.04s/it]
****************************************
EfficientNetB3
Downloading data from https://storage.googleapis.com/keras-applications/eff
```

```
43941888/43941136 [==============================] - 1s 0us/step
508/508 [==============================] - 179s 328ms/step - loss: 0.7081 -
```

```
# train_time for 1 step = 32 images
train_time = [304, 371, 468, 171, 237, 250, 328, 439, 624, 804, 1100, 561, 231, 10
print(len(train_time))
```

```
28
```

```
model_benchmarks['model_time'] = train_time
```

6.Metrics and Model Analysis

To summarize the previous efforts we need:

- visualize the results by converting it to a DataFrame,
- sorting in ascending order for `num_model_params` since our goal is to select the smallest model with good enough accuracy.

In this example, the `MobileNet` model already provides `97%` accuracy; thus, we can directly use it.

`MobileNetV2` and `NASNetMobile` are two other models we can consider for fine-tuning experiments.

If our selected model's accuracy is still insufficient for our task, we can experiment by fine-tuning these selected models further, adding data augmentation, etc.

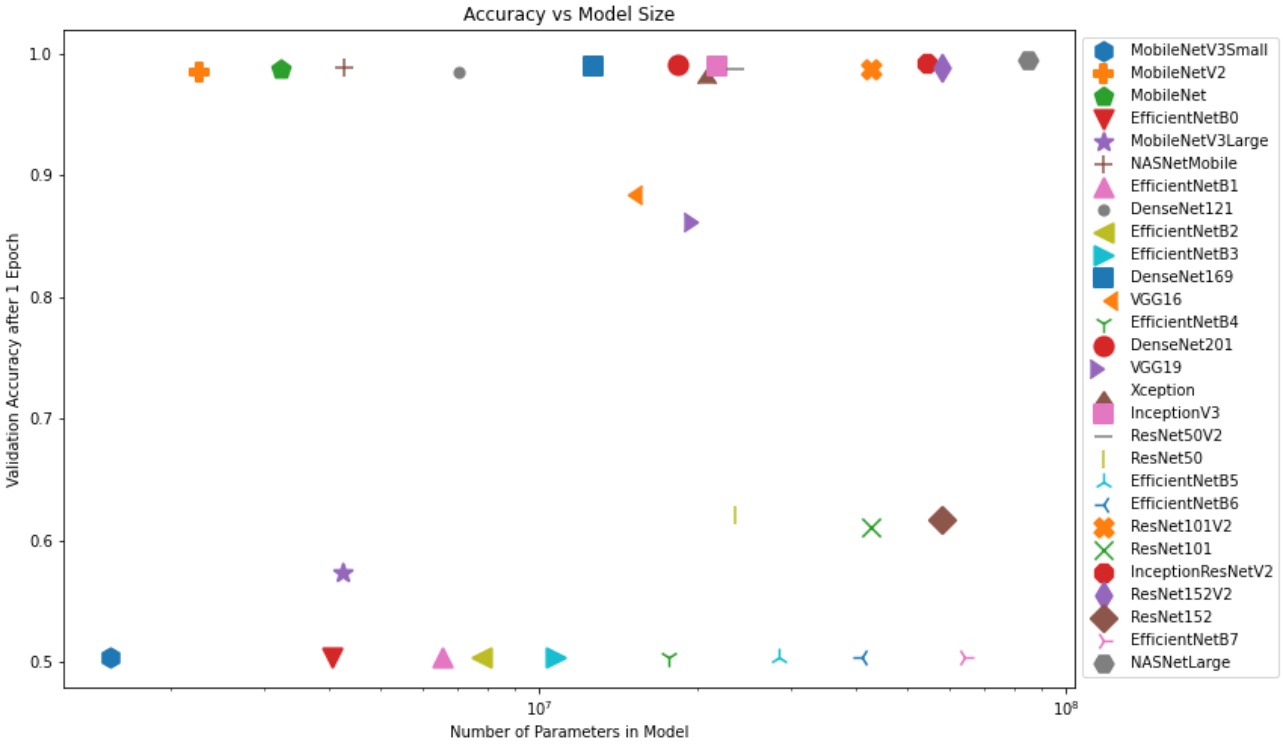
We will apply the typical DL model improvement experiments on a model that is already proven to be close to our requirements.

```
# Convert Results to DataFrame for easy viewing
benchmark_df = pd.DataFrame(model_benchmarks)
benchmark_df.sort_values('num_model_params', inplace=True) # sort in ascending ord
benchmark_df.to_csv('benchmark_df.csv', index=False) # write results to csv file
benchmark_df
```

	model_name	num_model_params	validation_accuracy	model_time
16	MobileNetV3Small	1529968	0.503797	91
14	MobileNetV2	2257984	0.984525	118
13	MobileNet	3228864	0.987391	102
3	EfficientNetB0	4049571	0.503797	171
15	MobileNetV3Large	4226432	0.573721	114
18	NASNetMobile	4269716	0.988967	204
4	EfficientNetB1	6575239	0.503797	237
0	DenseNet121	7037504	0.984238	304
5	EfficientNetB2	7768569	0.503797	250
6	EfficientNetB3	10783535	0.503797	328
1	DenseNet169	12642880	0.989827	371
25	VGG16	14714688	0.884511	333
7	EfficientNetB4	17673823	0.503797	439
2	DenseNet201	18321984	0.990830	468
26	VGG19	20024384	0.861728	393
27	Xception	20861480	0.987104	407
12	InceptionV3	21802784	0.989540	231
24	ResNet50V2	23564800	0.986674	268
23	ResNet50	23587712	0.620289	291
8	EfficientNetB5	28513527	0.503797	624
9	EfficientNetB6	40960143	0.503797	804
20	ResNet101V2	42626560	0.987677	473

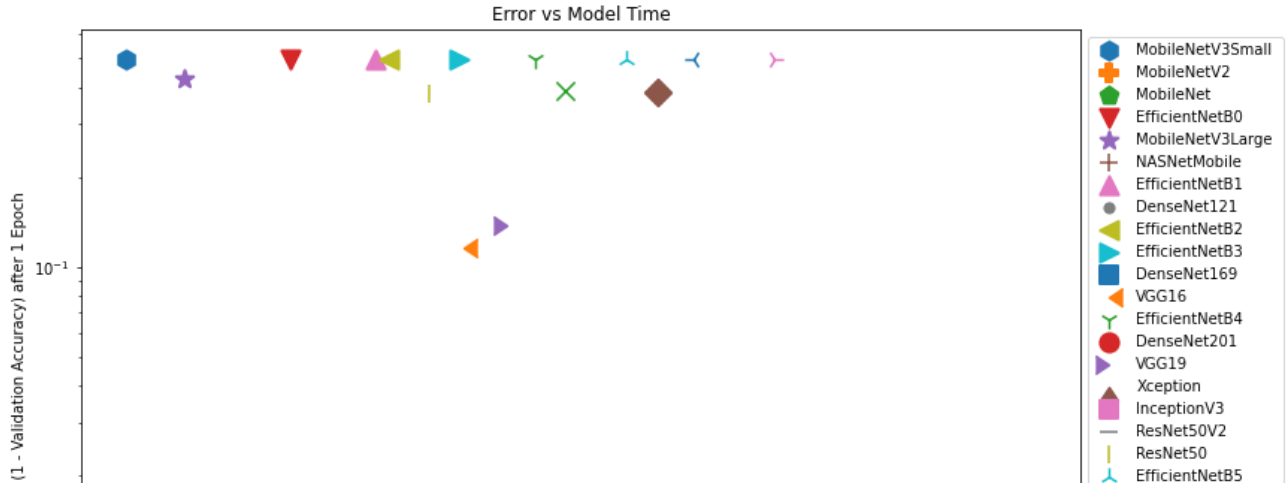
Plot - Accuracy vs. Log(Size)

```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.num_model_params, row.validation_accuracy, label=row.model_nam
plt.xscale('log')
plt.xlabel('Number of Parameters in Model')
plt.ylabel('Validation Accuracy after 1 Epoch')
plt.title('DogsCats - Accuracy vs Model Size')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```



Plot - Log(Error) vs. Log(Time)

```
# Loop over each row and plot the model_time vs error (1-validation_accuracy)
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.model_time, 1-row.validation_accuracy, label=row.model_name, m
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Model Training Time (per batch)')
plt.ylabel('Error (1 - Validation Accuracy) after 1 Epoch')
plt.title('DogsCats - Error vs Model Time')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```



Plot - Log(Time) vs. Log(Size)

```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.num_model_params, row.model_time, label=row.model_name, marker
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Model Size (Number of Parameters in Model)')
plt.ylabel('Model Training Time (per batch)')
plt.title('DogsCats - Model Time vs Size')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```

Conclusion

The presented **data-driven approach** allows Data Scientist to choose the most suitable model from among a group of pre-trained models in the Keras TF2 API.

In the most practical applications, Data Scientist would select the model with the least number of parameters that provided good enough accuracy for further exploration.

You can easily extend this approach to include other models not offered by Keras by manually adding items into the `model_dictionary`.

Part 2.Train + Validation on More Complex Dataset - CIFAR10

3.Dataset - `cifar10`

CIFAR is an acronym that stands for the Canadian Institute For Advanced Research and the CIFAR-10 dataset was developed along with the CIFAR-100 dataset (covered in the next section)

by researchers at the CIFAR institute.

The dataset is comprised of 60,000 32×32 pixel color photographs of objects from 10 classes, such as frogs, birds, cats, ships, etc.

These are very small images, much smaller than a typical photograph, and the dataset is intended for computer vision research.

CIFAR-10 is a dataset and was widely used for benchmarking computer vision algorithms in the field of machine learning. The problem is “solved.” Top performance on the problem is achieved by deep learning convolutional neural networks with a classification accuracy above 96% or 97% on the test dataset.

Download some images to run the experiment.

NOTE:

You could modify this step to load your data when you run the experiment for your use case.

```
# Download the training and validation data
#(train, validation), metadata = tfds.load('cifar10', split=['train[:70%]', 'train

# We use 10% of the whole dataset HERE to shorten this exercise time.
(train, validation), metadata = tfds.load('cifar10', split=['train[:7%]', 'train[7
                                          with_info=True, as_supervised=True)

# Number of training examples and labels
num_train = len(list(train))
num_validation = len(list(validation))
num_classes = len(metadata.features['label'].names)
num_iterations = int(num_train/batch_size)

# Print important info
print(f'Num train images: {num_train} \
        \nNum validation images: {num_validation} \
        \nNum classes: {num_classes} \
        \nNum iterations per epoch: {num_iterations}')
```

```
    Num train images: 3500
    Num validation images: 1500
    Num classes: 10
    Num iterations per epoch: 109
```

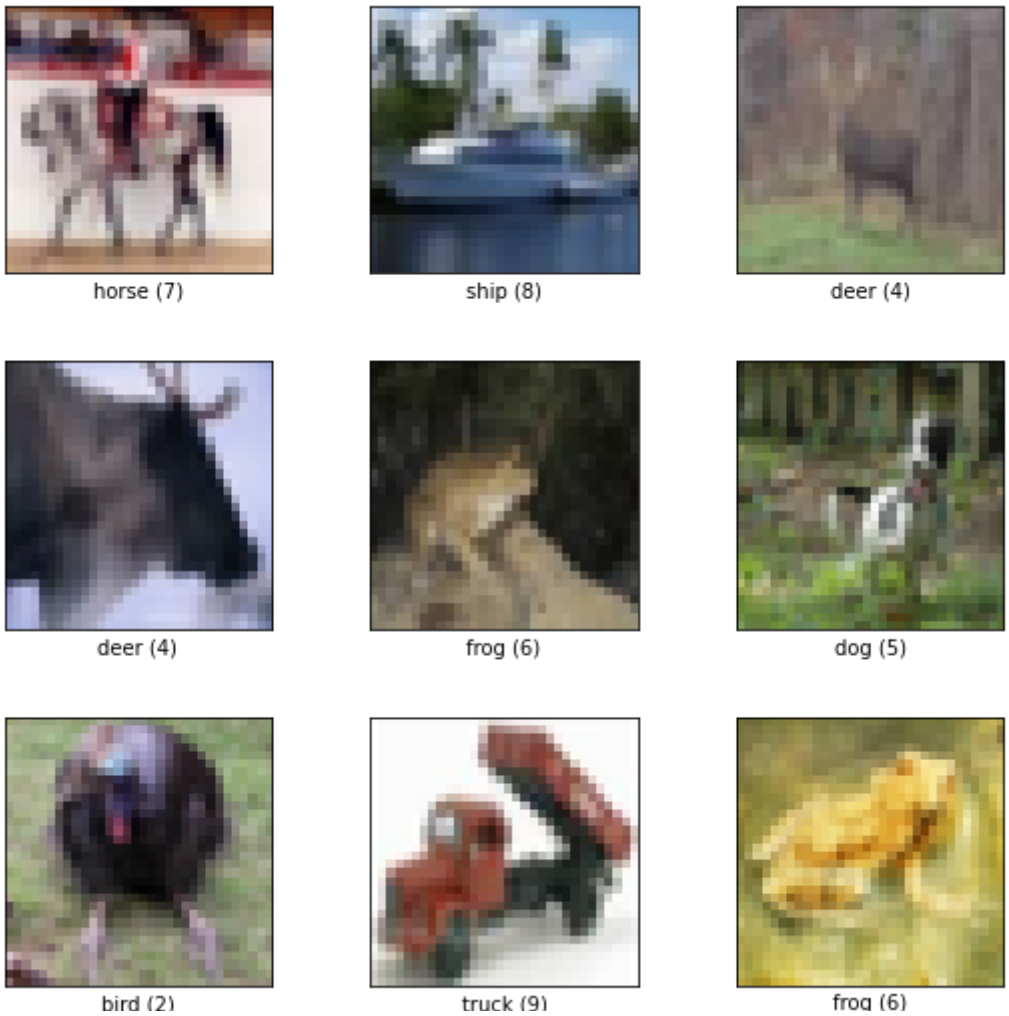
The example below creates a plot of the first nine images in the training dataset.

This function - `tfds.show_examples` - is for interactive use (Colab, Jupyter) **ONLY!**

It displays and return a plot of (rows*columns) images from a tf.data.Dataset.

```
fig = tfds.show_examples(train, metadata)
```

```
# show the figure
fig.show()
```



4.Image Pre-processing

- Images are **resized**, because some pre-trained models require images to be of size (224,224,3) while some require (331,331,3).
- Images are **normalized** them by dividing each pixel by 255.
- Labels are **one-hot encoded** so that we can use `categorical_crossentropy` loss during training.

```
%%time

def normalize_img(image, label, img_size):
    # Resize image to the desired img_size and normalize it
    # One hot encode the label
    image = tf.image.resize(image, img_size)
    image = tf.cast(image, tf.float32) / 255.
    label = tf.one_hot(label, depth=num_classes)
    return image, label

def preprocess_data(train, validation, batch_size, img_size):
    # Apply the normalize_img function on all train and validation data and create
```

```
    train_processed = train.map(lambda image, label: normalize_img(image, label, i
    train_processed = train_processed.batch(batch_size).repeat()

    validation_processed = validation.map(lambda image, label: normalize_img(image
    validation_processed = validation_processed.batch(batch_size)

    return train_processed, validation_processed

# Run preprocessing
train_processed_224, validation_processed_224 = preprocess_data(train, validation,
train_processed_331, validation_processed_331 = preprocess_data(train, validation,
```

```
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
CPU times: user 118 ms, sys: 15 ms, total: 133 ms
Wall time: 130 ms
```

5.Train All Models

Loop over each model by:

- download the pre-trained model without the output layers and freeze the weights.
- construct an empty `Sequential` model and first add the pre-trained model to it,
- add a single output Dense layer with `softmax` activation and compile it with `categorical_crossentropy` loss,
- train the model by calling `model.fit` for `3 epochs`,
- log the number of parameters (size of the model) and each model's final accuracy into a dictionary to visualize the results.

IMPORTANT:

The whole process takes ~35 min on the standard `NVIDIA Tesla T4` GPU on Google Colaboratory Platform for ... **1 (ONE!)** epoch and **10%** of the whole dataset only. :)

```
EPOCHS = 1

# Loop over each model available in Keras
model_benchmarks = {'model_name': [], 'num_model_params': [], 'validation_accuracy
for model_name, model in tqdm(model_dictionary.items()):
    # Special handling for "NASNetLarge" since it requires input images with size
    print('\n****************************************')
    print(model_name)
    if 'NASNetLarge' in model_name:
        input_shape=(331,331,3)
        train_processed = train_processed_331
        validation_processed = validation_processed_331
    else:
        input_shape=(224,224,3)
        train_processed = train_processed_224
        validation_processed = validation_processed_224

    # load the pre-trained model with global average pooling as the last layer and
    pre_trained_model = model(include_top=False, pooling='avg', input_shape=input_
    pre_trained_model.trainable = False

    # custom modifications on top of pre-trained model
    clf_model = tf.keras.models.Sequential()
    clf_model.add(pre_trained_model)
    clf_model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
    clf_model.compile(loss='categorical_crossentropy', metrics=['accuracy'])
    history = clf_model.fit(train_processed, epochs=EPOCHS, validation_data=valida
                            steps_per_epoch=num_iterations)

    # Calculate all relevant metrics
    model_benchmarks['model_name'].append(model_name)
    model_benchmarks['num_model_params'].append(pre_trained_model.count_params())
    model_benchmarks['validation_accuracy'].append(history.history['val_accuracy']
```

```
  0%|          | 0/28 [00:00<?, ?it/s]
****************************************
DenseNet121
109/109 [==============================] - 72s 320ms/step - loss: 1.9534 -
  4%|▌         | 1/28 [01:16<34:24, 76.45s/it]
```

```
****************************************
DenseNet169
109/109 [==============================] - 53s 391ms/step - loss: 1.7226 -
  7%|▋         | 2/28 [02:17<31:07, 71.84s/it]
****************************************
DenseNet201
109/109 [==============================] - 67s 499ms/step - loss: 1.5770 -
 11%|█         | 3/28 [03:34<30:34, 73.37s/it]
****************************************
EfficientNetB0
109/109 [==============================] - 28s 188ms/step - loss: 2.3129 -
 14%|█▍        | 4/28 [04:06<24:19, 60.82s/it]
****************************************
EfficientNetB1
109/109 [==============================] - 38s 255ms/step - loss: 2.3226 -
 18%|█▊        | 5/28 [04:48<21:12, 55.31s/it]
****************************************
EfficientNetB2
109/109 [==============================] - 40s 270ms/step - loss: 2.3287 -
 21%|██        | 6/28 [05:32<19:05, 52.05s/it]
****************************************
EfficientNetB3
109/109 [==============================] - 51s 355ms/step - loss: 2.3533 -
 25%|██▌       | 7/28 [06:29<18:40, 53.36s/it]
****************************************
EfficientNetB4
109/109 [==============================] - 66s 473ms/step - loss: 2.3562 -
 29%|██▊       | 8/28 [07:42<19:46, 59.33s/it]
****************************************
EfficientNetB5
109/109 [==============================] - 88s 655ms/step - loss: 2.3654 -
 32%|███▏      | 9/28 [09:21<22:31, 71.14s/it]
****************************************
EfficientNetB6
109/109 [==============================] - 113s 852ms/step - loss: 2.3852 -
 36%|███▌      | 10/28 [11:26<26:13, 87.40s/it]
****************************************
EfficientNetB7
109/109 [==============================] - 146s 1s/step - loss: 2.3916 - ac
 39%|███▉      | 11/28 [14:09<31:12, 110.17s/it]
****************************************
InceptionResNetV2
109/109 [==============================] - 79s 606ms/step - loss: 1.3040 -
 43%|████▎     | 12/28 [15:43<28:04, 105.29s/it]
****************************************
InceptionV3
109/109 [==============================] - 34s 251ms/step - loss: 1.5763 -
 46%|████▋     | 13/28 [16:23<21:25, 85.69s/it]
****************************************
MobileNet
109/109 [==============================] - 14s 108ms/step - loss: 1.6655 -
 50%|█████     | 14/28 [16:38<15:03, 64.54s/it]
```

```
# train_time for 1 step = 32 images
train_time = [320, 391, 499, 188, 255, 270, 355, 473, 655, 852, 1000, 606, 251, 10
print(len(train_time))
```

```
28
```

```
model_benchmarks['model_time'] = train_time
```

6.Metrics and Model Analysis

To summarize the previous efforts we need:

- visualize the results by converting it to a DataFrame,
- sorting in ascending order for `num_model_params` since our goal is to select the smallest model with good enough accuracy.

In this example, the `MobileNet` model already provides `97%` accuracy; thus, we can directly use it.

`MobileNetV2` and `NASNetMobile` are two other models we can consider for fine-tuning experiments.

If our selected model's accuracy is still insufficient for our task, we can experiment by fine-tuning these selected models further, adding data augmentation, etc.

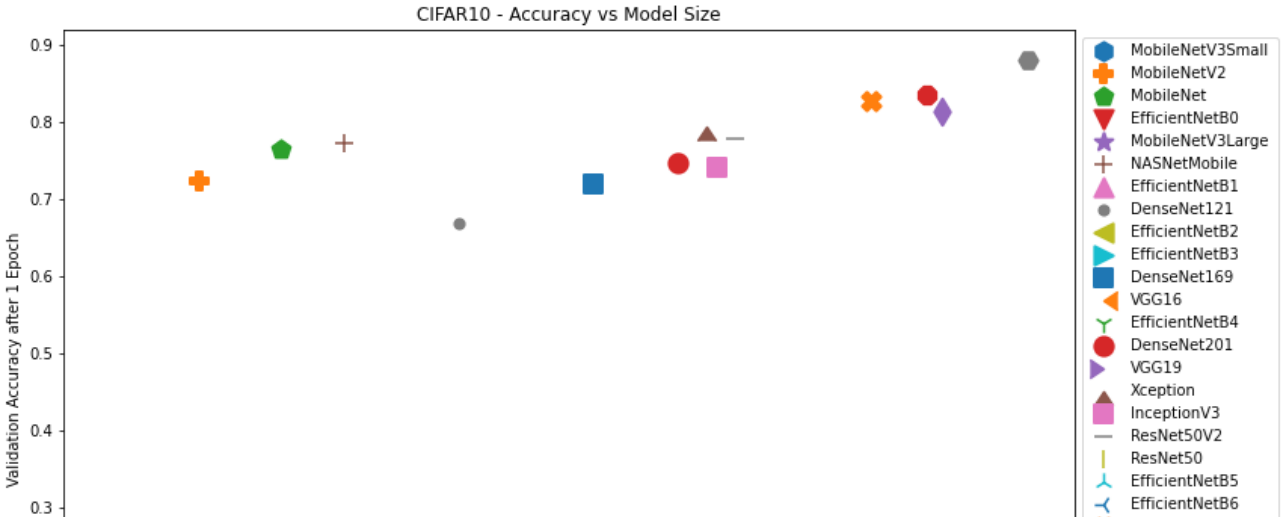
We will apply the typical DL model improvement experiments on a model that is already proven to be close to our requirements.

```
# Convert Results to DataFrame for easy viewing
benchmark_df = pd.DataFrame(model_benchmarks)
benchmark_df.sort_values('num_model_params', inplace=True) # sort in ascending ord
benchmark_df.to_csv('benchmark_df.csv', index=False) # write results to csv file
benchmark_df
```

	model_name	num_model_params	validation_accuracy	model_time
16	MobileNetV3Small	1529968	0.104667	68
14	MobileNetV2	2257984	0.723333	123
13	MobileNet	3228864	0.764667	108
3	EfficientNetB0	4049571	0.104667	188
15	MobileNetV3Large	4226432	0.104667	124
18	NASNetMobile	4269716	0.774000	234
4	EfficientNetB1	6575239	0.104667	255
0	DenseNet121	7037504	0.668667	320
5	EfficientNetB2	7768569	0.104667	270
6	EfficientNetB3	10783535	0.104667	355
1	DenseNet169	12642880	0.720000	391
25	VGG16	14714688	0.165333	415
7	EfficientNetB4	17673823	0.104667	473
2	DenseNet201	18321984	0.748000	499
26	VGG19	20024384	0.162667	398

Plot - Accuracy vs. Log(Size)

```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.num_model_params, row.validation_accuracy, label=row.model_nam
plt.xscale('log')
plt.xlabel('Number of Parameters in Model')
plt.ylabel('Validation Accuracy after 1 Epoch')
plt.title('CIFAR10 - Accuracy vs Model Size')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```

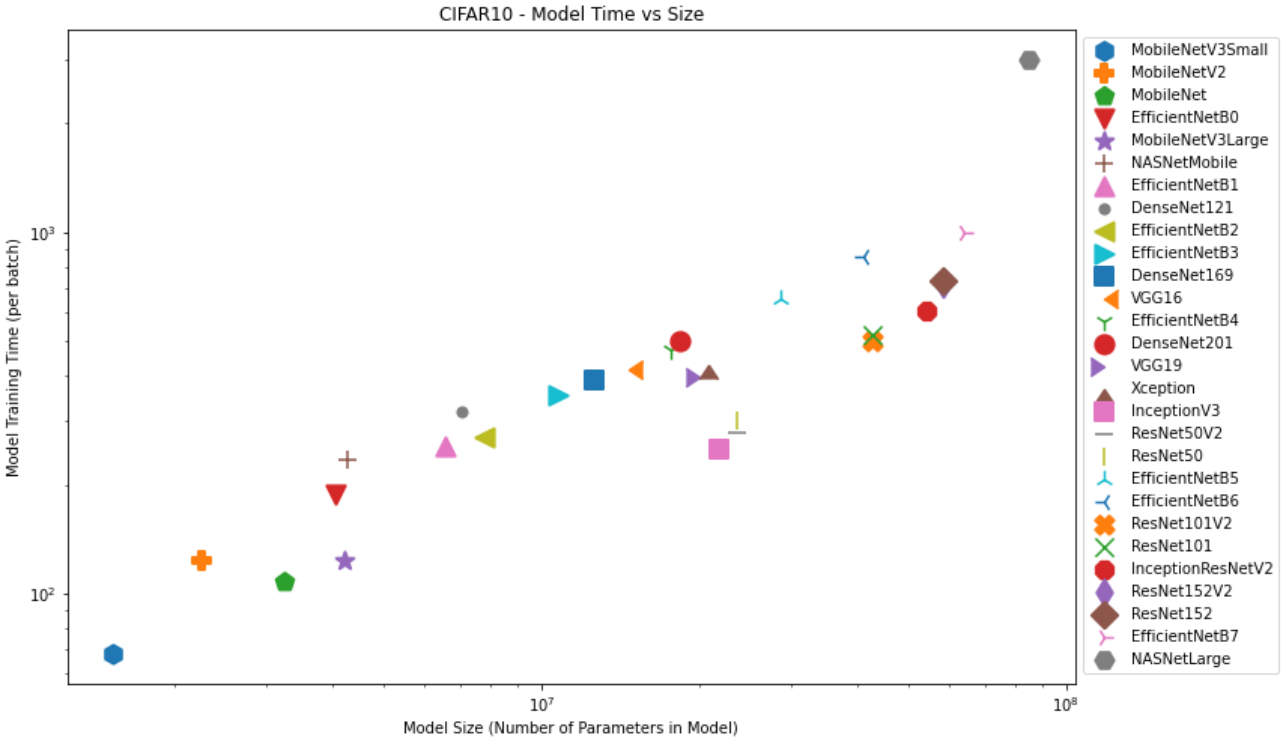


Plot - Log(Error) vs. Log(Time)

```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.model_time, 1-row.validation_accuracy, label=row.model_name, m
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Model Training Time (per batch)')
plt.ylabel('Error (1 - Validation Accuracy) after 1 Epoch')
plt.title('CIFAR10 - Error vs Model Time')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```

Plot - Log(Time) vs. Log(Size)


```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.num_model_params, row.model_time, label=row.model_name, marker
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Model Size (Number of Parameters in Model)')
plt.ylabel('Model Training Time (per batch)')
plt.title('CIFAR10 - Model Time vs Size')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```



Conclusion

The presented **data-driven approach** allows Data Scientist to choose the most suitable model from among a group of pre-trained models in the Keras TF2 API.

In the most practical applications, Data Scientist would select the model with the least number of parameters that provided good enough accuracy for further exploration.

You can easily extend this approach to include other models not offered by Keras by manually adding items into the `model_dictionary`

Part 3.Train + Validation on More Complex Dataset - CIFAR100

3.Dataset - `cifar100`

The CIFAR-100 dataset was prepared along with the CIFAR-10 dataset by academics at the Canadian Institute For Advanced Research (CIFAR).

The dataset is comprised of 60,000 32×32 pixel color photographs of objects from 100 classes, such as fish, flowers, insects, and much more.

Like CIFAR-10, the images are intentionally small and unrealistic photographs and the dataset is intended for computer vision research.

Download some images to run the experiment.

NOTE:

You could modify this step to load your data when you run the experiment for your use case.

```
# Download the training and validation data
#(train, validation), metadata = tfds.load('cifar100', split=['train[:70%]', 'trai

# We use 10% of the whole dataset HERE to shorten this exercise time.
(train, validation), metadata = tfds.load('cifar100', split=['train[:7%]', 'train[
                                          with_info=True, as_supervised=True)

# Number of training examples and labels
num_train = len(list(train))
num_validation = len(list(validation))
num_classes = len(metadata.features['label'].names)
num_iterations = int(num_train/batch_size)

# Print important info
print(f'Num train images: {num_train} \
        \nNum validation images: {num_validation} \
        \nNum classes: {num_classes} \
        \nNum iterations per epoch: {num_iterations}')
```

```
Num train images: 3500
Num validation images: 1500
```

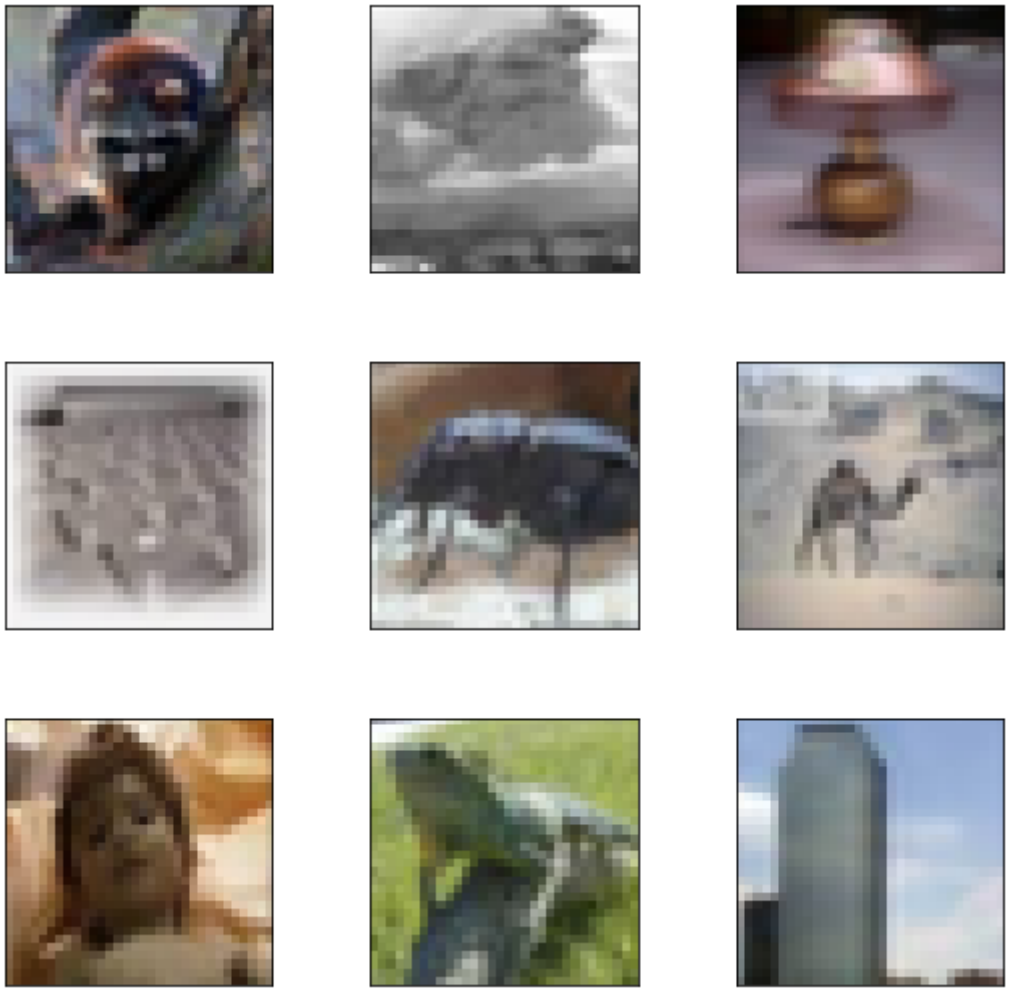
```
Num classes: 100
Num iterations per epoch: 109
```

The example below creates a plot of the first nine images in the training dataset.

This function - `tfds.show_examples` - is for interactive use (Colab, Jupyter) **ONLY!**

It displays and return a plot of (rows*columns) images from a tf.data.Dataset.

```
fig = tfds.show_examples(train, metadata)

# show the figure
fig.show()
```



4.Image Pre-processing

- Images are **resized**, because some pre-trained models require images to be of size (224,224,3) while some require (331,331,3).
- Images are **normalized** them by dividing each pixel by 255.
- Labels are **one-hot encoded** so that we can use `categorical_crossentropy` loss during training.

```
%%time
```

```
def normalize_img(image, label, img_size):
    # Resize image to the desired img_size and normalize it
    # One hot encode the label
    image = tf.image.resize(image, img_size)
    image = tf.cast(image, tf.float32) / 255.
    label = tf.one_hot(label, depth=num_classes)
    return image, label

def preprocess_data(train, validation, batch_size, img_size):
    # Apply the normalize_img function on all train and validation data and create
    train_processed = train.map(lambda image, label: normalize_img(image, label, i
    train_processed = train_processed.batch(batch_size).repeat()

    validation_processed = validation.map(lambda image, label: normalize_img(image
    validation_processed = validation_processed.batch(batch_size)

    return train_processed, validation_processed

# Run preprocessing
train_processed_224, validation_processed_224 = preprocess_data(train, validation,
train_processed_331, validation_processed_331 = preprocess_data(train, validation,
```

```
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function preprocess_data.<l
Cause: could not parse the source code of <function preprocess_data.<locals>.
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function preprocess_data.<locals>.<la
Cause: could not parse the source code of <function preprocess_data.<locals>.
```

```
To silence this warning, decorate the function with @tf.autograph.experimenta
CPU times: user 97.6 ms, sys: 12 ms, total: 110 ms
Wall time: 123 ms
```

5.Train All Models

Loop over each model by:

- download the pre-trained model without the output layers and freeze the weights.
- construct an empty `Sequential` model and first add the pre-trained model to it,
- add a single output Dense layer with `softmax` activation and compile it with `categorical_crossentropy` loss,
- train the model by calling `model.fit` for `3 epochs`,
- log the number of parameters (size of the model) and each model's final accuracy into a dictionary to visualize the results.

IMPORTANT:

The whole process takes ~32 min on the standard `NVIDIA Tesla T4` GPU on Google Colaboratory Platform for ... **1 (ONE!)** epoch and 10% of the whole dataset only. :)

```
EPOCHS = 1

# Loop over each model available in Keras
model_benchmarks = {'model_name': [], 'num_model_params': [], 'validation_accuracy
for model_name, model in tqdm(model_dictionary.items()):
    # Special handling for "NASNetLarge" since it requires input images with size
    print('\n**************************************')
    print(model_name)
    if 'NASNetLarge' in model_name:
        input_shape=(331,331,3)
        train_processed = train_processed_331
        validation_processed = validation_processed_331
    else:
        input_shape=(224,224,3)
        train_processed = train_processed_224
        validation_processed = validation_processed_224

    # load the pre-trained model with global average pooling as the last layer and
    pre_trained_model = model(include_top=False, pooling='avg', input_shape=input_
    pre_trained_model.trainable = False

    # custom modifications on top of pre-trained model
    clf_model = tf.keras.models.Sequential()
    clf_model.add(pre_trained_model)
    clf_model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
    clf_model.compile(loss='categorical_crossentropy', metrics=['accuracy'])
    history = clf_model.fit(train_processed, epochs=EPOCHS, validation_data=valida
                            steps_per_epoch=num_iterations)
```

```
# Calculate all relevant metrics
model_benchmarks['model_name'].append(model_name)
model_benchmarks['num_model_params'].append(pre_trained_model.count_params())
model_benchmarks['validation_accuracy'].append(history.history['val_accuracy'][
```

```
****************************************
InceptionV3
109/109 [==============================] - 31s 240ms/step - loss: 4.2334 -
 46%|████▌     | 13/28 [14:55<19:39, 78.62s/it]

****************************************
MobileNet
109/109 [==============================] - 13s 106ms/step - loss: 4.3385 -
 50%|█████     | 14/28 [15:09<13:51, 59.36s/it]
****************************************
MobileNetV2
109/109 [==============================] - 16s 120ms/step - loss: 4.3154 -
 54%|█████▎    | 15/28 [15:27<10:09, 46.87s/it]
****************************************
MobileNetV3Large
109/109 [==============================] - 20s 120ms/step - loss: 4.8150 -
 57%|█████▋    | 16/28 [15:49<07:54, 39.53s/it]
****************************************
MobileNetV3Small
109/109 [==============================] - 11s 63ms/step - loss: 4.7729 - a
 61%|██████    | 17/28 [16:02<05:46, 31.52s/it]
****************************************
NASNetLarge
109/109 [==============================] - 348s 3s/step - loss: 3.3606 - ac
 64%|██████▍   | 18/28 [22:02<21:40, 130.05s/it]
****************************************
NASNetMobile
109/109 [==============================] - 39s 231ms/step - loss: 4.0006 -
 68%|██████▊   | 19/28 [22:51<15:50, 105.65s/it]
****************************************
ResNet101
109/109 [==============================] - 62s 510ms/step - loss: 4.7730 -
 71%|███████   | 20/28 [23:57<12:30, 93.81s/it]
****************************************
ResNet101V2
109/109 [==============================] - 60s 490ms/step - loss: 3.7613 -
 75%|███████▍  | 21/28 [25:03<09:58, 85.55s/it]
****************************************
ResNet152
109/109 [==============================] - 91s 735ms/step - loss: 4.8159 -
 79%|███████▊  | 22/28 [26:40<08:54, 89.01s/it]
****************************************
ResNet152V2
109/109 [==============================] - 88s 719ms/step - loss: 3.7427 -
 82%|████████▏ | 23/28 [28:14<07:31, 90.39s/it]
****************************************
ResNet50
109/109 [==============================] - 38s 300ms/step - loss: 4.8275 -
 86%|████████▌ | 24/28 [28:54<05:01, 75.26s/it]
****************************************
ResNet50V2
109/109 [==============================] - 34s 278ms/step - loss: 3.8281 -
 89%|████████▉ | 25/28 [29:30<03:10, 63.40s/it]
****************************************
VGG16
```

```
109/109 [==============================] - 37s 333ms/step - loss: 4.6837 -
 93%|█████████ | 26/28 [30:07<01:51, 55.60s/it]
**************************************
VGG19
```

```
# train_time for 1 step = 32 images
train_time = [315, 394, 497, 180, 252, 273, 344, 464, 641, 834, 1000, 593, 240, 10
print(len(train_time))
```

```
28
```

```
model_benchmarks['model_time'] = train_time
```

6.Metrics and Model Analysis

To summarize the previous efforts we need:

- visualize the results by converting it to a DataFrame,
- sorting in ascending order for `num_model_params` since our goal is to select the smallest model with good enough accuracy.

In this example, the `MobileNet` model already provides `97%` accuracy; thus, we can directly use it.

`MobileNetV2` and `NASNetMobile` are two other models we can consider for fine-tuning experiments.

If our selected model's accuracy is still insufficient for our task, we can experiment by fine-tuning these selected models further, adding data augmentation, etc.

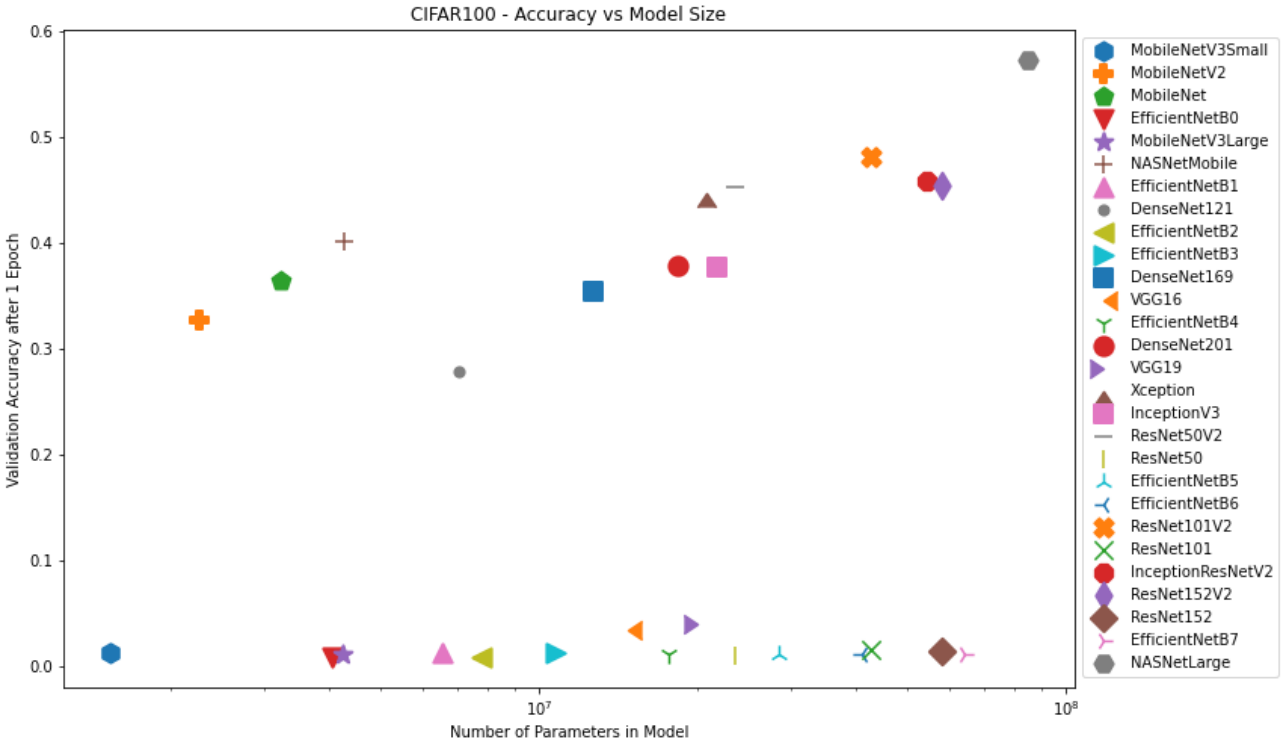
We will apply the typical DL model improvement experiments on a model that is already proven to be close to our requirements.

```
# Convert Results to DataFrame for easy viewing
benchmark_df = pd.DataFrame(model_benchmarks)
benchmark_df.sort_values('num_model_params', inplace=True) # sort in ascending ord
benchmark_df.to_csv('benchmark_df.csv', index=False) # write results to csv file
benchmark_df
```

	model_name	num_model_params	validation_accuracy	model_time
16	MobileNetV3Small	1529968	0.012000	63
14	MobileNetV2	2257984	0.327333	120
13	MobileNet	3228864	0.364667	106
3	EfficientNetB0	4049571	0.008667	180
15	MobileNetV3Large	4226432	0.011333	120
18	NASNetMobile	4269716	0.400667	231
4	EfficientNetB1	6575239	0.012667	252
0	DenseNet121	7037504	0.278000	315
5	EfficientNetB2	7768569	0.008000	273
6	EfficientNetB3	10783535	0.012667	344
1	DenseNet169	12642880	0.354667	394
25	VGG16	14714688	0.034000	333
7	EfficientNetB4	17673823	0.011333	464
2	DenseNet201	18321984	0.378667	497
26	VGG19	20024384	0.040000	397
27	Xception	20861480	0.446667	417
12	InceptionV3	21802784	0.377333	240
24	ResNet50V2	23564800	0.452000	278
23	ResNet50	23587712	0.009333	300
8	EfficientNetB5	28513527	0.011333	641
9	EfficientNetB6	40960143	0.011333	834
20	ResNet101V2	42626560	0.481333	490

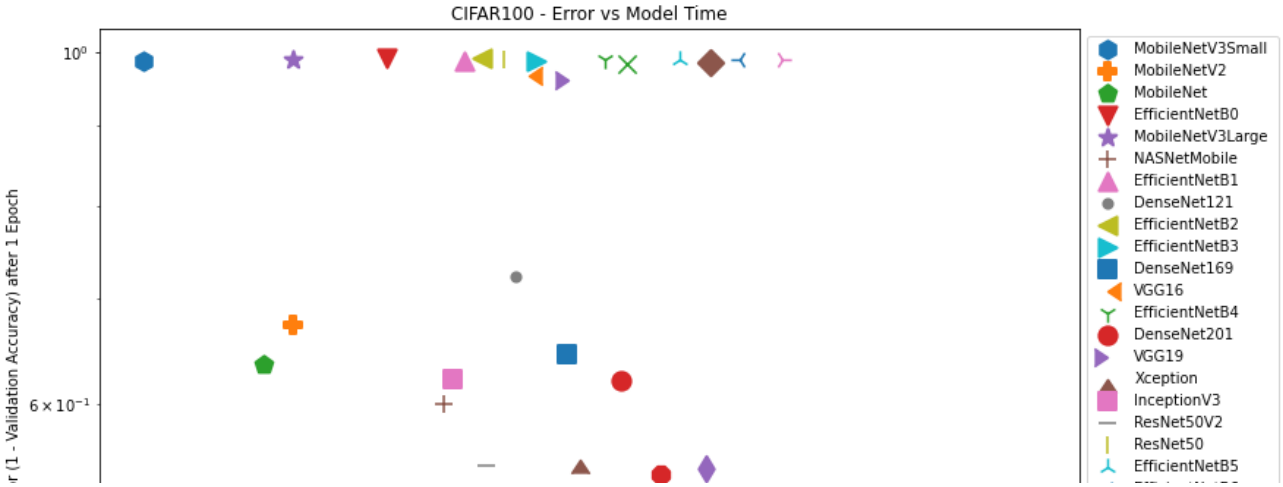
Plot - Accuracy vs. Log(Size)

```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.num_model_params, row.validation_accuracy, label=row.model_nam
plt.xscale('log')
plt.xlabel('Number of Parameters in Model')
plt.ylabel('Validation Accuracy after 1 Epoch')
plt.title('CIFAR100 - Accuracy vs Model Size')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```

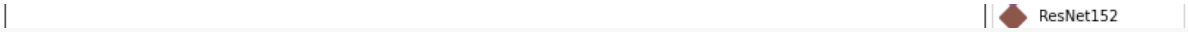


Plot - Log(Error) vs. Log(Time)

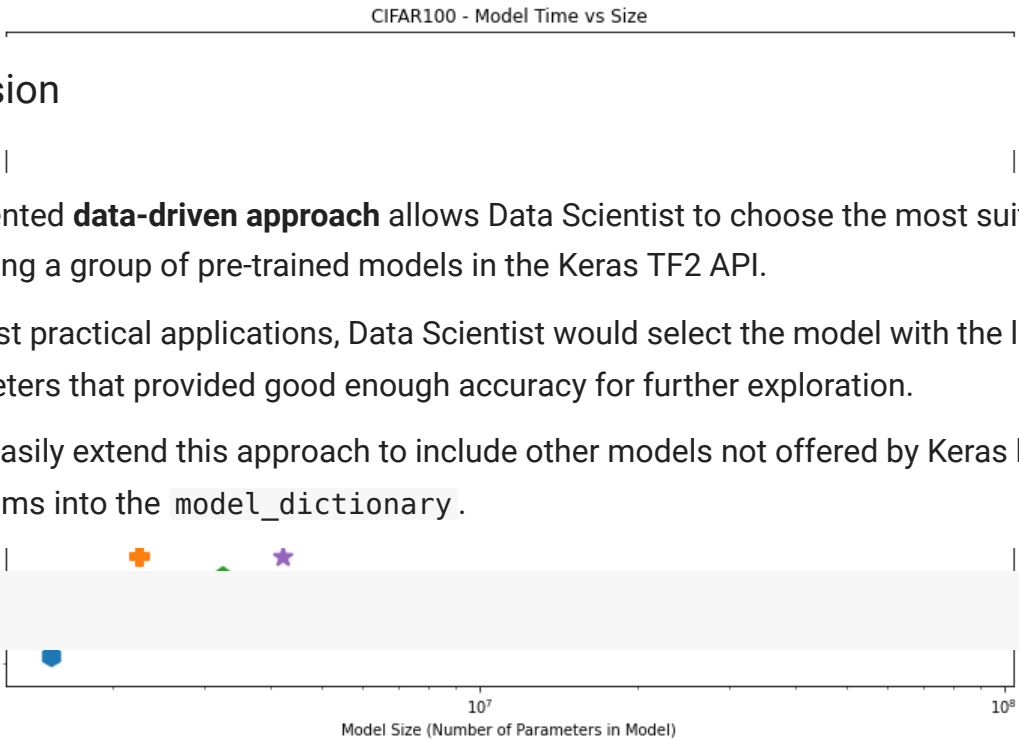
```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.model_time, 1-row.validation_accuracy, label=row.model_name, m
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Model Training Time (per batch)')
plt.ylabel('Error (1 - Validation Accuracy) after 1 Epoch')
plt.title('CIFAR100 - Error vs Model Time')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```



Plot - Log(Time) vs. Log(Size)




```
# Loop over each row and plot the num_model_params vs validation_accuracy
markers=[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","
plt.figure(figsize=(12,8))
for row in benchmark_df.itertuples():
    plt.scatter(row.num_model_params, row.model_time, label=row.model_name, marker
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Model Size (Number of Parameters in Model)')
plt.ylabel('Model Training Time (per batch)')
plt.title('CIFAR100 - Model Time vs Size')
plt.legend(bbox_to_anchor=(1, 1), loc='upper left'); # Move legend out of the plot
```



Conclusion

The presented **data-driven approach** allows Data Scientist to choose the most suitable model from among a group of pre-trained models in the Keras TF2 API.

In the most practical applications, Data Scientist would select the model with the least number of parameters that provided good enough accuracy for further exploration.

You can easily extend this approach to include other models not offered by Keras by manually adding items into the `model_dictionary`.

Нейронні мережі

Лекція_11

********** **********

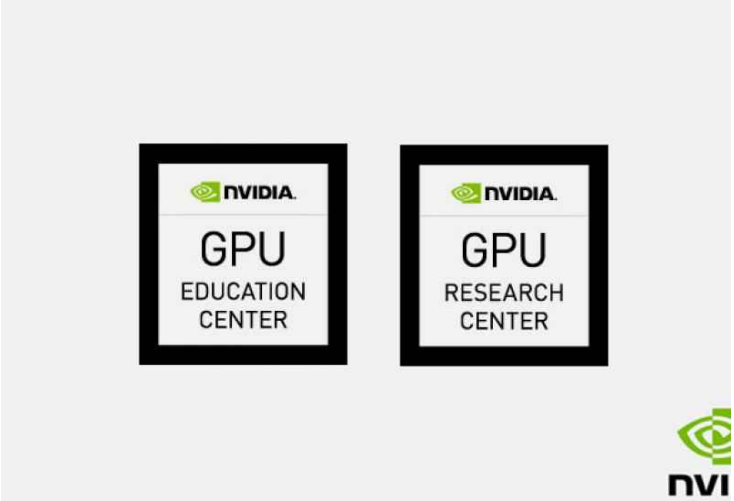
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 11 - Нейронні мережі - Сучасні CNN – Виявлення об'єктів

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1
Виявлення об'єктів
https://drive.google.com/file/d/1Dbub8UknvYD2463t3U1GjtmpMDeQWRwm/view?usp=sharing

Lecture 11 - Object Detection

(C) partially based on the works by d2l Open Source book authors, TFHub contributors, Planche, Martinez, Asadi, ...

Beside classification tasks, the other very important and canonical task is object detection. Object detection is used for detecting objects and their position in an image, for example, from self-driving cars to content moderation.

This DEMO will introduce techniques used for object detection with some attention to introduction of:

- object detection (OD) techniques,
- main OD approaches,
- OD from scratch (based on d2l Open Source book),
- using any image classifier as OD,
- using end-to-end OD,
- fast OD using YOLO architecture,
- improvements of OD by using Faster R-CNN architecture and related,
- using Faster R-CNN with the TensorFlow Object Detection API,
- using ODs from TFHub.

Part 0.Preparatory Actions

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
! cp /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/*.zip
! cp /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/*.jpg
! echo In /content:
! ls /content

! cp -r /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/te
! echo In test_images:
! ls /content/test_images

! cp -r /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/re
! echo In resources:
! ls /content/resources
```

```
In /content:
banana.jpg  detections_apple_87.jpg   detections_orange_93.jpg  fruits.zip
cafe.jpg    detections_banana_81.jpg  dog_example.jpg           me_dogs.jpg
```

```
cars.jpg    detections_banana_88.jpg  dog.jpg                    sample_data
catdog.jpg  detections_banana_91.jpg  drive
cat.jpg     detections_mixed_22.jpg   elephants.jpg
In test_images:
cafe.jpg  dog.jpg        me_dogs.jpg      result_dog.png
cars.jpg  elephants.jpg  result_cars.png
In resources:
anchors.json     mscoco_label_map.pbtxt
coco_labels.txt  ssd_efficientdet_d0_512x512_coco17_tpu8.txt
label_map.txt
```

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/d0/1f/13de7e8cafaba15
     |████████████████████████████████| 81kB 6.6MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: traitlets>=4.3.1 in /usr/local/lib/python3.7
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: nbformat>=4.2.0 in /usr/local/lib/python3.7/
Requirement already satisfied: ipython>=4.0.0; python_version >= "3.3" in /
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: jupyter-core>=4.4.0 in /usr/local/lib/python
Requirement already satisfied: jupyter-client>=5.2.0 in /usr/local/lib/pyth
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
```

```
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7/
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loca
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/lib
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dist
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pack
```

Part 1.Introduction to Object Detection - d2l Example

The section numeration in this Part is based on numeration of the original materials from [d2l Open Source book](#).

Please, use this Open Source book as a reference for any related questions!

Section 13.3.Object Detection and Bounding Boxes

In the previous section, we introduced many models for image classification.

In image ***classification*** tasks, we assume that:

- there is only **one** main **target** in the image,
- we only focus on how to **identify** the target **category**.

However, in many situations, there are:

- **multiple targets** in the image,
- we need **not only to classify** them, but also want to obtain their **specific positions** in the image.

In computer vision, we refer to such tasks as ***object detection*** (or object recognition).

Object detection is widely used in many fields:

- in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles in the captured video image,
- in manufacturing, robots often perform this type of task to detect targets of interest.
- in security, we need to detect abnormal targets, such as intruders or bombs.

In the next few sections, we will introduce multiple DL models used for object detection.

Import libraries

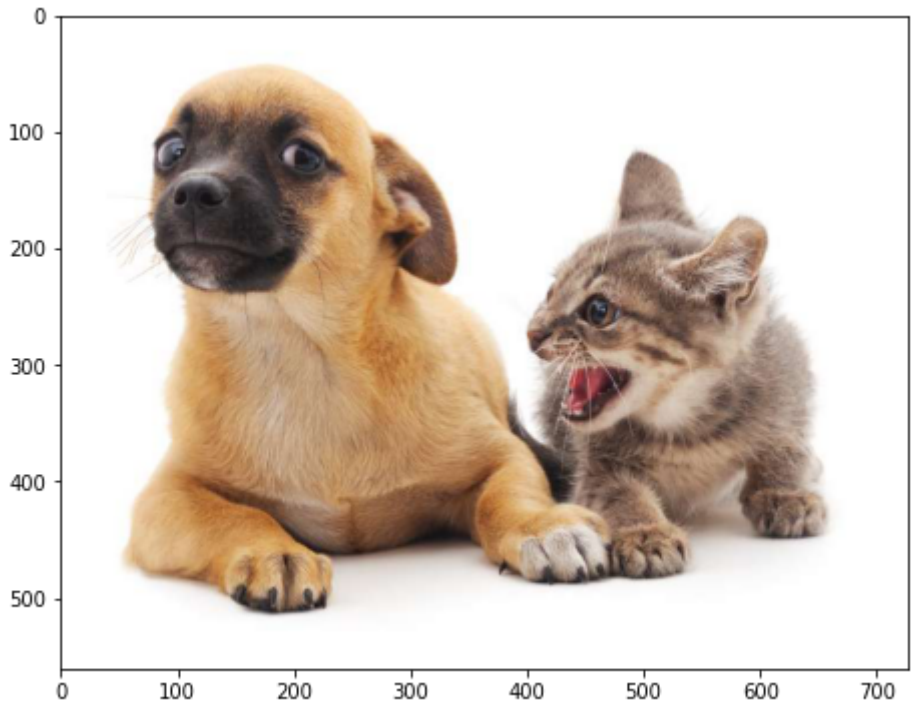
```
%matplotlib inline

# We'll use PyTorch in this Part! :)
import torch
#from d2l import torch as d2l
import matplotlib.pyplot as plt
```

Load Example Image

In this sample image we can see there is a dog on the left side of the image and a cat on the right.

They are the **two main targets** in this image.

```
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 10})
plt.grid(False)
image = plt.imread('./catdog.jpg')
plt.imshow(image);
```



Bounding Box

In object detection, we usually use a bounding box to describe the target location.

The **bounding box** is a rectangular box that can be determined by:

- **the two-corner** representation -> the x and y axis coordinates in the **upper-left corner** and the x and y axis coordinates in the **lower-right corner** of the rectangle,

OR

- **center-width-height** representation -> the x and y axis coordinates of the **bounding box center**, and its **width** and **height**.

Below we define functions to convert between these two representations:

- `box_corner_to_center` converts from **the two-corner** representation to the **center-width-height** presentation,

AND

- `box_center_to_corner` vice verse.

The input argument `boxes` can be:

- either a length 4 tensor,

or

- a $(N, 4)$ 2-dimensional tensor.

Box Conversion Functions

```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper_left, bottom_right) to (center, width, height)"""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = torch.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper_left, bottom_right)"""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

Define Bounding Boxes for Example Image

Let's define the **bounding boxes** of the dog and the cat in the image based on the coordinate information.

The **origin** of the coordinates in the image is the **upper left corner** of the image, and to the **right** and **down** are the **positive directions** of the x axis and the y axis, respectively.

```
# bbox is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```

Let's **verify** the correctness of box conversion functions by **converting twice**.

```
boxes = torch.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) - boxes
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

Visualize Bounding Boxes on Example Image

Let's draw the **bounding box** in the image to check if it is accurate.

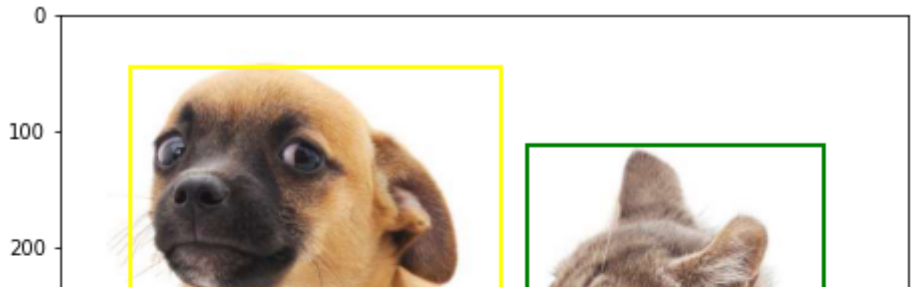
Before drawing the box, we will define a helper function `bbox_to_rect`. It represents the bounding box in the bounding box format of `matplotlib`.

```
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (top-left x, top-left y, bottom-right x,
    # bottom-right y) format to matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return plt.Rectangle(xy=(bbox[0], bbox[1]), width=bbox[2] - bbox[0],
                         height=bbox[3] - bbox[1], fill=False,
                         edgecolor=color, linewidth=2)
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

After loading the bounding box on the image, we can see that the main outline of the target is basically inside the box.

```
plt.figure(figsize=(8, 6))
fig = plt.imshow(image)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'yellow'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'green'));
```



Summary

- In object detection, we **not only need to identify all the objects** of interest in the image, **but also their positions**. The positions are generally represented by a **rectangular bounding box**.

Section 13.4.Anchor Boxes

Object detection algorithms usually:

- sample a **large number of regions** in the input image,
- determine **whether these regions contain objects** of interest, and
- **adjust the edges of the regions** so as to predict the **ground-truth (GT)** (namely, real!) bounding box of the target more accurately.

Different models may use different region sampling methods.

Let's consider one of these method:

*it generates **multiple** bounding boxes with **different** sizes and **aspect** ratios while centering on **each pixel**.*

These bounding boxes are called **anchor boxes**.

Let's practice with object detection based on anchor boxes in the following sections.

We need to modify the printing accuracy of PyTorch. Because printing tensors actually calls the print function of PyTorch, the floating-point numbers in tensors printed in this section are more concise.

```
torch.set_printoptions(2)
```

Generating Multiple Anchor Boxes

Assume that the input image has a height of h and width of w.

REMEMBER: *We generate anchor boxes with different shapes centered on each pixel of the image.*

ASSUME for the anchor box:

- the **size** is $s \in (0, 1]$,
- the **aspect ratio** is $r > 0$,

THEN:

- the **width** and **height** of the anchor box are $ws\sqrt{r}$ and $hs/\sqrt{r}$, respectively.

When the center position is given, an anchor box with known width and height is determined.

Below we set:

- a set of sizes $s_1, \ldots, s_n$

and

- a set of aspect ratios $r_1, \ldots, r_m$.

If we use a combination of all sizes and aspect ratios with each pixel as the center, the input image will have a total of $whnm$ anchor boxes.

Although these anchor boxes may **cover all** ground-truth bounding boxes, **the computational complexity is often excessive**.

But, we are usually only interested in a combination containing s_1 or r_1 sizes and aspect ratios, that is:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1).$$

That is, the number of anchor boxes centered on the same pixel is $n + m - 1$. For the entire input image, we will generate a total of $wh(n + m - 1)$ anchor boxes.

Generate Anchor Boxes

The above method of generating anchor boxes has been implemented in the `multibox_prior` function where we specify:

- `data` - the input,
- `sizes` - a set of sizes,
- `ratios` - a set of aspect ratios,

and this function will return all the **anchor boxes** entered.

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
#@save
def multibox_prior(data, sizes, ratios):
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = torch.tensor(sizes, device=device)
    ratio_tensor = torch.tensor(ratios, device=device)
    # Offsets are required to move the anchor to center of a pixel
```

```
    # Since pixel (height=1, width=1), we choose to offset our centers by 0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height  # Scaled steps in y axis
    steps_w = 1.0 / in_width  # Scaled steps in x axis

    # Generate all center points for the anchor boxes
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w)
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    # Generate boxes_per_pixel number of heights and widths which are later
    # used to create anchor box corner coordinates (xmin, xmax, ymin, ymax)
    # cat (various sizes, first ratio) and (first size, various ratios)
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:])))\
                   * in_height / in_width  # handle rectangular inputs
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    # Divide by 2 to get half height and half width
    anchor_manipulations = torch.stack(
        (-w, -h, w, h)).T.repeat(in_height * in_width, 1) / 2

    # Each center point will have boxes_per_pixel number of anchor boxes, so
    # generate grid of all anchor box centers with boxes_per_pixel repeats
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                           dim=1).repeat_interleave(boxes_per_pixel, dim=0)

    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)
```

We can see that the shape of the returned anchor box variable `y` is (batch size, number of anchor boxes, 4).

```
img = plt.imread('./catdog.jpg')
h, w = img.shape[0:2]

print(h, w)
X = torch.rand(size=(1, 3, h, w))  # Construct input data
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
torch.Size([1, 2042040, 4])
```

After changing the shape of the anchor box variable `y`

to

(image height, image width, number of anchor boxes centered on the same pixel, 4),

we can obtain all the anchor boxes centered on a specified pixel position.

In the following example, we access the first anchor box centered on (250, 250).

It has 4 elements:

- the x, y axis coordinates in the upper-left corner

and

- the x, y axis coordinates in the lower-right corner of the anchor box.

The coordinate values of the x and y axis **are divided by the width and height of the image**, respectively, so the value range is between 0 and 1.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

```
tensor([0.06, 0.07, 0.63, 0.82])
```

Draw Multiple Bounding Boxes on Example Image

In order to describe all anchor boxes centered on one pixel in the image, we first define the `show_bboxes` function to draw multiple bounding boxes on the image.

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

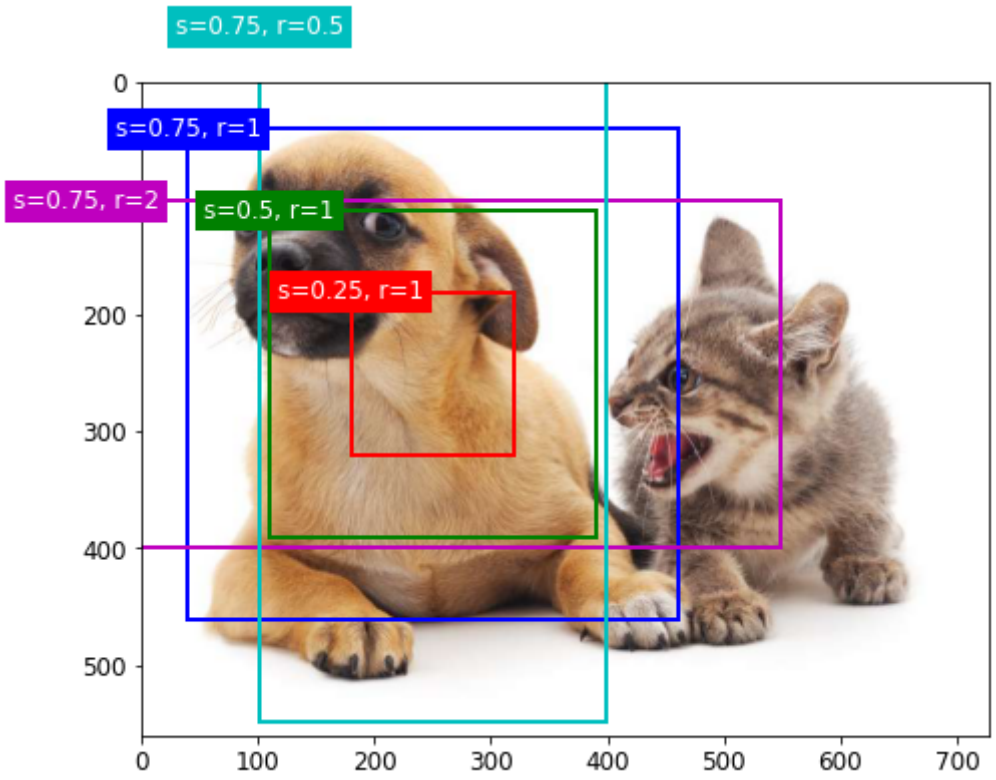
```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = _make_list(labels)
    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = bbox_to_rect(bbox.detach().numpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i], va='center',
                      ha='center', fontsize=12, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

As we just saw, the coordinate values of the x and y axis in the variable `boxes` have been divided by the width and height of the image, respectively.

When drawing images, we need to restore the original coordinate values of the anchor boxes and therefore define the variable `bbox_scale`.

Now, we can draw all the anchor boxes centered on (250, 250) in the image. As you can see, the blue anchor box with a size of 0.75 and an aspect ratio of 1 covers the dog in the image well.

```
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 12})
bbox_scale = torch.tensor((w, h, w, h))
fig = plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale, [
    's=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2', 's=0.75, r=0.5'
])
```



Metric -> Intersection over Union

If the ground-truth (GT) bounding box (BB) of the target is known, **how we can measure** that the anchor box (AB) covers the dog in the image well?

An intuitive method is **to measure the similarity** between ABs and GT BB.

Jaccard index measures the similarity between two sets.

Given sets $\mathcal{A}$ and $\mathcal{B}$, their **Jaccard index** is the size of their **intersection divided by** the size of their **union**:

$$J(\mathcal{A},\mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}.$$

In fact, we can consider the pixel area of a bounding box as a collection of pixels.

That is why, we can measure the similarity of the two BBs by the Jaccard index of their pixel sets.

When we measure the similarity of 2 BBs, we usually refer the Jaccard index as **intersection over union (IoU)**, which is the ratio of the intersecting area to the union area of the 2 BBs, as shown in **Figure** below.

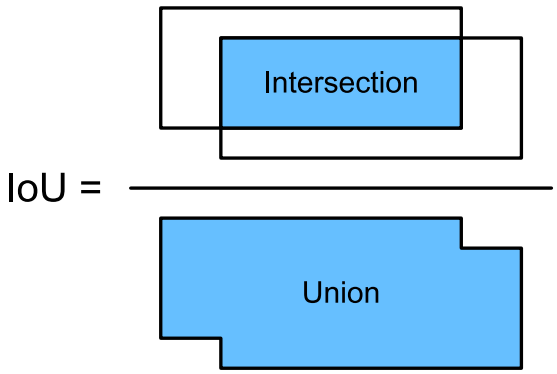


Figure. IoU is the ratio of the intersecting area to the union area of 2 BBs.

The value range of IoU is between 0 and 1:

- **0** means that there are **no overlapping** pixels between the 2 BBs,
- **1** means that the 2 BBs are **equal**.

Below we will use IoU to measure the similarity between ABs and GT BBs, and between different

Measure IoU

```
#@save
def box_iou(boxes1, boxes2):
    """Compute IOU between two sets of boxes of shape (N,4) and (M,4)."""
    # Compute box areas
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    area1 = box_area(boxes1)
    area2 = box_area(boxes2)
    lt = torch.max(boxes1[:, None, :2], boxes2[:, :2])  # [N,M,2]
    rb = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])  # [N,M,2]
    wh = (rb - lt).clamp(min=0)  # [N,M,2]
    inter = wh[:, :, 0] * wh[:, :, 1]  # [N,M]
    unioun = area1[:, None] + area2 - inter
    return inter / unioun
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

Mean Average Precision - mAP

Why compute such a **fraction** and not just use the **intersection**?

While the **intersection** would provide a good indicator of how much two sets/boxes overlap, this value is absolute and **not relative**.

Therefore, two big boxes would probably overlap by many more pixels than two small boxes. This is why this ratio is used—it will always be between 0 (if the two boxes do not overlap) and 1 (if two boxes overlap completely).

When computing the average precision, we say that two boxes overlap if their IoU is above a certain threshold. The threshold usually chosen is 0.5.

Average precision gives information about the performance of a model for a single class. To get a global score, we use **mean Average Precision (mAP)**. This corresponds to the mean of the average precision for each class. If the dataset has 10 classes, we will compute the average precision for each class and take the average of those numbers.

Mean average precision is used in at least two object detection challenges:

- PASCAL Visual Object Classes (usually referred to as Pascal VOC),
- and Common Objects in Context (usually referred to as COCO).

The latter is larger and contains more classes; therefore, the scores obtained are usually lower than for the former.

Notations and traditions:

- For the Pascal VOC challenge, 0.5 is also used—we say that we use **mAP@0.5** (pronounced **mAP at 0.5**).
- For the COCO challenge, a slightly different metric is used **mAP@[0.5:0.95]**. This means that we compute mAP@0.5, mAP@0.55, ..., mAP@0.95, and take the average.

Averaging over IoUs rewards models with better localization

Labeling Training Set Anchor Boxes

In the training set, we consider **each AB** as a **training example**.

To **train the object detection** model, we need to mark two types of labels for each anchor box:

- first, the **category** of the target contained in the AB,
- second, the **offset** of the GT BB relative to the AB.

In object detection, we:

- **generate** multiple ABs,
- **predict** the categories and offsets for each AB,
- **adjust** the AB position according to the predicted offset to obtain the BBs to be used for prediction, and
- finally filter out the prediction BBs that need to be output.

In the object detection training set, each image is **labelled** with the **location** of the GT BB and the **category** of the target contained.

After the ABs are generated, we primarily label ABs based on the location and category information of the GT BBs similar to the ABs.

How do we assign GT BBs to ABs similar to them?

Assume that

- the ABs in the image are $A_1, A_2, \ldots, A_{n_a}$ and
- the GT BBs are $B_1, B_2, \ldots, B_{n_b}$ and $n_a \geq n_b$.

Let's define matrix

$\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$,

where element x_{ij} in the i^{th} row and j^{th} column is the IoU of the AB A_i to the GT BB B_j.

We need:

- to find the largest element in the matrix $\mathbf{X}$ and record the row index and column index of the element as i_1, j_1,
- to assign the GT BB B_{j_1} to the AB A_{i_1},

NOTE: AB A_{i_1} and GT BB B_{j_1} have the highest similarity among all the "AB--GT BB" pairings.

- discard all elements in the i_1th row and the j_1th column in the matrix $\mathbf{X}$,
- find the largest remaining element in the matrix $\mathbf{X}$ and record the row index and column index of the element as i_2, j_2,
- assign GT BB B_{j_2} to AB A_{i_2} and then discard all elements in the i_2th row and the j_2th column in the matrix $\mathbf{X}$,

NOTE: At this point, elements in two rows and two columns in the matrix $\mathbf{X}$ have been discarded.

- proceed until all elements in the n_b column in the matrix $\mathbf{X}$ are discarded,

NOTE: At this time, we have assigned a ground-truth bounding box to each of the n_b anchor boxes.

- only traverse the remaining $n_a - n_b$ ABs. Given AB A_i, find the BB B_j with the largest IoU with A_i according to the i^{th} row of the matrix $\mathbf{X}$, and only assign GT BB B_j to AB A_i when the IoU is greater than the predetermined threshold.

Example (in the figure below):

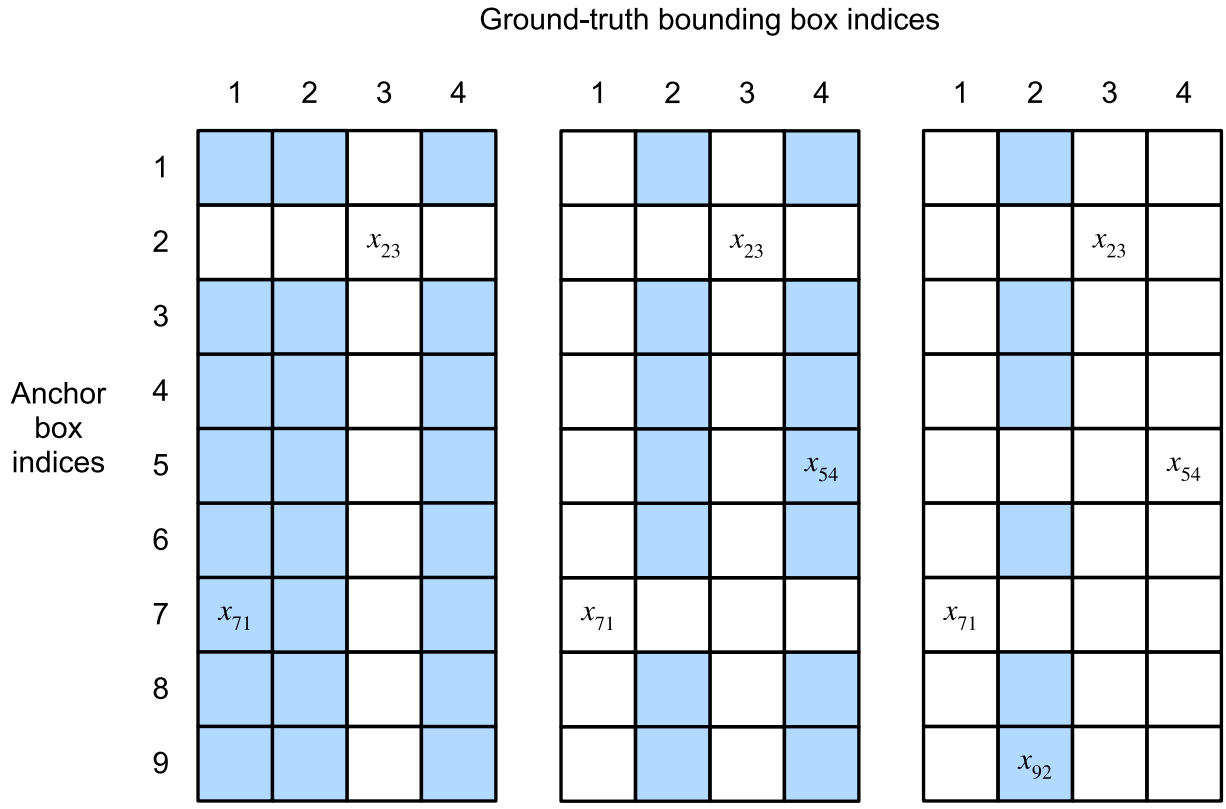
As shown in **Figure** (left) below, assuming that the maximum value in the matrix $\mathbf{X}$ is x_{23}, we will assign GT BB B_3 to AB A_2.

Then, we discard all the elements in row 2 and column 3 of the matrix, find the largest element x_{71} of the remaining shaded area, and assign GT BB B_1 to AB A_7.

Then, as shown in **Figure** (middle), discard all the elements in row 7 and column 1 of the matrix, find the largest element x_{54} of the remaining shaded area, and assign GT BB B_4 to AB A_5.

Finally, as shown in **Figure** (right), discard all the elements in row 5 and column 4 of the matrix, find the largest element x_{92} of the remaining shaded area, and assign GT BB B_2 to AB A_9.

After that, we only need to traverse the remaining ABs of A_1, A_3, A_4, A_6, A_8 and determine whether to assign GT BBs to the remaining ABs according to the threshold.



Match GT BBs to ABs

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
#@save
def match_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """Assign ground-truth bounding boxes to anchor boxes similar to them."""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # Element `x_ij` in the `i^th` row and `j^th` column is the IoU
    # of the anchor box `anc_i` to the ground-truth bounding box `box_j`
    jaccard = box_iou(anchors, ground_truth)
    # Initialize the tensor to hold assigned ground truth bbox for each anchor
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype=torch.long,
                                  device=device)
    # Assign ground truth bounding box according to the threshold
    max_ious, indices = torch.max(jaccard, dim=1)
    anc_i = torch.nonzero(max_ious >= 0.5).reshape(-1)
    box_j = indices[max_ious >= 0.5]
    anchors_bbox_map[anc_i] = box_j
    # Find the largest iou for each bbox
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
```

```
    return anchors_bbox_map
```

Now we can **label** the **categories** and **offsets** of the ABs.

If an AB A is assigned to GT BB B:

- the **category** of the AA A is set to the category of GT BB B,
- the **offset** of the AB A is set according to the relative position of the central coordinates of B and A and the relative sizes of the two boxes.

Because the **positions** and **sizes** of various boxes in the dataset **may vary**, these *relative* positions and *relative* sizes usually require some **special transformations** to make the offset **distribution** more **uniform** and **easier** to **fit**.

Assume the center coordinates of AB A and its assigned GT BB B are $(x_a, y_a), (x_b, y_b)$, the widths of A and B are w_a, w_b, and their heights are h_a, h_b, respectively.

In this case, a common technique is to label the offset of A as

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right)$$

The default values of the constant are

$$\mu_x = \mu_y = \mu_w = \mu_h = 0,$$
$$\sigma_x = \sigma_y = 0.1,$$
$$\sigma_w = \sigma_h = 0.2.$$

This transformation is implemented below in the `offset_boxes` function.

If an AB is **not assigned to** a GT BB, we only need to set the **category** of the AB to **background** and ...:

- ABs whose category is background are often referred to as **negative ABs**,
- and the rest are referred to as **positive ABs**.

Special Transformations of Relative Positions nad Sizes

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    c_anc = box_corner_to_center(anchors)
    c_assigned_bb = box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * torch.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = torch.cat([offset_xy, offset_wh], axis=1)
    return offset
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
#@save
```

"@save" is not an allowed annotation - allowed values include [@param, @title,

```
def multibox_target(anchors, labels):
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = match_anchor_to_bbox(label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze(-1)).repeat(
            1, 4)
        # Initialize class_labels and assigned bbox coordinates with zeros
        class_labels = torch.zeros(num_anchors, dtype=torch.long,
                                   device=device)
        assigned_bb = torch.zeros((num_anchors, 4), dtype=torch.float32,
                                  device=device)
        # Assign class labels to the anchor boxes using matched gt bbox labels
        # If no gt bbox is assigned to an anchor box, then let the
        # class_labels and assigned_bb remain zero, i.e the background class
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # offset transformations
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

@markdown].

Example

Let's define GT BBs for the cat and dog in the example image, where

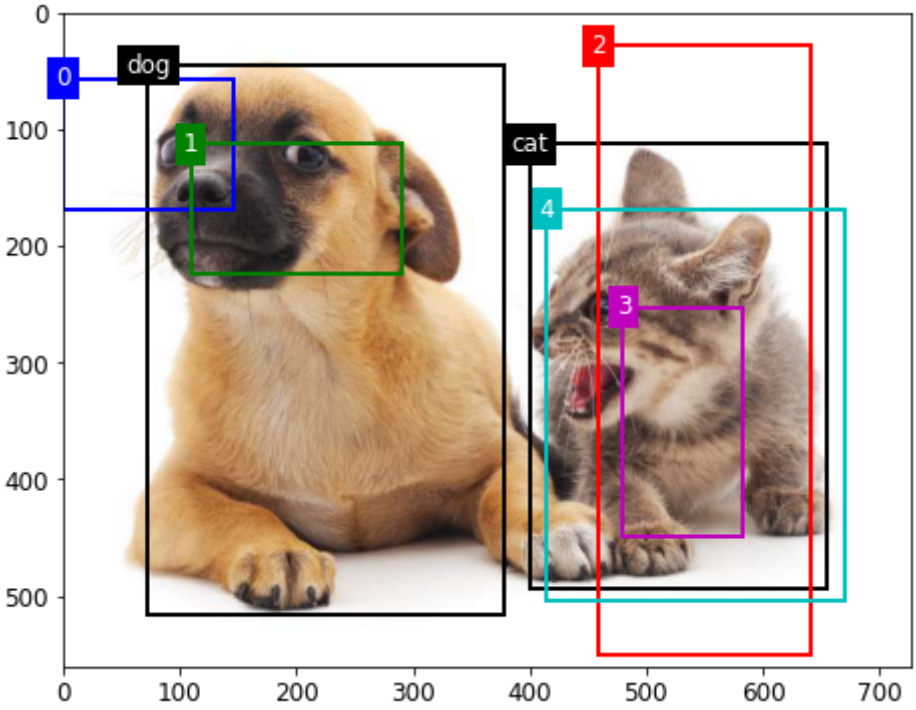
- the first element is **category** (0 for dog, 1 for cat),
- and the remaining four elements are the x, y axis **coordinates at top-left** corner and x, y axis **coordinates at lower-right** corner (the value range is between 0 and 1).

Let's construct 5 ABs to be labeled by the coordinates of the upper-left corner and the lower-right corner, which are recorded as $A_0, \ldots, A_4$, respectively (the index in the program starts from 0).

Let's draw the positions of these ABs and the GT BBs in the image.

```
ground_truth = torch.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                             [1, 0.55, 0.2, 0.9, 0.88]])
anchors = torch.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                        [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                        [0.57, 0.3, 0.92, 0.9]])

plt.figure(figsize=(8, 6))
```

```
plt.rcParams.update({'font.size': 12})
fig = plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



Let's label categories and offsets for ABs by using the `multibox_target` function.

This function sets the background category to 0 and increments the integer index of the target category from zero by 1 (1 for dog and 2 for cat).

Let's add example dimensions to the ABs and GT BBs and construct random predicted results with a shape of (batch size, number of categories including background, number of ABs) by using the `unsqueeze` function.

```
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

There are 3 items in the returned result, all of which are in the tensor format.

The 3rd item is represented by the category labeled for the AB.

```
labels[2]
```

```
    tensor([[0, 1, 2, 0, 2]])
```

Let's analyze these labelled categories based on positions of anchor boxes and ground-truth bounding boxes in the image.

- First, in all "AB--GT BB" pairs, the IoU of AA A_4 to the GT BB of the cat is the largest, so the category of AB A_4 is labeled as cat.
- Without considering AB A_4 or the GT BB of the cat, in the remaining "AB--GT BB" pairs, the pair with the largest IoU is AB A_1 and the GT BB of the dog, so the category of AB A_1 is labeled as dog.
- Next, traverse the remaining 3 unlabeled ABs:
    - the category of the GT BB with the largest IoU with AB A_0 is dog, but the IoU is smaller than the threshold (the default is 0.5), so the category is labeled as **background**;
    - the category of the GT BB with the largest IoU with AB A_2 is cat and the IoU is greater than the threshold, so the category is labeled as **cat**;
    - the category of the GT BB with the largest IoU with AB A_3 is cat, but the IoU is smaller than the threshold, so the category is labeled as **background**.

The second item of the return value is a **mask variable**, with the shape of (batch size, four times the number of anchor boxes).

The elements in the mask variable correspond 1-to-1 with the 4 offset values of each AB.

Because we do not care about background detection, offsets of the negative class should not affect the target function.

By multiplying by element, the 0 in the mask variable can filter out negative class offsets before calculating target function.

```
labels[1]
```

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1.,
1.,
         1., 1.]])
```

The first item returned is the 4 offset values labeled for each AB, with the offsets of negative class ABs labeled as 0.

```
labels[0]
```

```
tensor([[-0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
          2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
          4.17e-06,  6.26e-01]])
```

Bounding Boxes for Prediction

During model prediction phase, we:

- generate multiple ABs for the image,
- then predict categories and offsets for these ABs one by one,
- then, obtain prediction BBs (PBBs) based on ABs and their predicted offsets.

Below we implement function `offset_inverse` which takes in anchors and offset predictions as inputs and applies inverse offset transformations to return the PBB coordinates.

```
#@save
def offset_inverse(anchors, offset_preds):
    c_anc = box_corner_to_center(anchors)
    c_pred_bb_xy = (offset_preds[:, :2] * c_anc[:, 2:] / 10) + c_anc[:, :2]
    c_pred_bb_wh = torch.exp(offset_preds[:, 2:] / 5) * c_anc[:, 2:]
    c_pred_bb = torch.cat((c_pred_bb_xy, c_pred_bb_wh), axis=1)
    predicted_bb = box_center_to_corner(c_pred_bb)
    return predicted_bb
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

When there are many ABs, many similar PBBs may be output for the same target.

To simplify the results, we can remove similar prediction BBs by **non-maximum suppression (NMS)**:

- for a PBB B, the model calculates the predicted probability for each category - assume the **largest predicted probability** is p, the category corresponding to this probability is the predicted category of B (we also refer to p as the **confidence level** of PBB B),
- on the same image, we sort the PBBs with predicted categories other than background by confidence level from high to low, and obtain the list L,
- select the PBB B_1 with highest confidence level from L as a baseline and remove all non-benchmark PBBs with an IoU with B_1 greater than a certain **threshold** from L,

NOTE: The **threshold** here is a preset **hyperparameter**.

- at this point, L retains the PBB with the highest confidence level and removes other PBBs similar to it,
- select the PBB B_2 with the 2nd highest confidence level from L as a baseline, and remove all non-benchmark PBBs with an IoU with B_2 greater than a certain threshold from L,
- repeat this process until all PBBs in L have been used as a baseline,
- at this time, the IoU of any pair of PBBs in L is less than the threshold,
- finally, output all PBBs in the list L.

```
#@save
def nms(boxes, scores, iou_threshold):
    # sorting scores by the descending order and return their indices
    B = torch.argsort(scores, dim=-1, descending=True)
    keep = []  # boxes indices that will be kept
    while B.numel() > 0:
        i = B[0]
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),
                      boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return torch.tensor(keep, device=boxes.device)

#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                       pos_threshold=0.00999999978):
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = torch.max(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, 0.5)
        # Find all non_keep indices and set the class_id to background
        all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
        combined = torch.cat((keep, all_idx))
        uniques, counts = combined.unique(return_counts=True)
        non_keep = uniques[counts == 1]
        all_id_sorted = torch.cat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted]
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        # threshold to be a positive prediction
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = torch.cat(
            (class_id.unsqueeze(1), conf.unsqueeze(1), predicted_bb), dim=1)
        out.append(pred_info)
    return torch.stack(out)
```

Example

Let's construct 4 anchor boxes.

For the sake of simplicity, we assume that predicted offsets are all 0. This means that the PBBs are ABs.

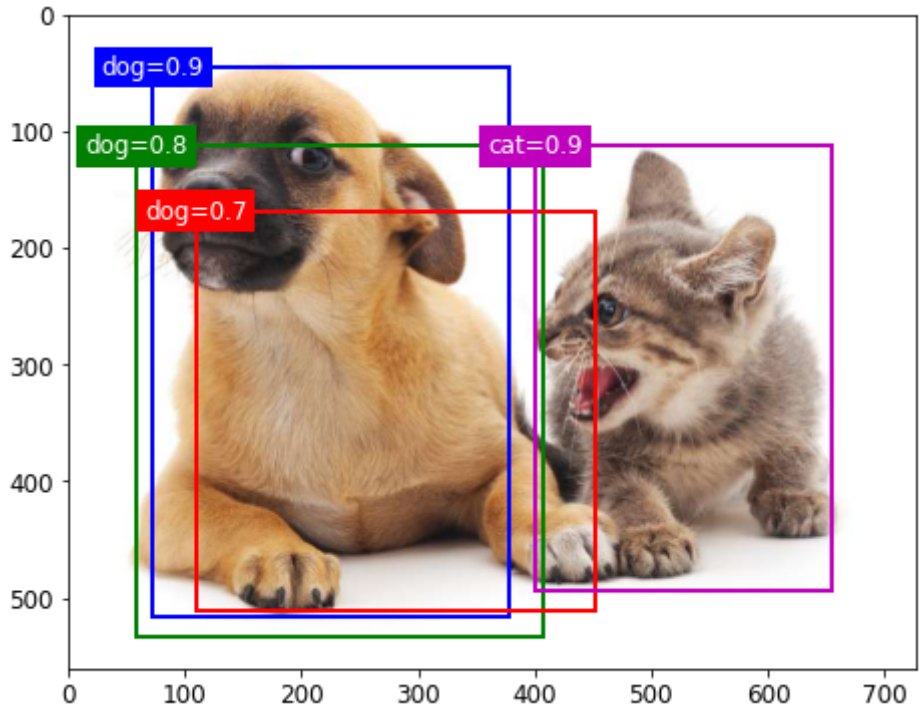
Finally, we construct a predicted probability for each category.

```
anchors = torch.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                        [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = torch.tensor([0] * anchors.numel())
cls_probs = torch.tensor([[0] * 4,  # Predicted probability for background
                          [0.9, 0.8, 0.7,
                           0.1],  # Predicted probability for dog
                          [0.1, 0.2, 0.3,
```

```
                                0.9]])  # Predicted probability for cat
```

Print PPBs and their confidence levels on the image.

```
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 12})
fig = plt.imshow(image)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



Let's use the `multibox_detection` function to perform NMS and set the threshold to 0.5. This adds an example dimension to the tensor input.

We can see that the shape of the returned result is (batch size, number of anchor boxes, 6).

The 6 elements of each row represent the output information for the same PBB.

- The first element is the predicted category index, which starts from 0 (0 is dog, 1 is cat). The value -1 indicates background or removal in NMS.
- The second element is the confidence level of PBB.
- The remaining four elements are the x, y axis coordinates of the upper-left corner and the x, y axis coordinates of the lower-right corner of the PBB (the value range is between 0 and 1).

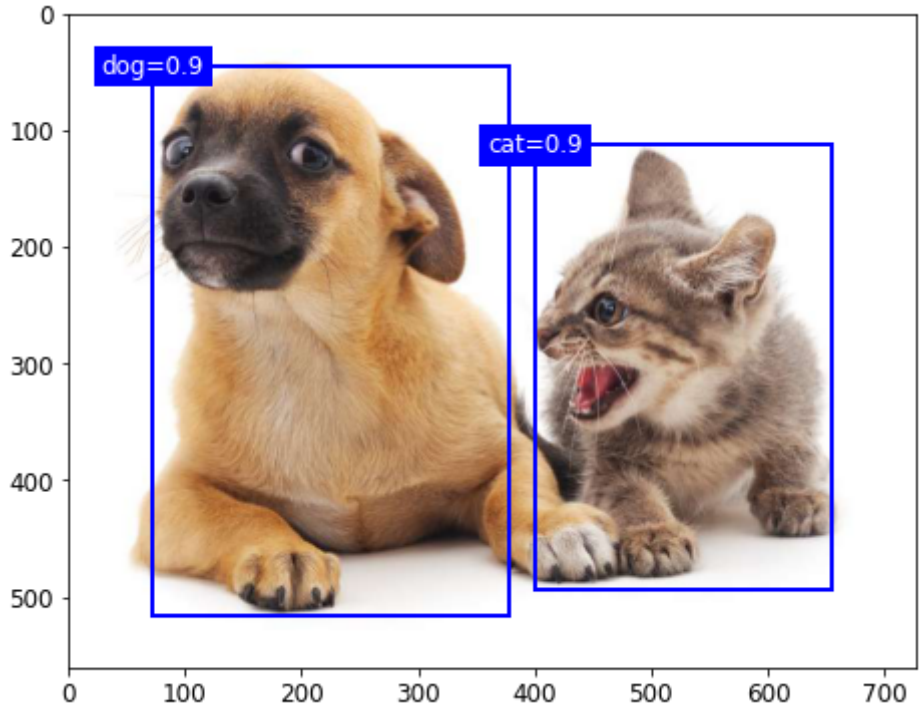
```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0), nms_threshold=0.5)
output
```

```
    tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
```

```
[ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
[-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
[-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

We remove the PBB of category -1 and visualize the results retained by NMS.

```
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 12})
fig = plt.imshow(image)
for i in output[0].detach().numpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)
```



In practice, we can remove PBBs with lower confidence levels before performing NMS, thereby reducing the amount of computation for NMS. We can also filter the output of NMS, for example, by only retaining results with higher confidence levels as the final output.

Summary

- We generate multiple ABs with different sizes and aspect ratios, centered on each pixel.
- IoU, also called Jaccard index, measures the similarity of two BBs. It is the ratio of the intersecting area to the union area of 2 BBs.
- In the training set, we mark two types of labels for each AB: one is the category of the target contained in the AB and the other is the offset of the GT BB relative to the AB.
- When predicting, we can use non-maximum suppression (NMS) to remove similar prediction BBs, thereby simplifying the results.

Section 13.5.Multiscale Object Detection

Above we generated multiple ABs centered on each pixel of the input image. These ABs are used to sample different regions of the input image.

However, if ABs are generated centered on each pixel of the image, soon there will be too many ABs for us to compute. For example, we assume that the input image has a height and a width of 561 and 728 pixels respectively.

If 5 different shapes of ABs are generated centered on each pixel, over **2 million** ABs ($561 \times 728 \times 5$) **need to be predicted and labeled** on the image!

BUT ... it is not difficult to reduce the number of ABs.

- An easy way is to apply uniform sampling on a small portion of pixels from the input image and generate ABs centered on the sampled pixels.
- In addition, we can generate ABs of varied numbers and sizes on multiple scales.

NOTE: Smaller objects are more likely to be positioned on the image than larger ones.

Here, we will use a simple example: Objects with shapes of $1 \times 1, 1 \times 2$, and 2×2 may have 4, 2, and 1 possible position(s) on an image with the shape 2×2. Therefore, when using smaller ABs to detect smaller objects, we can sample more regions; when using larger ABs to detect larger objects, we can sample fewer regions.

To demonstrate how to generate ABs on multiple scales, let us read an image with a height and width of 561×728 pixels.

```
%matplotlib inline
import torch
#from d2l import torch as d2l

img = plt.imread('./catdog.jpg')
h, w = img.shape[0:2]
h, w
```

```
(561, 728)
```

Before (during the previus lectures on CNN), the 2D array **output** of the convolutional neural network (CNN) is called a **feature map**.

Let's determine the midpoints of ABs uniformly sampled on any image by defining the shape of the feature map.

The function `display_anchors` is defined below.

We are going to generate ABs `anchors` centered on each unit (pixel) on the feature map `fmap`.

Since the coordinates of axes x and y in ABs `anchors` have been divided by the width and height of the feature map `fmap`, values between 0 and 1 can be used to represent relative positions of ABs in the feature map.

Since the midpoints of ABs `anchors` overlap with all the units on feature map `fmap`, the relative spatial positions of the midpoints of the `anchors` on any image must have a uniform distribution.

Specifically, when the width and height of the feature map are set to `fmap_w` and `fmap_h` respectively, the function will conduct uniform sampling for `fmap_h` rows and `fmap_w` columns of pixels and use them as midpoints to generate ABs with size `s` (we assume that the length of list `s` is 1) and different aspect ratios (`ratios`).

```
def display_anchors(fmap_w, fmap_h, s):
    #set_figsize()
    # The values from the first two dimensions will not affect the output
    fmap = torch.zeros((1, 10, fmap_h, fmap_w))
    anchors = multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = torch.tensor((w, h, w, h))
    show_bboxes(plt.imshow(img).axes, anchors[0] * bbox_scale)
```

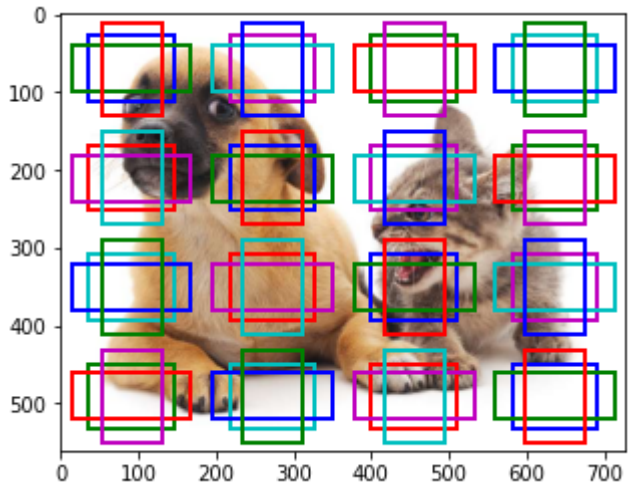
Let's focus on the detection of small objects.

In order to make it easier to distinguish upon display, the ABs with different midpoints here do not overlap.

Let's assume that the size of the ABs is 0.15 and the height and width of the feature map are 4.

We can see that the midpoints of ABs from the 4 rows and 4 columns on the image are uniformly distributed.

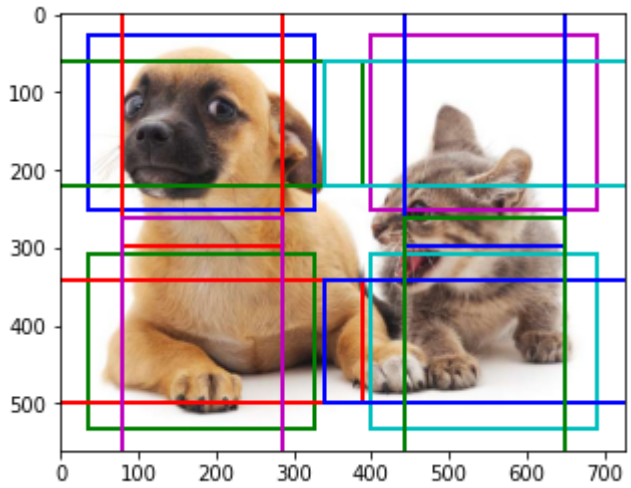
```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```



Let's reduce the height and width of the feature map by half and use a larger AB to detect larger objects.

When the size is set to 0.4, overlaps will occur between regions of some anchor boxes.

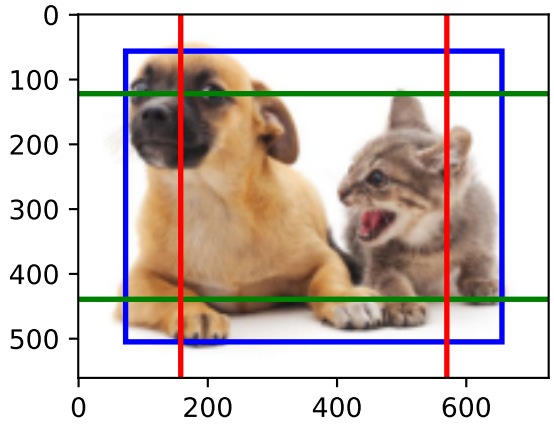
```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



Let's reduce the height and width of the feature map by half and increase the AB size to 0.8.

Now the midpoint of the AB is the center of the image.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



Since we have generated ABs of different sizes on multiple scales, we will use them to detect objects of various sizes at different scales.

Let's introduce a method based on convolutional neural networks (CNNs).

At a certain scale, suppose we generate $h \times w$ sets of ABs with different midpoints based on c_i feature maps with the shape $h \times w$ and the number of ABs in each set is a.

For example, for the first scale of the experiment, we generate 16 sets of ABs with different midpoints based on 10 (number of channels) feature maps with a shape of 4×4, and each set contains 3 ABs.

Next, each AB is labeled with a category and offset based on the classification and position of the GT BB.

At the current scale, the object detection model needs to predict the category and offset of $h \times w$ sets of ABs with different midpoints based on the input image.

We assume that the c_i feature maps are the intermediate output of the CNN based on the input image. Since each feature map has $h \times w$ different spatial positions, the same position will have c_i units. According to the definition of receptive field (in the previous lectures on CNN), the c_i units of the feature map at the same spatial position have the same receptive field on the input image. Thus, they represent the information of the input image in this same receptive field. Therefore, we can transform the c_i units of the feature map at the same spatial position into the categories and offsets of the a ABs generated using that position as a midpoint. It is not hard to see that, in essence, we use the information of the input image in a certain receptive field to predict the category and offset of the ABs close to the field on the input image.

When the feature maps of different layers have receptive fields of different sizes on the input image, they are used to detect objects of different sizes. For example, we can design a network to have a wider receptive field for each unit in the feature map that is closer to the output layer, to detect objects with larger sizes in the input image.

We will implement a multiscale object detection model in the following section.

Summary

- We can generate ABs with different numbers and sizes on multiple scales to detect objects of different sizes on multiple scales.
- The shape of the feature map can be used to determine the midpoint of the ABs that uniformly sample any image.
- We use the information for the input image from a certain receptive field to predict the category and offset of the ABs close to that field on the image.

Section 13.6.The Object Detection Dataset

In the object detection field ... **there are no small datasets** like MNIST or Fashion-MNIST.

In order to quickly test models, we are going to assemble a small dataset:

- **generate** 1000 banana images of different angles and sizes,
- then, **collect** a series of **background** images and **place** a banana image **at a random position** on each image.

Get Custom d2l-Dataset

The banana detection dataset with all the images and csv label files can be downloaded directly from the Internet.

```
%matplotlib inline
import os
import pandas as pd
import torch
import torchvision
from d2l import torch as d2l

#@save
d2l.DATA_HUB['banana-detection'] = (d2l.DATA_URL + 'banana-detection.zip',
                            '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

Read and Prepare the Custom Dataset

We are going to read the object detection dataset in the `read_data_bananas` function.

The dataset includes a csv file for target class labels and GT BB coordinates in the `corner` format.

Let's define:

- `BananasDataset` to create the `Dataset` instance,
- `load_data_bananas` function to return the dataloaders.

There is no need to read the test dataset in random order.

```
#@save
def read_data_bananas(is_train=True):
    """Read the bananas dataset images and labels."""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir,
                             'bananas_train' if is_train else 'bananas_val',
                             'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(
            torchvision.io.read_image(
                os.path.join(data_dir,
                             'bananas_train' if is_train else 'bananas_val',
                             'images', f'{img_name}')))
        # Since all images have same object class i.e. category '0',
        # the `label` column corresponds to the only object i.e. banana
        # The target is as follows : (`label`, `xmin`, `ymin`, `xmax`, `ymax`)
        targets.append(list(target))
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
    return images, torch.tensor(targets).unsqueeze(1) / 256

#@save
class BananasDataset(torch.utils.data.Dataset):
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (
            f' training examples' if is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])

    def __len__(self):
        return len(self.features)

#@save
def load_data_bananas(batch_size):
    """Load the bananas dataset."""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                             batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return (train_iter, val_iter)
```

Let's read a minibatch and print the shape of the image and label.

The shape of the image is the same as in the previous experiment (batch size, number of channels, height, width).

The shape of the label is (batch size, m, 5), where m is equal to the maximum number of BBs contained in a single image in the dataset.

Although computation for the minibatch is very efficient, it requires each image to contain the same number of BBs so that they can be placed in the same batch.

Since each image may have a different number of BBs, we can add illegal BBs to images that have less than m BBs until each image contains m BBs. Thus, we can read a minibatch of images each time.

The label of each BB in the image is represented by a tensor of length 5:

- the first element in the tensor is the category of the object contained in the BB. When the value is -1, the BB is an illegal BB for filling purpose,
- the remaining 4 elements of the array represent the x, y axis coordinates of the upper-left corner of the bounding box and the x, y axis coordinates of the lower-right corner of the BB (the value range is between 0 and 1). The banana dataset here has only one BB per image, so $m = 1$.

```
%%time

batch_size, edge_size = 32, 256
```

```
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

```
Downloading ../data/banana-detection.zip from http://d2l-data.s3-accelerate.a
read 1000 training examples
read 100 validation examples
CPU times: user 4.12 s, sys: 818 ms, total: 4.94 s
Wall time: 7.41 s
```

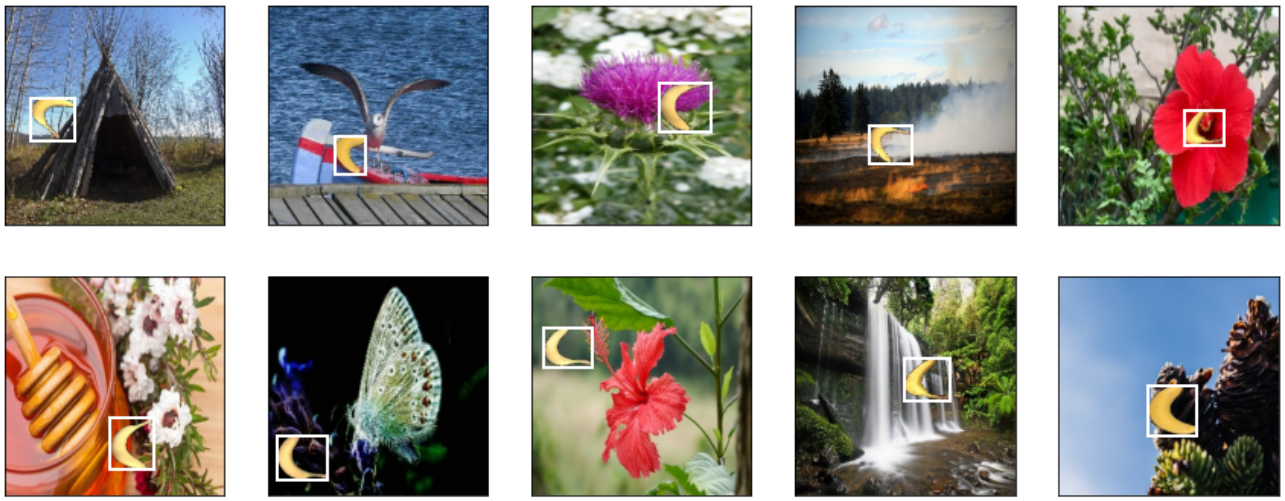
Demonstration

We have 10 images with BBs on them.

We can see that the angle, size, and position of banana are different in each image.

Of course, this is a simple artificial dataset. In actual practice, the data are usually much more complicated.

```
imgs = (batch[0][0:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=3)
for ax, label in zip(axes, batch[1][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



Summary

- The banana detection dataset we synthesized can be used to test object detection models.
- The data reading for object detection is similar to that for image classification. However, after we introduce BBs, the label shape and image augmentation (e.g., random cropping) are changed.

Section 13.7.Single Shot Multibox Detection (SSD)

In the previous few sections, we have introduced:

- bounding boxes,
- anchor boxes,
- multiscale object detection,
- custom datasets.

Now, we will use this background knowledge to construct another object detection model, namely, **single shot multibox detection (SSD)** proposed in the following famous paper:

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single Shot multibox Detector. *European conference on computer vision* (pp. 21–37).

This quick and easy model is already widely used. Some of the design concepts and implementation details of this model are also applicable to other object detection models.

Model

This **figure** shows the design of an SSD model.

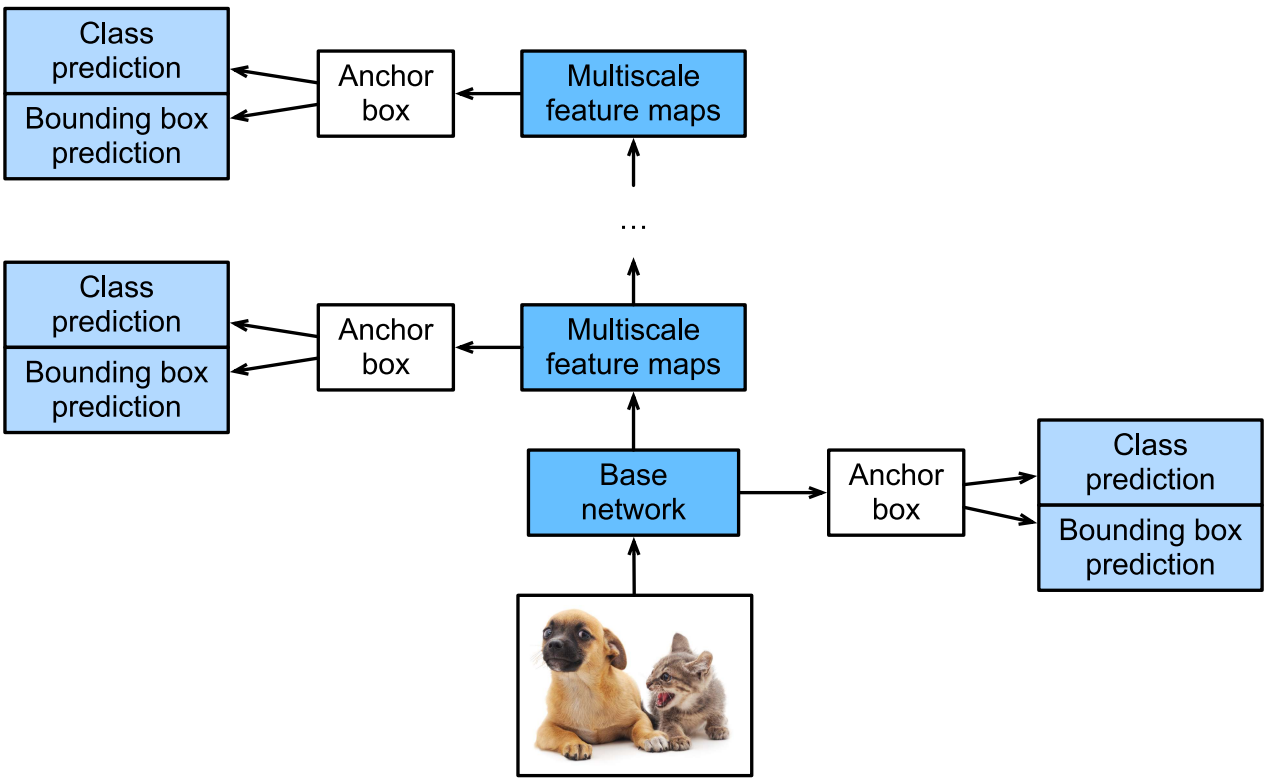


Figure. The SSD is composed of a base network block and several multiscale feature blocks connected in a series.

The model's main components are:

- a **base network block**,
- several **multiscale feature blocks** connected in a series.

The **base network block** is used to extract features of original images, and it generally takes the form of a deep CNN.

NOTE: The original paper on SSDs chooses to place a truncated **VGG** before the classification layer, but this is now commonly replaced by **ResNet**.

Let's design the base network so that it outputs larger heights and widths. In this way, more ABs are generated based on this feature map, allowing us to detect smaller objects. Next, each multiscale feature block reduces the height and width of the feature map provided by the previous layer (for example, it may reduce the sizes by half). The blocks then use each element in the feature map to expand the receptive field on the input image. In this way, the closer a multiscale feature block is to the top of SSD (see Figure) the smaller its output feature map, and the fewer the anchor boxes that are generated based on the feature map. In addition, the closer a feature block is to the top, the larger the receptive field of each element in the feature map and the better suited it is to detect larger objects. As the SSD generates different numbers of ABs of different sizes based on the base network block and each multiscale feature block and then predicts the categories and offsets (i.e., PBBs) of the ABs in order to detect objects of different sizes, SSD is a multiscale object detection model.

Next, let's describe the implementation of the modules in SSD (see Figure).

First, we need to discuss the implementation of category prediction and bounding box prediction layers.

Category Prediction Layer

Set the number of object categories to q. In this case, the number of AB categories is $q+1$, with 0 indicating an AB that only contains background.

For a certain scale, set the height and width of the feature map to h and w, respectively. If we use each element as the center to generate a ABs, we need to classify a total of hwa ABs.

If we use a fully connected layer (FCN) for the output, this will likely result in an excessive number of model parameters. Recall how we used convolutional layer channels to output category predictions before for NiN. SSD uses the same method to reduce the model complexity.

Specifically, the category prediction layer uses a convolutional layer that maintains the input height and width. Thus, the output and input have a one-to-one correspondence to the spatial coordinates along the width and height of the feature map. Assuming that the output and input have the same spatial coordinates (x, y), the channel for the coordinates (x, y) on the output feature map contains the category predictions for all ABs generated using the input feature map coordinates (x, y) as the center. Therefore, there are $a(q+1)$ output channels, with the output

channels indexed as $i(q+1)+j$ $(0 \leq j \leq q)$ representing the predictions of the category index j for the AB index i.

Now, we will define a category prediction layer of this type. After we specify the parameters a and q, it uses a 3×3 convolutional layer with a padding of 1. The heights and widths of the input and output of this convolutional layer remain unchanged.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

Bounding Box Prediction Layer

The design of the bounding box prediction layer is similar to that of the category prediction layer.

The only difference is that, here, we need to predict 4 offsets for each AB, rather than $q+1$ categories.

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

Concatenating Predictions for Multiple Scales

As we mentioned, SSD uses feature maps based on multiple scales to generate ABs and predict their categories and offsets. Because the shapes and number of ABs centered on the same element differ for the feature maps of different scales, the prediction outputs at different scales may have different shapes.

In the following example, we use the same batch of data to construct feature maps of two different scales, `Y1` and `Y2`. Here, `Y2` has half the height and half the width of `Y1`. Using category prediction as an example, we assume that each element in the `Y1` and `Y2` feature maps generates five (Y1) or three (Y2) ABs. When there are 10 object categories, the number of category prediction output channels is either $5 \times (10+1) = 55$ or $3 \times (10+1) = 33$. The format of the prediction output is (batch size, number of channels, height, width). As you can see, except for the batch size, the sizes of the other dimensions are different. Therefore, we must transform them into a consistent format and concatenate the predictions of the multiple scales to facilitate subsequent computation.

```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
(Y1.shape, Y2.shape)
```

```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

The channel dimension contains the predictions for all ABs with the same center. We first move the channel dimension to the final dimension. Because the batch size is the same for all scales, we can convert the prediction results to binary format (batch size, height $\times$ width $\times$ number of channels) to facilitate subsequent concatenation on the 1^{st} dimension.

```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

Thus, regardless of the different shapes of `Y1` and `Y2`, we can still concatenate the prediction results for the two different scales of the same batch.

```
concat_preds([Y1, Y2]).shape
```

```
torch.Size([2, 25300])
```

Height and Width Downsample Block

For multiscale object detection, we define the following `down_sample_blk` block, which reduces the height and width by 50%.

This block consists of two 3×3 convolutional layers with a padding of 1 and a 2×2 maximum pooling layer with a stride of 2 connected in a series.

As we know, 3×3 convolutional layers with a padding of 1 do not change the shape of feature maps.

However, the subsequent pooling layer directly reduces the size of the feature map by half.

Because $1 \times 2 + (3 - 1) + (3 - 1) = 6$, each element in the output feature map has a receptive field on the input feature map of the shape 6×6.

As you can see, the height and width downsample block enlarges the receptive field of each element in the output feature map.

```
def down_sample_blk(in_channels, out_channels):
```

```
    blk = []
    for _ in range(2):
        blk.append(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

By testing forward computation in the height and width downsample block, we can see that it changes the number of input channels and halves the height and width.

```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

```
torch.Size([2, 10, 10, 10])
```

Base Network Block

The base network block is used to extract features from original images.

To simplify the computation, we will construct a small base network.

This network consists of three height and width downsample blocks connected in a series, so it doubles the number of channels at each step.

When we input an original image with the shape 256×256, the base network block outputs a feature map with the shape 32×32.

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i + 1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

```
torch.Size([2, 64, 32, 32])
```

The Complete Model

The SSD model contains a total of 5 modules.

Each module outputs a feature map used to generate ABs and predict the categories and offsets of these ABs.

The first module is the base network block, modules two to four are height and width downsample blocks, and the fifth module is a global maximum pooling layer that reduces the height and width to 1.

Therefore, modules two to five are all multiscale feature blocks (see on Figure above).

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1, 1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

Now, we will define the forward computation process for each module.

In contrast to the previously-described CNNs, this module not only returns feature map `Y` output by convolutional computation, but also the anchor boxes of the current scale generated from `Y` and their predicted categories and offsets.

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

As we mentioned, the closer a multiscale feature block is to the top in SSD (see on Figure above), the larger the objects it detects and the larger the ABs it must generate.

Here, we first divide the interval from 0.2 to 1.05 into 5 equal parts to determine the sizes of smaller ABs at different scales: 0.2, 0.37, 0.54, etc.

Then, according to $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$, and similar formulas, we determine the sizes of larger ABs at the different scales.

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

Now, we can define the complete model, `TinySSD`.

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
```

```
        for i in range(5):
            # The assignment statement is self.blk_i = get_blk(i)
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(
                self, f'cls_{i}',
                cls_predictor(idx_to_in_channels[i], num_anchors,
                              num_classes))
            setattr(self, f'bbox_{i}',
                    bbox_predictor(idx_to_in_channels[i], num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # getattr(self, 'blk_%d' % i) accesses self.blk_i
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        # In the reshape function, 0 indicates that the batch size remains
        # unchanged
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(cls_preds.shape[0], -1,
                                      self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

We now create an SSD model instance and use it to perform forward computation on image minibatch `X`, which has a height and width of 256 pixels.

As we verified previously, the first module outputs a feature map with the shape 32×32.

Because modules two to four are height and width downsample blocks, module five is a global pooling layer, and each element in the feature map is used as the center for 4 anchor boxes, a total of $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ ABs are generated for each image at the five scales.

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
```

Training

Now, we will explain, step by step, how to train the SSD model for object detection.

Data Reading and Initialization

We read the banana detection dataset we created in the previous section.

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

There is 1 category in the banana detection dataset. After defining the module, we need to initialize the model parameters and define the optimization algorithm.

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

Defining Loss and Evaluation Functions

Object detection is subject to two types of losses:

- the **AB category loss** - we can simply reuse the cross-entropy loss function we used in image classification,
- the positive **AB offset loss** - offset prediction is a normalization problem, ... but, here, we do not use the squared loss introduced previously, ... rather, we use the L_1 norm loss, which is the **absolute value of the difference between the predicted value and the ground-truth value**.

The mask variable `bbox_masks` removes negative ABs and padding ABs from the loss calculation.

Finally, we **add** the **AB category loss** and **AB offset losses** to find the **final loss function** for the model.

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

We can use the **accuracy** rate to evaluate the classification results.

As we use the L_1 norm loss, we will use the average absolute error to evaluate the bounding box prediction results.

```
def cls_eval(cls_preds, cls_labels):
    # Because the category prediction results are placed in the final
    # dimension, argmax must specify this dimension
    return float(
        (cls_preds.argmax(dim=-1).type(cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

Training the Model

During model training, we must:

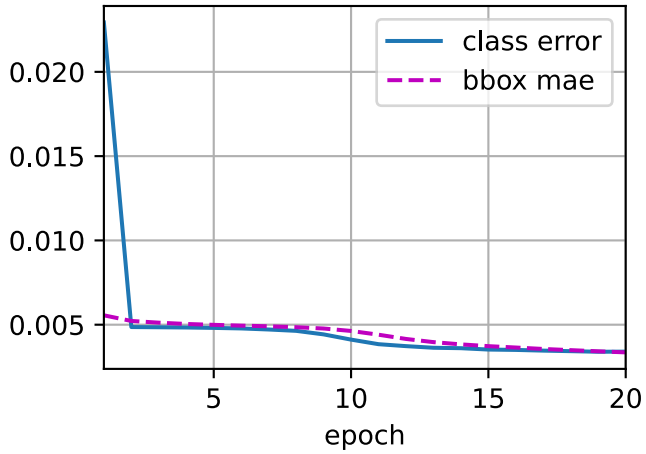
- **generate** multiscale ABs (`anchors`) in the model's forward computation process,
- **predict** the **category** (`cls_preds`) and **offset** (`bbox_preds`) for each AB,
- **label** the category (`cls_labels`) and offset (`bbox_labels`) of each generated AB based on the label information `Y`,
- finally, **calculate the loss function** using the predicted and labeled category and offset values.

NOTE: To simplify the code, we do not evaluate the training dataset here.

```
%%time

num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # accuracy_sum, mae_sum, num_examples, num_labels
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # Generate multiscale anchor boxes and predict the category and
        # offset of each
        anchors, cls_preds, bbox_preds = net(X)
        # Label the category and offset of each anchor box
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
        # Calculate the loss function using the predicted and labeled
        # category and offset values
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
```

```
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.40e-03, bbox mae 3.36e-03
3141.7 examples/sec on cuda:0
CPU times: user 2min 31s, sys: 1min 15s, total: 3min 47s
Wall time: 3min 53s
```



Prediction

In the prediction stage, we want to detect all objects of interest in the image, and below let's:

- read the test image and transform its size,
- convert it to the four-dimensional format required by the convolutional layer.

```
X = torchvision.io.read_image('./banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```

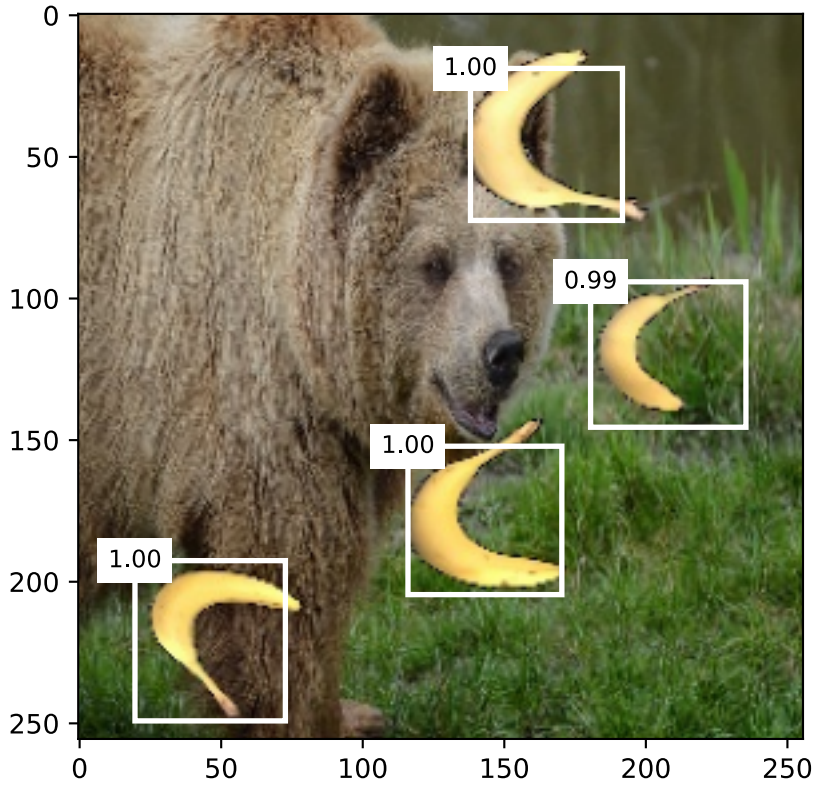
Using the `multibox_detection` function, we predict the BBs based on the ABs and their predicted offsets.

Then, we use non-maximum suppression to remove similar BBs.

```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

Finally, we take all the BBs with a confidence level of at least 0.9 and display them as the final output.

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```



Summary

- SSD is a multiscale object detection model. This model generates different numbers of ABs of different sizes based on the base network block and each multiscale feature block and predicts the categories and offsets of the ABs to detect objects of different sizes.
- During SSD model training, the loss function is calculated using the predicted and labeled category and offset values.

Section 13.8.Region-based CNNs (R-CNNs)

Region-based convolutional neural networks or **regions with CNN features** (R-CNNs) is a pioneering approach that applies DL models to object detection that was presented in the famous paper:

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014).Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).

In this section, we will discuss R-CNNs and a series of improvements made to them:

- **Fast R-CNN** -> Girshick, R. (2015). Fast R-CNN. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448),
- **Faster R-CNN** -> Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99),
- **Mask R-CNN** -> He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask R-CNN. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).

Due to space limitations, we will confine our discussion to the **designs of these models only**.

R-CNNs

R-CNN models:

- select several proposed regions from an image (for example, ABs are one type of selection method) and then label their categories and BBs (e.g., offsets),
- use a CNN to perform forward computation to extract features from each proposed area,
- use the features of each proposed region to predict their categories and BBs.

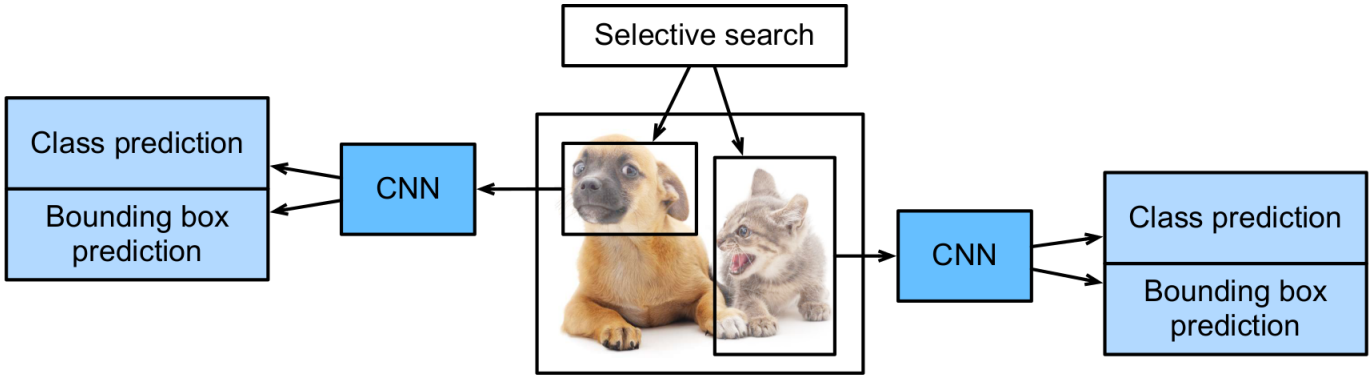


Figure. R-CNN model.

Specifically, R-CNNs are composed of four main parts:

1. **Selective search** is performed on the input image to select multiple high-quality proposed regions (see Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, 104(2),

154–171.). These proposed regions are generally selected on multiple scales and have different shapes and sizes. The category and GT BB of each proposed region is labeled.
2. A pre-trained CNN is selected and placed, in truncated form, before the output layer. It transforms each proposed region into the input dimensions required by the network and uses forward computation to output the features extracted from the proposed regions.
3. The features and labeled category of each proposed region are combined as an example to train multiple support vector machines for object classification. Here, each support vector machine is used to determine whether an example belongs to a certain category.
4. The features and labeled bounding box of each proposed region are combined as an example to train a linear regression model for GT BB prediction.

Although R-CNN models use pre-trained CNNs to effectively extract image features, the main downside is the **slow speed**. As you can imagine, we can select thousands of proposed regions from a single image, requiring thousands of forward computations from the CNN to perform object detection. This massive computing load means that **R-CNNs are not widely used in actual applications**.

Fast R-CNN

The main performance bottleneck of an R-CNN model is the need to independently extract features for each proposed region.

As these regions have a high degree of overlap, independent feature extraction results in a high volume of repetitive computations.

Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole.

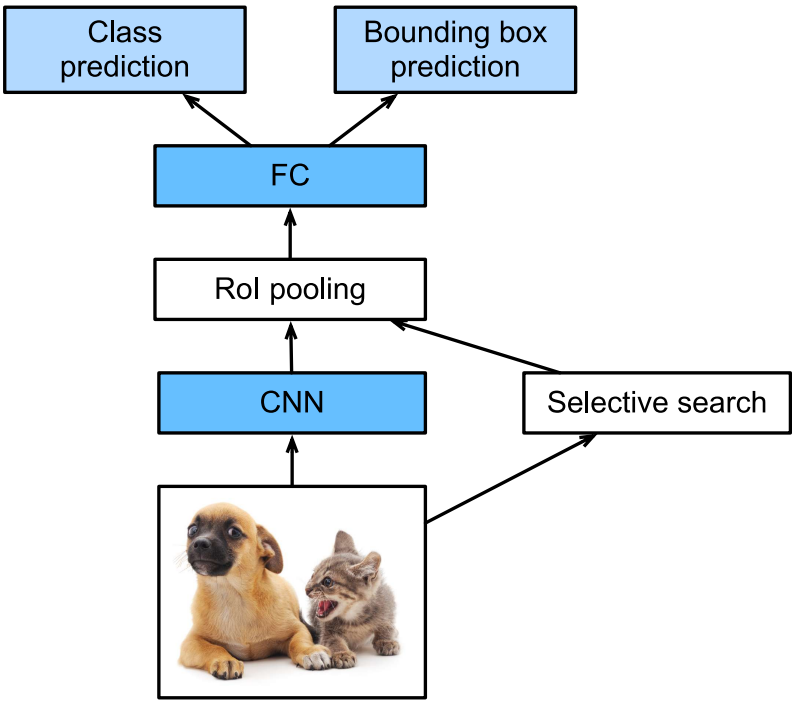


Figure. Fast R-CNN model.

The primary computation steps in Fast R-CNN model are described below:

1. Compared to an R-CNN model, a Fast R-CNN model uses the entire image as the CNN input for feature extraction, rather than each proposed region. Moreover, this network is generally trained to update the model parameters. As the input is an entire image, the CNN output shape is $1 \times c \times h_1 \times w_1$.
2. Assuming selective search generates n proposed regions, their different shapes indicate **regions of interests (RoIs)** of different shapes on the CNN output. Features of the same shapes must be extracted from these RoIs (here we assume that the height is h_2 and the width is w_2). Fast R-CNN introduces RoI pooling, which uses the CNN output and RoIs as input to output a concatenation of the features extracted from each proposed region with the shape $n \times c \times h_2 \times w_2$.
3. A fully connected layer is used to transform the output shape to $n \times d$, where d is determined by the model design.
4. During category prediction, the shape of the fully connected layer output is again transformed to $n \times q$ and we use softmax regression (q is the number of categories). During bounding box prediction, the shape of the fully connected layer output is again transformed to $n \times 4$. This means that we predict the category and bounding box for each proposed region.

The RoI pooling layer in Fast R-CNN is somewhat different from the pooling layers we have discussed before.

In a normal pooling layer, we set the pooling window, padding, and stride to control the output shape.

In an RoI pooling layer, we can directly specify the output shape of each region, such as specifying the height and width of each region as h_2, w_2.

Assuming that the height and width of the RoI window are h and w, this window is divided into a grid of sub-windows with the shape $h_2 \times w_2$. The size of each sub-window is about $(h/h_2) \times (w/w_2)$. The sub-window height and width must always be integers and the largest element is used as the output for a given sub-window. This allows the RoI pooling layer to extract features of the same shape from RoIs of different shapes.

In **Figure** below, we select an 3×3 region as an RoI of the 4×4 input. For this RoI, we use a 2×2 RoI pooling layer to obtain a single 2×2 output. When we divide the region into four sub-windows, they respectively contain the elements 0, 1, 4, and 5 (5 is the largest); 2 and 6 (6 is the largest); 8 and 9 (9 is the largest); and 10.

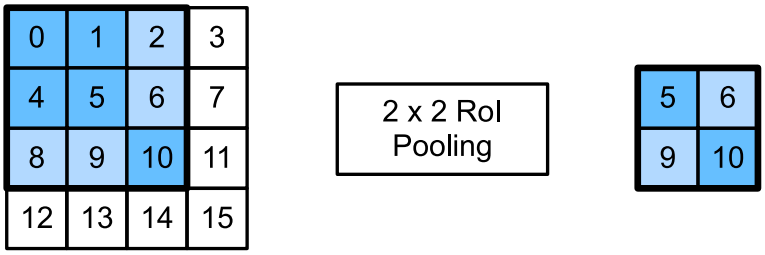


Figure. 2×2 RoI pooling layer

We use the `roi_pool` function from `torchvision` to demonstrate the RoI pooling layer computation.

Assume that the CNN extracts the feature `X` with both a height and width of 4 and only a single channel.

```
import torch
import torchvision

X = torch.arange(16.).reshape(1, 1, 4, 4)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]])
```

Assume that the height and width of the image are both 40 pixels and that selective search generates two proposed regions on the image.

Each region is expressed as five elements: the region's object category and the x, y coordinates of its upper-left and bottom-right corners.

```
rois = torch.Tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Because the height and width of `X` are $1/10$ of the height and width of the image, the coordinates of the two proposed regions are multiplied by 0.1 according to the `spatial_scale`, and then the RoIs are labeled on `X` as `X[:, :, 0:3, 0:3]` and `X[:, :, 1:4, 0:4]`, respectively.

Finally, we divide the two RoIs into a sub-window grid and extract features with a height and width of 2.

```
torchvision.ops.roi_pool(X, rois, output_size=(2, 2), spatial_scale=0.1)
```

```
tensor([[[[ 5.,  6.],
          [ 9., 10.]]],
```

```
[[[ 9., 11.],
  [13., 15.]]])
```

Faster R-CNN

In order to obtain precise object detection results, Fast R-CNN generally requires that many proposed regions be generated in selective search.

Faster R-CNN replaces **selective search** with a **region proposal network**. This reduces the number of proposed regions generated, while ensuring precise object detection.

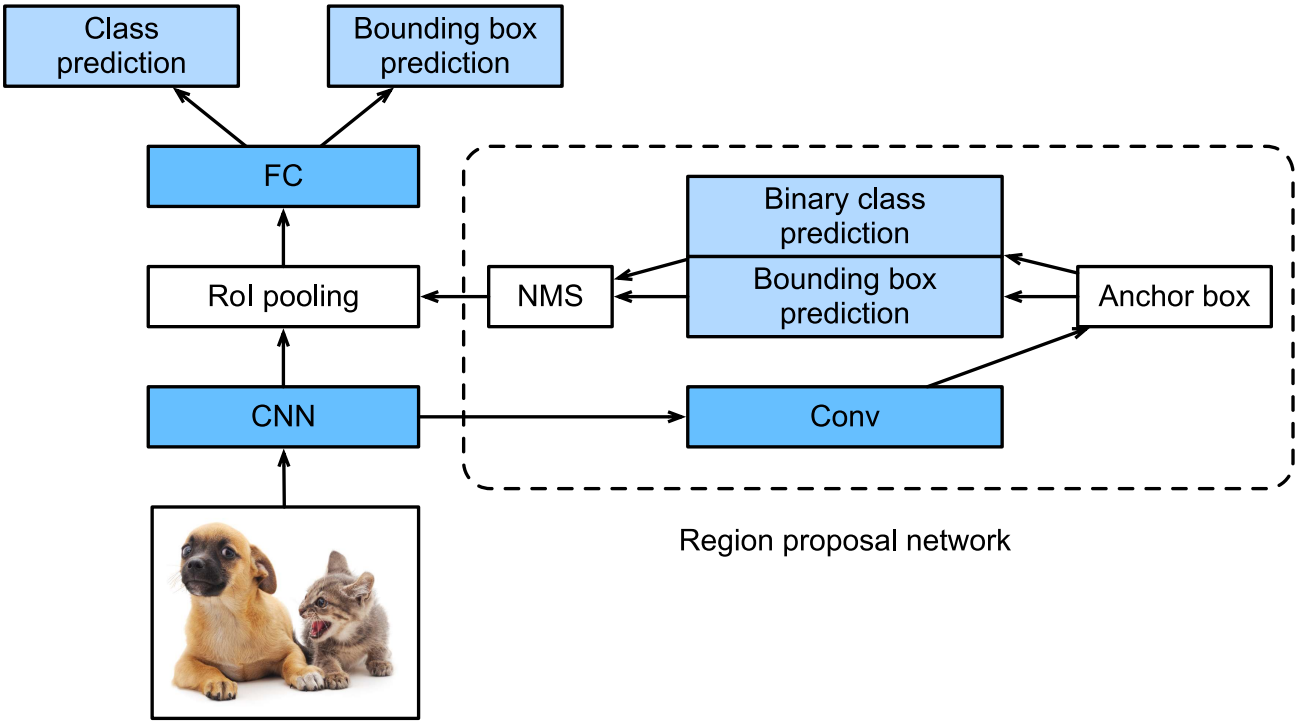


Figure. Faster R-CNN model.

Compared to Fast R-CNN, Faster R-CNN only changes the method for generating proposed regions from selective search to region proposal network.

The other parts of the model remain unchanged.

The detailed region proposal network computation process is described below:

1. We use a 3×3 convolutional layer with a padding of 1 to transform the CNN output and set the number of output channels to c. This way, each element in the feature map the CNN extracts from the image is a new feature with a length of c.
2. We use each element in the feature map as a center to generate multiple ABs of different sizes and aspect ratios and then label them.
3. We use the features of the elements of length c at the center on the ABs to predict the binary category (object or background) and BB for their respective ABs.
4. Then, we use non-maximum suppression to remove similar BB results that correspond to category predictions of "object". Finally, we output the predicted BBs as the proposed regions required by the RoI pooling layer.

NOTE: As a part of the Faster R-CNN model, the region proposal network is trained together with the rest of the model.

In addition, the Faster R-CNN object functions include the category and BB predictions in object detection, as well as the binary category and BB predictions for the ABs in the region proposal network.

Finally, the region proposal network can learn how to generate high-quality proposed regions, which reduces the number of proposed regions while maintaining the precision of object detection.

Mask R-CNN

If training data is labeled with the pixel-level positions of each object in an image, a **Mask R-CNN** model can effectively use these detailed labels to further improve the precision of object detection.

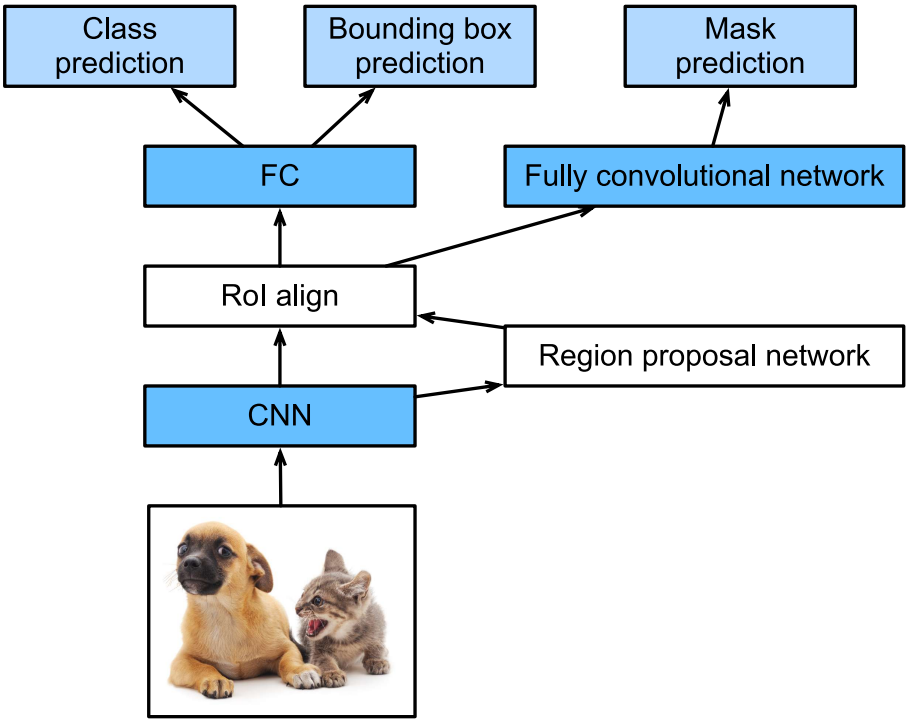


Figure. Mask R-CNN model.

As shown in this Figure, Mask R-CNN is a modification to the Faster R-CNN model.

Mask R-CNN models replace the RoI pooling layer with an RoI alignment layer.

This allows the use of bilinear interpolation to retain spatial information on feature maps, making Mask R-CNN better suited for pixel-level predictions.

The RoI alignment layer outputs feature maps of the same shape for all RoIs.

This not only predicts the categories and bounding boxes of RoIs, but allows us to use an additional fully convolutional network to predict the pixel-level positions of objects.

Summary

- **R-CNN model** selects several proposed regions and uses a CNN to perform forward computation and extract the features from each proposed region. It then uses these features to predict the categories and bounding boxes of proposed regions.
- **Fast R-CNN** improves on the R-CNN by **only performing CNN forward computation on the image as a whole**. It introduces an RoI pooling layer to extract features of the same shape from RoIs of different shapes.
- **Faster R-CNN** improves by **replacing** the selective search used in Fast R-CNN **by a region proposal network**. This reduces the number of proposed regions generated, while ensuring precise object detection.
- **Mask R-CNN** uses the same basic structure as Faster R-CNN, but **adds a fully convolution layer** to help locate objects at the pixel level and further improve the precision of object detection.

Part 2.Object Detection - Use Any Image Classifier as an Object Detector

Let's implement **a naive approach** that allows us to turn any image classifier into an object detector, by leveraging the iterative approach of extracting ROIs (sliding windows) at different levels of perspective (image pyramid).

```
import cv2
import imutils
import numpy as np
from tensorflow.keras.applications import imagenet_utils
from tensorflow.keras.applications.inception_resnet_v2 import *
from tensorflow.keras.applications.vgg19 import *
from tensorflow.keras.applications.mobilenet_v2 import *
from tensorflow.keras.preprocessing.image import img_to_array

import matplotlib.pyplot as plt
```

```
class ObjectDetector(object):
    def __init__(self, classifier,
                 preprocess_fn=lambda x: x,
                 #input_size=(299, 299), # For inception_resnet_v2
                 input_size=(224, 224), # For VGG
                 confidence=0.98,
                 window_step_size=16,
                 pyramid_scale=1.5,
                 roi_size=(200, 150),
                 nms_threshold=0.3):
        self.classifier = classifier
```

```
        self.preprocess_fn = preprocess_fn
        self.input_size = input_size
        self.confidence = confidence

        self.window_step_size = window_step_size

        self.pyramid_scale = pyramid_scale
        self.roi_size = roi_size
        self.nms_threshold = nms_threshold

    def sliding_window(self, image):
        for y in range(0,
                       image.shape[0],
                       self.window_step_size):
            for x in range(0,
                           image.shape[1],
                           self.window_step_size):
                y_slice = slice(y, y + self.roi_size[1], 1)
                x_slice = slice(x, x + self.roi_size[0], 1)

                yield x, y, image[y_slice, x_slice]

    def pyramid(self, image):
        yield image

        while True:
            width = int(image.shape[1] / self.pyramid_scale)
            image = imutils.resize(image, width=width)

            if (image.shape[0] < self.roi_size[1] or
                    image.shape[1] < self.roi_size[0]):
                break

            yield image

    def non_max_suppression(self, boxes, probabilities):
        if len(boxes) == 0:
            return []

        if boxes.dtype.kind == 'i':
            boxes = boxes.astype(np.float)

        pick = []

        x_1 = boxes[:, 0]
        y_1 = boxes[:, 1]
        x_2 = boxes[:, 2]
        y_2 = boxes[:, 3]

        area = (x_2 - x_1 + 1) * (y_2 - y_1 + 1)
        indexes = np.argsort(probabilities)

        while len(indexes) > 0:
            last = len(indexes) - 1
            i = indexes[last]
```

```
            pick.append(i)

            xx_1 = np.maximum(x_1[i], x_1[indexes[:last]])
            yy_1 = np.maximum(y_1[i], y_1[indexes[:last]])
            xx_2 = np.maximum(x_2[i], x_2[indexes[:last]])
            yy_2 = np.maximum(y_2[i], y_2[indexes[:last]])

            width = np.maximum(0, xx_2 - xx_1 + 1)
            height = np.maximum(0, yy_2 - yy_1 + 1)

            overlap = (width * height) / area[indexes[:last]]

            redundant_boxes = \
                np.where(overlap > self.nms_threshold)[0]
            to_delete = np.concatenate(
                ([last], redundant_boxes))
            indexes = np.delete(indexes, to_delete)

        return boxes[pick].astype(np.int)

    def detect(self, image):
        rois = []
        locations = []

        for img in self.pyramid(image):
            scale = image.shape[1] / float(img.shape[1])

            for x, y, roi_original in \
                    self.sliding_window(img):
                x = int(x * scale)
                y = int(y * scale)
                w = int(self.roi_size[0] * scale)
                h = int(self.roi_size[1] * scale)

                roi = cv2.resize(roi_original,
                                 self.input_size)
                roi = img_to_array(roi)
                roi = self.preprocess_fn(roi)

                rois.append(roi)
                locations.append((x, y, x + w, y + h))

        rois = np.array(rois, dtype=np.float32)

        predictions = self.classifier.predict(rois)
        predictions = \
            imagenet_utils.decode_predictions(predictions,
                                              top=1)

        labels = {}
        for i, pred in enumerate(predictions):
            _, label, proba = pred[0]

            if proba >= self.confidence:
                box = locations[i]
```

```
                label_detections = labels.get(label, [])
                label_detections.append({'box': box,
                                         'proba': proba})
                labels[label] = label_detections

        return labels
```

```
%%time

#model = InceptionResNetV2(weights='imagenet',  # 20 sec
model = MobileNetV2(weights='imagenet', # 1.2 sec
#model = VGG19(weights='imagenet',  # 13 sec
                          include_top=True)
object_detector = ObjectDetector(model, preprocess_input)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
14540800/14536120 [==============================] - 0s 0us/step
CPU times: user 988 ms, sys: 50.1 ms, total: 1.04 s
Wall time: 1.2 s
```

```
image = cv2.imread('dog.jpg')
#image = cv2.imread('cat.jpg')
image = imutils.resize(image, width=600)
```

```
%%time

# MobileNetV2, 42 sec - 1st attempt
# VGG19, 1 min 12 sec - 1st attempt, 35 sec - 2 attempt

labels = object_detector.detect(image)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/
40960/35363 [==================================] - 0s 0us/step
CPU times: user 7.36 s, sys: 7.37 s, total: 14.7 s
Wall time: 42.5 s
```

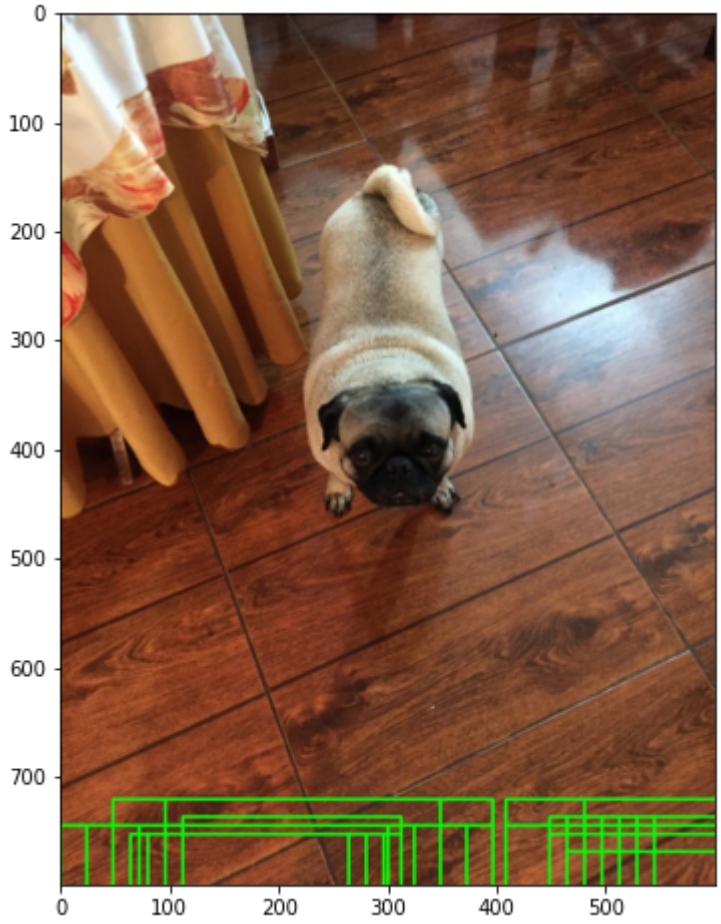
```
print(labels)
```

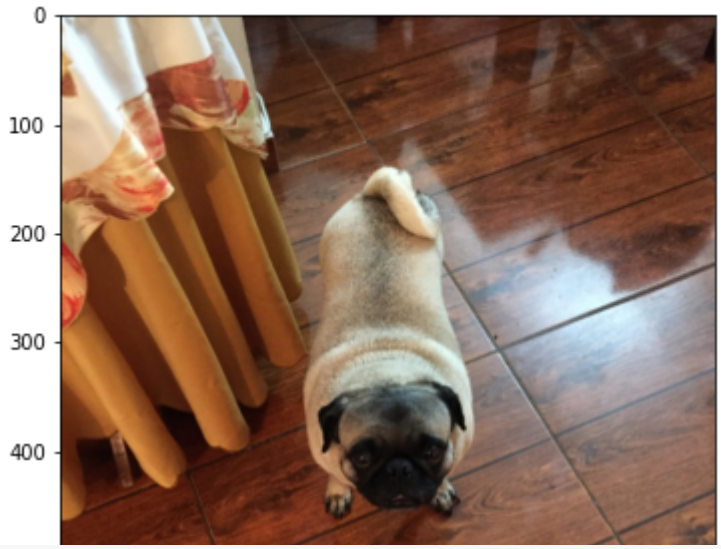
```
{'velvet': [{'box': (112, 736, 312, 886), 'proba': 0.98589164}, {'box': (448,
```

```
%%time

GREEN = (0, 255, 0)
for i, label in enumerate(labels.keys()):
    clone = image.copy()

    for detection in labels[label]:
        box = detection['box']
```

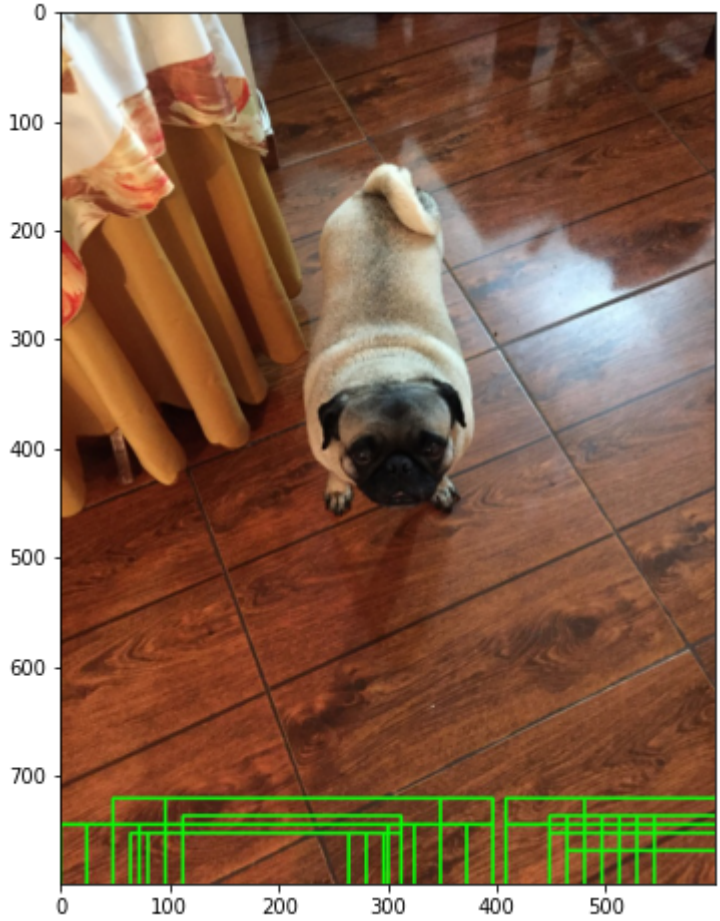
```
        probability = detection['proba']

        x_start, y_start, x_end, y_end = box
        cv2.rectangle(clone, (x_start, y_start),
                      (x_end, y_end), (0, 255, 0), 2)

    cv2.imwrite(f'Before_{i}.jpg', clone)
    filename = 'Before_' + str(i) + '.jpg'
    plt.figure(figsize=(12,8))
    img = plt.imread(filename)
    plt.imshow(img)

    clone = image.copy()
    boxes = np.array([d['box'] for d in labels[label]])
    probas = np.array([d['proba'] for d in labels[label]])
    boxes = object_detector.non_max_suppression(boxes,
                                                probas)

    for x_start, y_start, x_end, y_end in boxes:
        cv2.rectangle(clone,
                      (x_start, y_start),
                      (x_end, y_end),
                      GREEN,
                      2)

        if y_start - 10 > 10:
            y = y_start - 10
        else:
            y = y_start + 10

        cv2.putText(clone,
                    label,
                    (x_start, y),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    .45,
                    GREEN,
                    2)

    cv2.imwrite(f'After_{i}.jpg', clone)
    filename = 'After_' + str(i) + '.jpg'
    plt.figure(figsize=(12,8))
    img = plt.imread(filename)
    plt.imshow(img)
```

```
CPU times: user 71 ms, sys: 1.16 ms, total: 72.1 ms
Wall time: 84.9 ms
```






```
plt.figure(figsize=(12,8))
img = plt.imread('./Before_0.jpg')
plt.imshow(img);

plt.figure(figsize=(12,8))
img = plt.imread('./After_0.jpg')
plt.imshow(img);
```




```
!ls -all
```

```
total 1088
drwxr-xr-x 1 root root   4096 Apr 13 07:36 .
drwxr-xr-x 1 root root   4096 Apr 13 07:27 ..
-rw-r--r-- 1 root root 182707 Apr 13 07:56 After_0.jpg
-rw------- 1 root root  44367 Apr 13 07:34 banana.jpg
-rw-r--r-- 1 root root 184017 Apr 13 07:56 Before_0.jpg
-rw------- 1 root root  32576 Apr 13 07:34 catdog.jpg
```

```
drwxr-xr-x 4 root root   4096 Apr  7 13:35 .config
-rw------- 1 root root 643434 Apr 13 07:34 dog.jpg
drwx------ 5 root root   4096 Apr 13 07:33 drive
drwxr-xr-x 1 root root   4096 Apr  7 13:36 sample_data
```

Summary

In this section, we implemented a reusable class that easily allows us to turn any image classifier into an object detector, by leveraging the iterative approach of extracting ROIs (sliding windows) at different levels of perspective (image pyramid) and passing them to such a classifier to determine where objects are in a photo, and what they are.

Also, we used NMS to reduce the amount of non-informative, duplicate detections that are characteristic of this strategy.

Although this a great first attempt at creating an object detector, it has its flaws:

- It's incredibly slow, which makes it unusable in real-time situations.
- The accuracy of the bounding boxes depends heavily on the parameter selection for the image pyramid, the sliding window, and the ROI size.
- The architecture is not end-to-end trainable, which means that errors in BB predictions are not backpropagated through the network in order to produce better, more accurate detections in the future, by updating its weights.

Instead, we're stuck with pre-trained models that limit themselves to infer but not to learn because the framework does not allow them to.

However, don't rule out this approach yet! If you're working with images that present very little variation in size and perspective, and your application definitely doesn't operate in a real-time context, the strategy implemented in this recipe can work wonders for your project!

Part 3.Object Detection - Use End-to-End Object Detector - YOLO

YOLO (You Only Look Once) is one of the fastest object detection algorithms available.

The latest version, YOLOv3, can run at more than 170 frames per second (FPS) on a modern GPU for an image size of 256 × 256.

In this section, we will introduce the theoretical concept behind its architecture.

First released in 2015, YOLO outperformed almost all other object detection architectures, both in terms of speed and accuracy. Since then, the architecture has been improved several times.

The more detailed information can be obtained from the following three papers:

- You Only Look Once: Unified, real-time object detection (2015), Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi
- YOLO9000: Better, Faster, Stronger (2016), Joseph Redmon and Ali Farhadi
- YOLOv3: An Incremental Improvement (2018), Joseph Redmon and Ali Farhadi, arXiv preprint arXiv:1804.02767.
- YOLOv4: Optimal speed and accuracy of object detection (2020). Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. arXiv preprint arXiv:2004.10934.
- YOLOv5 ultralytics/yolov5, Glenn Jocher; Alex Stoken; Jirka Borovec; NanoCode012; Ayush Chaurasia; TaoXie; Liu Changyu; Abhiram V; Laughing; tkianai; yxNONG; Adam Hogan; lorenzomammana; AlexWang1900; Jan Hajek; Laurentiu Diaconu; Marc; Yonghye Kwon; oleg; wanghaoyang0106; Yann Defretin; Aditya Lohia; ml5ah; Ben Milanko; Benjamin Fineran; Daniel Khromov; Ding Yiwei; Doug; Durgesh; Francisco Ingham, **April 11, 2021 ... :)**

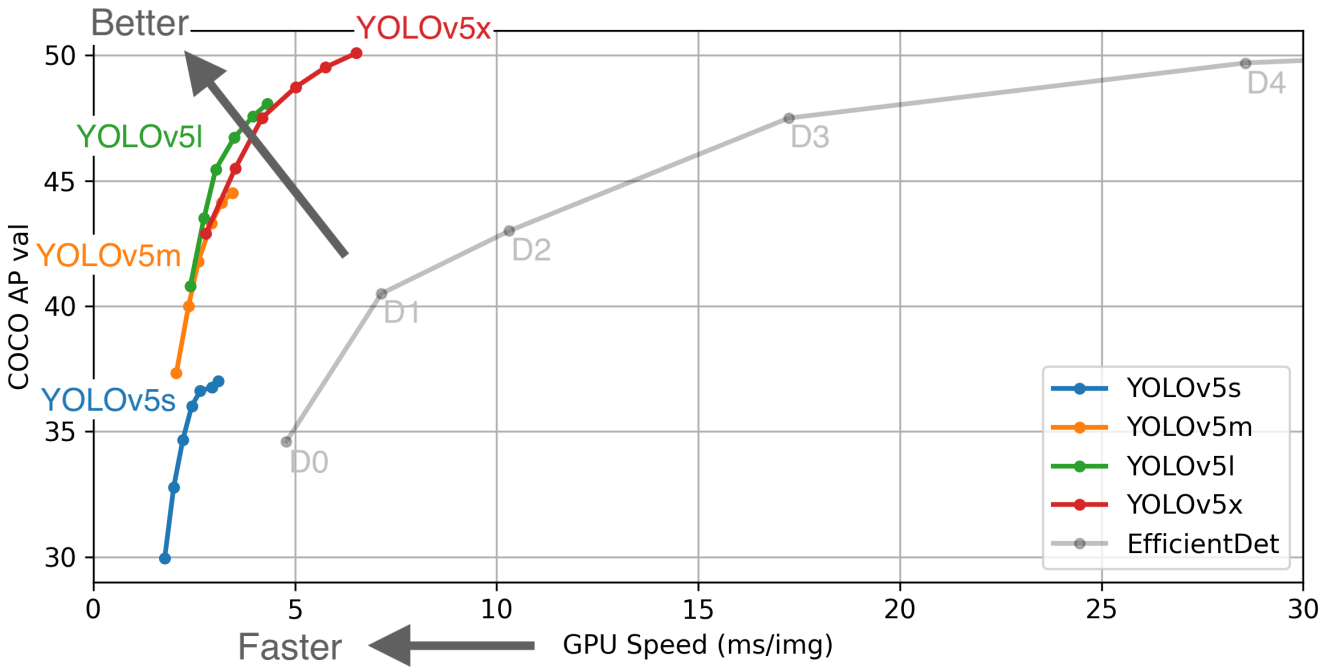


Figure. GPU Speed measures end-to-end time per image averaged over 5000 COCO val2017 images using a V100 GPU with batch size 32, and includes image preprocessing, PyTorch FP16 inference, postprocessing and NMS. EfficientDet data from google/automl at batch size 8.

NOTE: The main author of the YOLO paper maintains a deep learning framework called Darknet. This hosts the official implementation of YOLO and can be used to reproduce the paper's results. It is coded in C++ and is not based on TensorFlow.

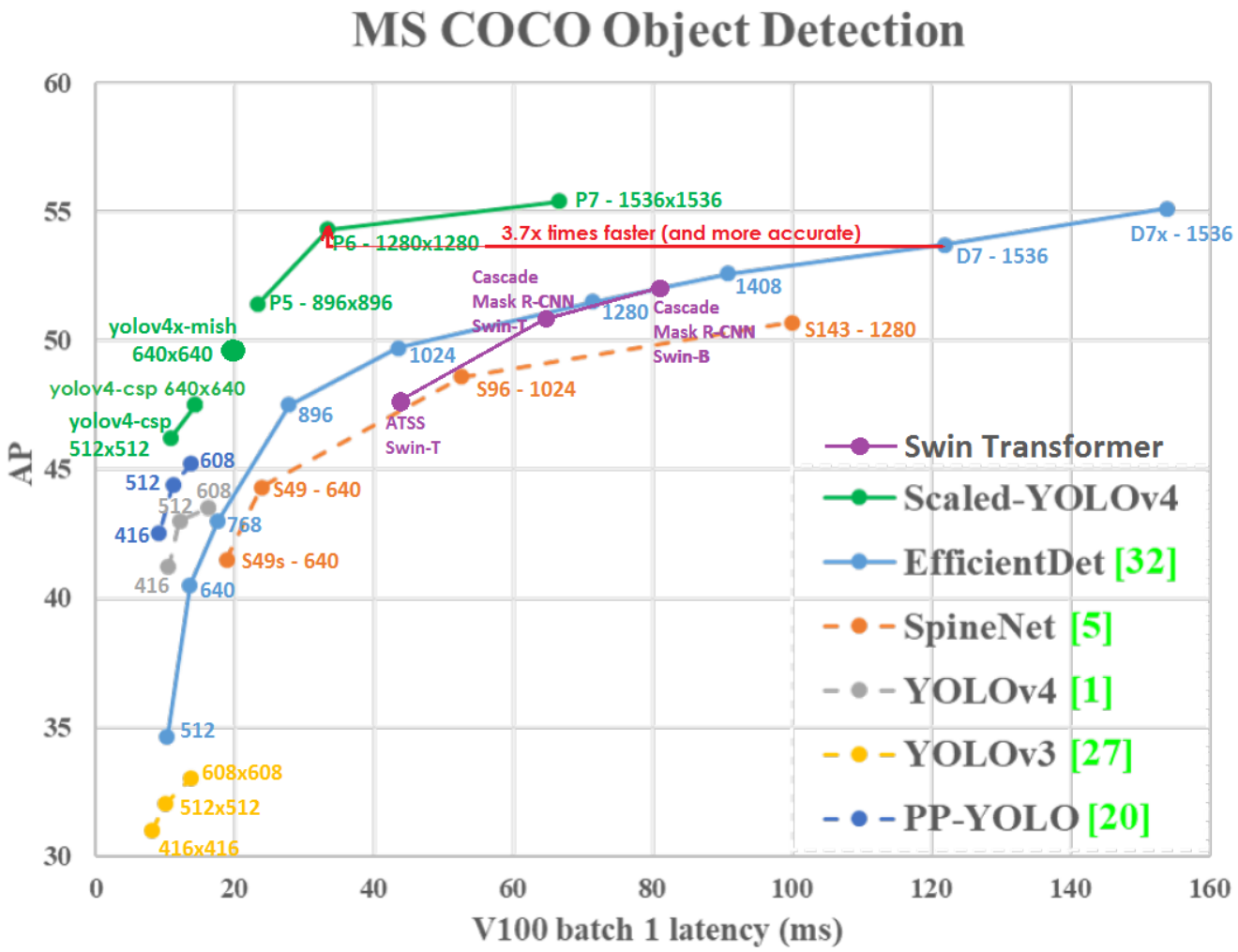


Figure. Latency comparison.

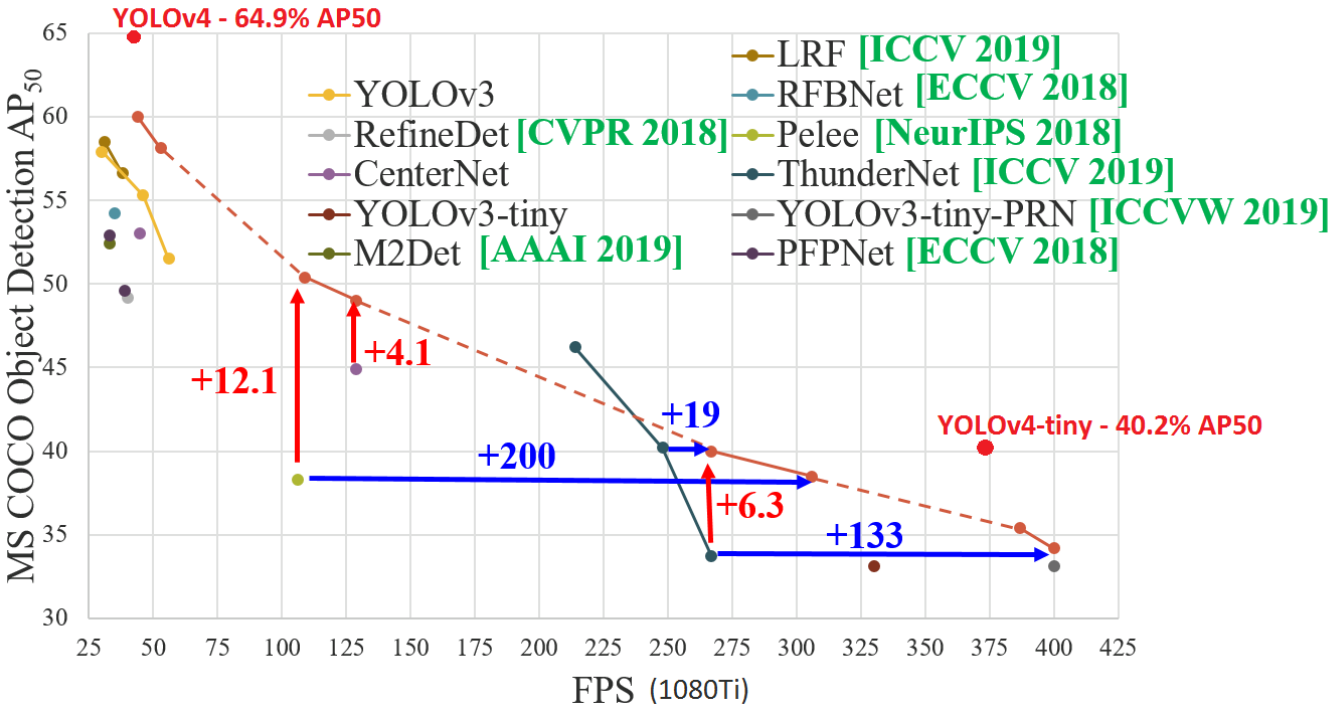


Figure. AP50:95 - FPS (Tesla V100) comparison.

Workflow Description

In the previous Part we learned how to turn any image classifier into an object detector, by embedding it in a traditional framework that relies on image pyramids and sliding windows.

BUT, we also learned that this approach isn't ideal because it doesn't allow the network to learn from its mistakes.

The reason why DL has conquered the field of object detection is due to its end-to-end approach:

- the network **not only** figures out how to **classify** an object,
- but also discovers how to **produce the best bounding box** possible to locate each element in the image.

Thanks to this end-to-end strategy, a network can detect a myriad objects in a single pass!

Of course, this makes such object detectors incredibly efficient!

One of the seminal end-to-end object detectors is **YOLO**, and in this recipe, we'll learn how to detect objects with a pre-trained YOLOv3 model.

Our implementation is heavily inspired by the amazing keras-yolo3 repository implemented by Huynh Ngoc Anh (on GitHub as `experiencor`), which you can consult here:

https://github.com/experiencor/keras-yolo3

Because we'll use a pre-trained YOLO model, we need to download the weights: https://pjreddie.com/media/files/yolov3.weights

NOTE: These weights are the same ones used by the original authors of YOLO. Refer to the See also section to learn more about YOLO.

Strengths and Limitations

YOLO is known for its speed. BUT ...

- it has been recently outperformed in terms of accuracy **bold text** by Faster R-CNN, but ... this statement should be proved! :)
- due to the way it detects objects, YOLO is **NOT so good with smaller objects** (for example, it would have trouble detecting single birds from a flock), but ... this statement should be proved! :)
- it also struggles to properly **detect objects that deviate too much** from the training set (unusual aspect ratios or appearance), but ... it is typical for most DL models! :)

Nevertheless, the architecture is constantly evolving, and those issues are being worked on.

Main Concepts

The core idea of YOLO is this: **reframe object detection as a single regression problem.**

Instead of using a sliding window or another complex technique, we will divide the input into a $w \times h$ grid, as represented in this diagram:

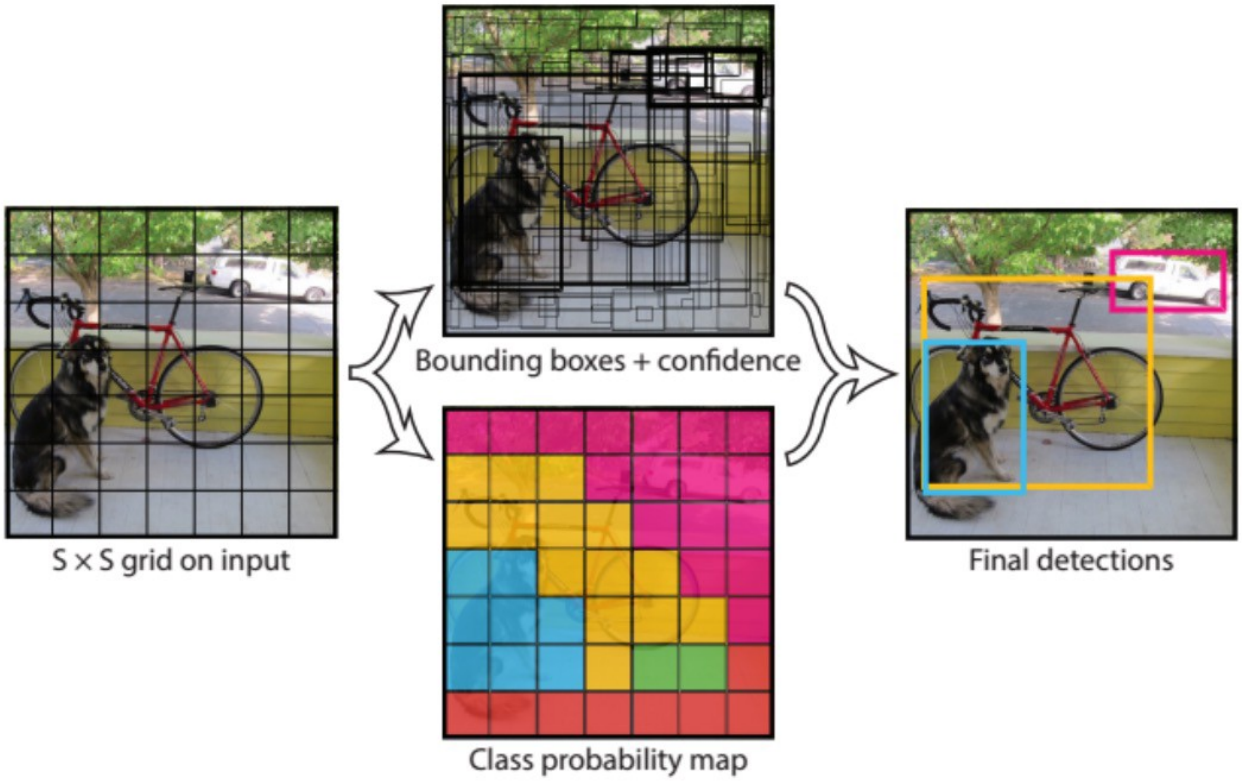


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

For each part of the grid, we will define *B* BBs.

Our only task will be to predict the following for each bounding box:

- the center of the BB,
- the width and height of the BB,
- the probability that this BB contains an object,
- the class of the object.

Since all those predictions are numbers, we have therefore transformed the object detection problem into a regression problem.

It is important to make a distinction between the grid cells that divide the pictures into equal parts (w × h parts to be precise) and the bounding boxes that will locate the objects. Each grid cell contains B BBs. Therefore, there will be w × h × B possible bounding boxes in the end.

In practice, the concepts used by YOLO are a bit more complex than this:

- What if there are several objects in one part of the grid?
- What if an object overlaps several parts of the grid?
- How do we choose a loss to train our model? ... But all these questions are much out of

```
import glob
import json
import struct

import matplotlib.pyplot as plt
import numpy as np
import tqdm
from matplotlib.patches import Rectangle
from tensorflow.keras.layers import *
from tensorflow.keras.models import *
from tensorflow.keras.preprocessing.image import *
```

```
class WeightReader:
    def __init__(self, weight_file):
        with open(weight_file, 'rb') as w_f:
            major, = struct.unpack('i', w_f.read(4))
            minor, = struct.unpack('i', w_f.read(4))
            revision, = struct.unpack('i', w_f.read(4))

            if (major * 10 + minor) >= 2 and \
                    major < 1000 and \
                    minor < 1000:
                w_f.read(8)
            else:
                w_f.read(4)

            binary = w_f.read()

        self.offset = 0
        self.all_weights = np.frombuffer(binary,
                                         dtype='float32')

    def read_bytes(self, size):
        self.offset = self.offset + size
        return self.all_weights[self.offset - size:self.offset]

    def load_weights(self, model):
        for i in tqdm.tqdm(range(106)):
            try:
                conv_layer = model.get_layer(f'conv_{i}')

                if i not in [81, 93, 105]:
                    norm_layer = model.get_layer(f'bnorm_{i}')
                    size = np.prod(norm_layer.
                                   get_weights()[0].shape)
                    bias = self.read_bytes(size)
                    scale = self.read_bytes(size)
                    mean = self.read_bytes(size)
                    var = self.read_bytes(size)
```

```
                    norm_layer.set_weights([scale, bias,
                                            mean, var])

                if len(conv_layer.get_weights()) > 1:
                    bias = self.read_bytes(np.prod(
                        conv_layer.get_weights()[1].shape))

                    kernel = self.read_bytes(np.prod(
                        conv_layer.get_weights()[0].shape))

                    kernel = kernel.reshape(list(reversed(
                        conv_layer.get_weights()[0].shape)))

                    kernel = kernel.transpose([2, 3, 1, 0])

                    conv_layer.set_weights([kernel, bias])
                else:
                    kernel = self.read_bytes(np.prod(
                        conv_layer.get_weights()[0].shape))

                    kernel = kernel.reshape(list(reversed(
                        conv_layer.get_weights()[0].shape)))

                    kernel = kernel.transpose([2, 3, 1, 0])

                    conv_layer.set_weights([kernel])
            except ValueError:
                pass

    def reset(self):
        self.offset = 0


class BoundBox(object):
    def __init__(self, x_min, y_min, x_max, y_max,
                 objness=None,
                 classes=None):
        self.xmin = x_min
        self.ymin = y_min
        self.xmax = x_max
        self.ymax = y_max
        self.objness = objness
        self.classes = classes
        self.label = -1
        self.score = -1

    def get_label(self):
        if self.label == -1:
            self.label = np.argmax(self.classes)

        return self.label

    def get_score(self):
        if self.score == -1:
```

```
            self.score = self.classes[self.get_label()]

        return self.score


class YOLO(object):
    def __init__(self, weights_path,
                 anchors_path='resources/anchors.json',
                 labels_path='resources/coco_labels.txt',
                 class_threshold=0.65):
        self.weights_path = weights_path
        self.model = self._load_yolo()

        self.labels = []
        with open(labels_path, 'r') as f:
            for l in f:
                self.labels.append(l.strip())

        with open(anchors_path, 'r') as f:
            self.anchors = json.load(f)

        self.class_threshold = class_threshold

    def _conv_block(self, input, convolutions, skip=True):
        x = input
        count = 0
        for conv in convolutions:
            if count == (len(convolutions) - 2) and skip:
                skip_connection = x

            count += 1

            if conv['stride'] > 1:
                x = ZeroPadding2D(((1, 0), (1, 0)))(x)

            x = Conv2D(conv['filter'],
                       conv['kernel'],
                       strides=conv['stride'],
                       padding=('valid' if conv['stride'] > 1
                                else 'same'),
                       name=f'conv_{conv["layer_idx"]}',
                       use_bias=(False if conv['bnorm']
                                 else True))(x)

            if conv['bnorm']:
                name = f'bnorm_{conv["layer_idx"]}'
                x = BatchNormalization(epsilon=1e-3,
                                       name=name)(x)
            if conv['leaky']:
                name = f'leaky_{conv["layer_idx"]}'
                x = LeakyReLU(alpha=0.1, name=name)(x)

        return Add()([skip_connection, x]) if skip else x

    def _make_yolov3_architecture(self):
```

```
        input_image = Input(shape=(None, None, 3))

        # Layer  0 => 4
        x = self._conv_block(input_image, [
            {'filter': 32, 'kernel': 3, 'stride': 1,
             'bnorm': True,
             'leaky': True, 'layer_idx': 0},
            {'filter': 64, 'kernel': 3, 'stride': 2,
             'bnorm': True,
             'leaky': True, 'layer_idx': 1},
            {'filter': 32, 'kernel': 1, 'stride': 1,
             'bnorm': True,
             'leaky': True, 'layer_idx': 2},
            {'filter': 64, 'kernel': 3, 'stride': 1,
             'bnorm': True,
             'leaky': True, 'layer_idx': 3}])

        # Layer  5 => 8
        x = self._conv_block(x, [
            {'filter': 128, 'kernel': 3, 'stride': 2,
             'bnorm': True, 'leaky': True, 'layer_idx': 5},
            {'filter': 64, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 6},
            {'filter': 128, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 7}])

        # Layer  9 => 11
        x = self._conv_block(x, [
            {'filter': 64, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 9},
            {'filter': 128, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 10}])

        # Layer 12 => 15
        x = self._conv_block(x, [
            {'filter': 256, 'kernel': 3, 'stride': 2,
             'bnorm': True, 'leaky': True, 'layer_idx': 12},
            {'filter': 128, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 13},
            {'filter': 256, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 14}])

        # Layer 16 => 36
        for i in range(7):
            x = self._conv_block(x, [
                {'filter': 128, 'kernel': 1, 'stride': 1,
                 'bnorm': True, 'leaky': True,
                 'layer_idx': 16 + i * 3},
                {'filter': 256, 'kernel': 3, 'stride': 1,
                 'bnorm': True, 'leaky': True,
                 'layer_idx': 17 + i * 3}])
        skip_36 = x

        # Layer 37 => 40
        x = self._conv_block(x, [
```

```
            {'filter': 512, 'kernel': 3, 'stride': 2,
             'bnorm': True, 'leaky': True, 'layer_idx': 37},
            {'filter': 256, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 38},
            {'filter': 512, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 39}])

        # Layer 41 => 61
        for i in range(7):
            x = self._conv_block(x, [
                {'filter': 256, 'kernel': 1, 'stride': 1,
                 'bnorm': True, 'leaky': True,
                 'layer_idx': 41 + i * 3},
                {'filter': 512, 'kernel': 3, 'stride': 1,
                 'bnorm': True, 'leaky': True,
                 'layer_idx': 42 + i * 3}])
        skip_61 = x

        # Layer 62 => 65
        x = self._conv_block(x, [
            {'filter': 1024, 'kernel': 3, 'stride': 2,
             'bnorm': True, 'leaky': True, 'layer_idx': 62},
            {'filter': 512, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 63},
            {'filter': 1024, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 64}])

        # Layer 66 => 74
        for i in range(3):
            x = self._conv_block(x, [
                {'filter': 512, 'kernel': 1, 'stride': 1,
                 'bnorm': True, 'leaky': True,
                 'layer_idx': 66 + i * 3},
                {'filter': 1024, 'kernel': 3, 'stride': 1,
                 'bnorm': True, 'leaky': True,
                 'layer_idx': 67 + i * 3}])

        # Layer 75 => 79
        x = self._conv_block(x, [
            {'filter': 512, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 75},
            {'filter': 1024, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 76},
            {'filter': 512, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 77},
            {'filter': 1024, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 78},
            {'filter': 512, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 79}],
                            skip=False)

        # Layer 80 => 82
        yolo_82 = self._conv_block(x, [
            {'filter': 1024, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 80},
```

```
            {'filter': 255, 'kernel': 1, 'stride': 1,
             'bnorm': False, 'leaky': False,
             'layer_idx': 81}], skip=False)

        # Layer 83 => 86
        x = self._conv_block(x, [
            {'filter': 256, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 84}],
                             skip=False)
        x = UpSampling2D(2)(x)
        x = Concatenate()([x, skip_61])

        # Layer 87 => 91
        x = self._conv_block(x, [
            {'filter': 256, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 87},
            {'filter': 512, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 88},
            {'filter': 256, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 89},
            {'filter': 512, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 90},
            {'filter': 256, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 91}],
                             skip=False)
        # Layer 92 => 94
        yolo_94 = self._conv_block(x, [
            {'filter': 512, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 92},
            {'filter': 255, 'kernel': 1, 'stride': 1,
             'bnorm': False, 'leaky': False,
             'layer_idx': 93}], skip=False)

        # Layer 95 => 98
        x = self._conv_block(x, [
            {'filter': 128, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 96}],
                             skip=False)
        x = UpSampling2D(2)(x)
        x = Concatenate()([x, skip_36])

        # Layer 99 => 106
        yolo_106 = self._conv_block(x, [
            {'filter': 128, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 99},
            {'filter': 256, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 100},
            {'filter': 128, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 101},
            {'filter': 256, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 102},
            {'filter': 128, 'kernel': 1, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 103},
            {'filter': 256, 'kernel': 3, 'stride': 1,
             'bnorm': True, 'leaky': True, 'layer_idx': 104},
```

```
            {'filter': 255, 'kernel': 1, 'stride': 1,
             'bnorm': False, 'leaky': False,
             'layer_idx': 105}], skip=False)

        return Model(inputs=input_image,
                     outputs=[yolo_82, yolo_94, yolo_106])

    def _load_yolo(self):
        model = self._make_yolov3_architecture()
        weight_reader = WeightReader(self.weights_path)
        weight_reader.load_weights(model)
        model.save('model.h5')

        model = load_model('model.h5')

        return model

    @staticmethod
    def _sigmoid(x):
        return 1.0 / (1.0 + np.exp(-x))

    def _decode_net_output(self, network_output,
                           anchors,
                           obj_thresh,
                           network_height,
                           network_width):
        grid_height, grid_width = network_output.shape[:2]
        nb_box = 3
        network_output = network_output.reshape(
            (grid_height, grid_width, nb_box, -1))

        boxes = []
        network_output[..., :2] = \
            self._sigmoid(network_output[..., :2])
        network_output[..., 4:] = \
            self._sigmoid(network_output[..., 4:])
        network_output[..., 5:] = \
            (network_output[..., 4][..., np.newaxis] *
             network_output[..., 5:])
        network_output[..., 5:] *= \
            network_output[..., 5:] > obj_thresh

        for i in range(grid_height * grid_width):
            r = i / grid_width
            c = i % grid_width

            for b in range(nb_box):
                objectness = \
                    network_output[int(r)][int(c)][b][4]

                if objectness.all() <= obj_thresh:
                    continue

                x, y, w, h = \
                    network_output[int(r)][int(c)][b][:4]
```

```
                x = (c + x) / grid_width
                y = (r + y) / grid_height
                w = (anchors[2 * b] * np.exp(w) /
                     network_width)
                h = (anchors[2 * b + 1] * np.exp(h) /
                     network_height)

                classes = network_output[int(r)][c][b][5:]
                box = BoundBox(x_min=x - w / 2,
                               y_min=y - h / 2,
                               x_max=x + w / 2,
                               y_max=y + h / 2,
                               objness=objectness,
                               classes=classes)
                boxes.append(box)

        return boxes

    @staticmethod
    def _correct_yolo_boxes(boxes,
                            image_height,
                            image_width,
                            network_height,
                            network_width):
        new_w, new_h = network_width, network_height

        for i in range(len(boxes)):
            x_offset = (network_width - new_w) / 2.0
            x_offset /= network_width
            x_scale = float(new_w) / network_width

            y_offset = (network_height - new_h) / 2.0
            y_offset /= network_height
            y_scale = float(new_h) / network_height

            boxes[i].xmin = int((boxes[i].xmin - x_offset) /
                                x_scale * image_width)
            boxes[i].xmax = int((boxes[i].xmax - x_offset) /
                                x_scale * image_width)
            boxes[i].ymin = int((boxes[i].ymin - y_offset) /
                                y_scale * image_height)
            boxes[i].ymax = int((boxes[i].ymax - y_offset) /
                                y_scale * image_height)

    @staticmethod
    def _interval_overlap(interval_a, interval_b):
        x1, x2 = interval_a
        x3, x4 = interval_b

        if x3 < x1:
            if x4 < x1:
                return 0
            else:
                return min(x2, x4) - x1
        else:
```

```
            if x2 < x3:
                return 0
            else:
                return min(x2, x4) - x3

    def _bbox_iou(self, box1, box2):
        intersect_w = self._interval_overlap(
            [box1.xmin, box1.xmax],
            [box2.xmin, box2.xmax])
        intersect_h = self._interval_overlap(
            [box1.ymin, box1.ymax],
            [box2.ymin, box2.ymax])

        intersect = intersect_w * intersect_h

        w1, h1 = box1.xmax - box1.xmin, box1.ymax - box1.ymin
        w2, h2 = box2.xmax - box2.xmin, box2.ymax - box2.ymin

        union = w1 * h1 + w2 * h2 - intersect
        return float(intersect) / union

    def _non_max_suppression(self, boxes, nms_thresh):
        if len(boxes) > 0:
            nb_class = len(boxes[0].classes)
        else:
            return

        for c in range(nb_class):
            sorted_indices = np.argsort(
                [-box.classes[c] for box in boxes])

            for i in range(len(sorted_indices)):
                index_i = sorted_indices[i]

                if boxes[index_i].classes[c] == 0:
                    continue

                for j in range(i + 1, len(sorted_indices)):
                    index_j = sorted_indices[j]
                    iou = self._bbox_iou(boxes[index_i],
                                         boxes[index_j])
                    if iou >= nms_thresh:
                        boxes[index_j].classes[c] = 0

    def _get_boxes(self, boxes):
        v_boxes, v_labels, v_scores = [], [], []

        for box in boxes:
            for i in range(len(self.labels)):
                if box.classes[i] > self.class_threshold:
                    v_boxes.append(box)
                    v_labels.append(self.labels[i])
                    v_scores.append(box.classes[i] * 100)

        return v_boxes, v_labels, v_scores
```

```
    @staticmethod
    def _draw_boxes(filename, v_boxes, v_labels, v_scores):
        data = plt.imread(filename)
        plt.imshow(data)

        ax = plt.gca()

        for i in range(len(v_boxes)):
            box = v_boxes[i]

            y1, x1, y2, x2 = \
                box.ymin, box.xmin, box.ymax, box.xmax

            width = x2 - x1
            height = y2 - y1

            rectangle = Rectangle((x1, y1), width, height,
                                  fill=False, color='white')

            ax.add_patch(rectangle)
            label = f'{v_labels[i]} ({v_scores[i]:.3f})'
            plt.text(x1, y1, label, color='yellow')
        plt.show()

    def detect(self, image, width, height):
        image = np.expand_dims(image, axis=0)
        preds = self.model.predict(image)

        boxes = []

        for i in range(len(preds)):
            boxes.extend(
                self._decode_net_output(preds[i][0],
                                        self.anchors[i],
                                        self.class_threshold,
                                        416,
                                        416))

        self._correct_yolo_boxes(boxes, height, width, 416,
                                 416)
        self._non_max_suppression(boxes, .5)

        valid_boxes, valid_labels, valid_scores = \
            self._get_boxes(boxes)

        for i in range(len(valid_boxes)):
            print(valid_labels[i], valid_scores[i])

        self._draw_boxes(image_path,
                         valid_boxes,
                         valid_labels,
                         valid_scores)
```

```
! wget https://pjreddie.com/media/files/yolov3.weights
```

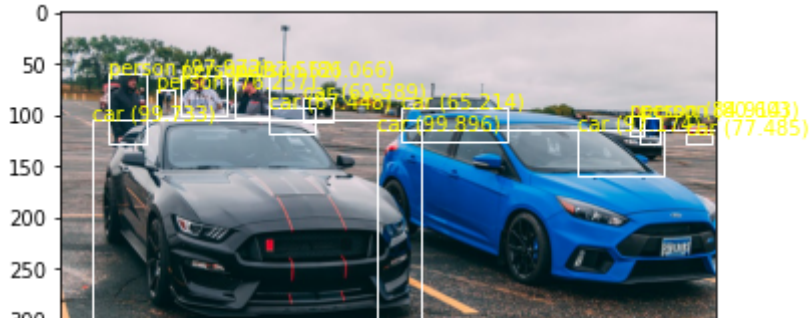
```
--2021-04-13 09:19:20--  https://pjreddie.com/media/files/yolov3.weights
Resolving pjreddie.com (pjreddie.com)... 128.208.4.108
Connecting to pjreddie.com (pjreddie.com)|128.208.4.108|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 248007048 (237M) [application/octet-stream]
Saving to: ‘yolov3.weights’

yolov3.weights      100%[===================>] 236.52M  23.1MB/s    in 11s

2021-04-13 09:19:31 (21.7 MB/s) - ‘yolov3.weights’ saved [248007048/248007048
```

```
model = YOLO(weights_path='yolov3.weights')
```

```
100%|██████████| 106/106 [00:01<00:00, 89.24it/s]
WARNING:tensorflow:No training configuration found in the save file, so the m
```

```
%%time

for image_path in glob.glob('test_images/*.jpg'):
    image = load_img(image_path, target_size=(416, 416))
    image = img_to_array(image)
    image = image.astype('float32') / 255.0

    original_image = load_img(image_path)
    width, height = original_image.size

    model.detect(image, width, height)
```

```
car 99.89618062973022
car 99.73328709602356
person 97.55168557167053
person 97.97219038009644
car 65.21447896957397
car 97.17357754707336
person 86.0664427280426
person 76.23717188835144
car 69.58921551704407
car 67.44755506515503
person 84.91390943527222
person 80.60300350189209
car 77.48510241508484
```



Summary

Even though the result is crowded, a quick glance reveals the network was able to identify both cars in the foreground, as well as the people in the background. This is an interesting example because it demonstrates the incredible power of YOLO as an end-to-end object detector, which in a single pass was capable of classifying and localizing many different objects, at varying scales.

In this section, we discovered the immense power of end-to-end object detectors— particularly, one of the most famous and impressive of all: YOLO.

Although YOLO was originally implemented in C++, we leveraged the fantastic Python adaptation by Huynh Ngoc Anh to perform object detection in our own images using a pre-trained version (specifically, version 3) of this architecture on the seminal COCO dataset.

As you might have noticed, YOLO and many other end-to-end object detectors are very complex networks, but their advantage over traditional approaches such as image pyramids and sliding windows is evident. Not only are the results way better, but they also come through faster thanks to the ability of YOLO to look once at the input image in order to produce all the relevant detections.

NOTE:

YOLO is a milestone when it comes to deep learning and object detection, so reading the paper is a pretty smart time investment. You can find it here: https://arxiv.org/abs/1506.02640 You can learn more about YOLO directly from the author's website, here: https://pjreddie.com/darknet/yolo/

If you are interested in exploring keras-yolo3 , the tool we based our implementation on, refer to this link: https://github.com/experiencor/keras-yolo3

Part 4.Object Detection - Use TensorFlow Object Detection API

But what if you want to train an end-to-end object detector on your own data?

Are you doomed to rely on out-of-the-box solutions?

Do you need to spend hours deciphering cryptic papers in order to implement such networks?

Well, that's one option, but there's another one, which we'll explore in the next recipe, and it entails the **TensorFlow Object Detection API**, an experimental repository of state-of- the-art architectures that will ease and boost your object detection endeavors!

NOTE: BUT ... this is very-very-very **"hemorrhoid method"** also!

Install necessary EXPERIMENTAL libraries

```
!  pip install tf_slim
```

```
Collecting tf_slim
  Downloading https://files.pythonhosted.org/packages/02/97/b0f4a64df018ca018
     |████████████████████████████████| 358kB 13.1MB/s
Requirement already satisfied: absl-py>=0.2.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages
Installing collected packages: tf-slim
Successfully installed tf-slim-1.1.0
```

```
! pip install tf-models-official
```

```
Collecting tf-models-official
  Downloading https://files.pythonhosted.org/packages/57/4a/23a08f8fd2747867e
     |████████████████████████████████| 1.1MB 17.3MB/s
Requirement already satisfied: scipy>=0.19.1 in /usr/local/lib/python3.7/dist
Collecting seqeval
  Downloading https://files.pythonhosted.org/packages/9d/2d/233c79d5b4e5ab1db
     |████████████████████████████████| 51kB 7.7MB/s
Requirement already satisfied: pycocotools in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: gin-config in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: Cython in /usr/local/lib/python3.7/dist-packag
Collecting py-cpuinfo>=3.3.0
  Downloading https://files.pythonhosted.org/packages/e6/ba/77120e44cbe971915
     |████████████████████████████████| 102kB 13.0MB/s
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: google-cloud-bigquery>=0.31.0 in /usr/local/li
Requirement already satisfied: psutil>=5.4.3 in /usr/local/lib/python3.7/dist
Collecting tensorflow-addons
  Downloading https://files.pythonhosted.org/packages/74/e3/56d2fe76f0bb7c88e
     |████████████████████████████████| 706kB 47.7MB/s
Requirement already satisfied: kaggle>=1.3.9 in /usr/local/lib/python3.7/dist
Requirement already satisfied: tf-slim>=1.1.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: tensorflow>=2.4.0 in /usr/local/lib/python3.7/
Requirement already satisfied: tensorflow-datasets in /usr/local/lib/python3.
Requirement already satisfied: google-api-python-client>=1.6.7 in /usr/local/
Collecting opencv-python-headless
  Downloading https://files.pythonhosted.org/packages/6d/6d/92f377bece9b0ec9c
     |████████████████████████████████| 37.6MB 73kB/s
Collecting dataclasses
  Downloading https://files.pythonhosted.org/packages/26/2f/1095cdc2868052dd1
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: tensorflow-hub>=0.6.0 in /usr/local/lib/python
Collecting pyyaml>=5.1
  Downloading https://files.pythonhosted.org/packages/7a/a5/393c087efdc78091a
     |████████████████████████████████| 645kB 48.7MB/s
Requirement already satisfied: Pillow in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: oauth2client in /usr/local/lib/python3.7/dist-
Collecting sentencepiece
  Downloading https://files.pythonhosted.org/packages/f5/99/e0808cb947ba10f57
     |████████████████████████████████| 1.2MB 50.3MB/s
Requirement already satisfied: pandas>=0.22.0 in /usr/local/lib/python3.7/dis
Collecting tensorflow-model-optimization>=0.4.1
  Downloading https://files.pythonhosted.org/packages/55/38/4fd48ea1bfcb0b6e3
     |████████████████████████████████| 174kB 61.1MB/s
Requirement already satisfied: scikit-learn>=0.21.3 in /usr/local/lib/python3
Requirement already satisfied: setuptools>=18.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: protobuf>=3.6.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: google-cloud-core<2.0dev,>=1.0.3 in /usr/local
Requirement already satisfied: google-resumable-media!=0.4.0,<0.5.0dev,>=0.3.
Requirement already satisfied: typeguard>=2.7 in /usr/local/lib/python3.7/dis
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: certifi in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: urllib3 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.7/di
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: python-slugify in /usr/local/lib/python3.7/dis
Requirement already satisfied: absl-py>=0.2.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: tensorflow-estimator<2.5.0,>=2.4.0 in /usr/loc
Requirement already satisfied: flatbuffers~=1.12.0 in /usr/local/lib/python3.
Requirement already satisfied: keras-preprocessing~=1.1.2 in /usr/local/lib/p
```

```
Requirement already satisfied: opt-einsum~=3.3.0 in /usr/local/lib/python3.7/
Requirement already satisfied: termcolor~=1.1.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: typing-extensions~=3.7.4 in /usr/local/lib/pyt
Requirement already satisfied: h5py~=2.10.0 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: astunparse~=1.6.3 in /usr/local/lib/python3.7/
Requirement already satisfied: gast==0.3.3 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: tensorboard~=2.4 in /usr/local/lib/python3.7/d
Requirement already satisfied: wheel~=0.35 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: wrapt~=1.12.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: google-pasta~=0.2 in /usr/local/lib/python3.7/
Requirement already satisfied: grpcio~=1.32.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: attrs>=18.1.0 in /usr/local/lib/python3.7/dist
Requirement already satisfied: dm-tree in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: dill in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: tensorflow-metadata in /usr/local/lib/python3.
Requirement already satisfied: importlib-resources; python_version < "3.9" in
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: promise in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: httplib2<1dev,>=0.15.0 in /usr/local/lib/pytho
Requirement already satisfied: google-api-core<2dev,>=1.21.0 in /usr/local/li
Requirement already satisfied: uritemplate<4dev,>=3.0.0 in /usr/local/lib/pyt
Requirement already satisfied: google-auth-httplib2>=0.0.3 in /usr/local/lib/
Requirement already satisfied: google-auth>=1.16.0 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /u
Requirement already satisfied: pyasn1>=0.1.7 in /usr/local/lib/python3.7/dist
Requirement already satisfied: rsa>=3.1.4 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pyasn1-modules>=0.0.5 in /usr/local/lib/python
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/
Requirement already satisfied: text-unidecode>=1.3 in /usr/local/lib/python3.
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/di
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/li
Requirement already satisfied: googleapis-common-protos<2,>=1.52.0 in /usr/lo
Requirement already satisfied: zipp>=0.4; python_version < "3.8" in /usr/loca
Requirement already satisfied: packaging>=14.3 in /usr/local/lib/python3.7/di
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/pytho
Requirement already satisfied: importlib-metadata; python_version < "3.8" in
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/pyt
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/di
Building wheels for collected packages: seqeval, py-cpuinfo
  Building wheel for seqeval (setup.py) ... done
  Created wheel for seqeval: filename=seqeval-1.2.2-cp37-none-any.whl size=16
  Stored in directory: /root/.cache/pip/wheels/52/df/1b/45d75646c37428f7e6262
  Building wheel for py-cpuinfo (setup.py) ... done
  Created wheel for py-cpuinfo: filename=py_cpuinfo-8.0.0-cp37-none-any.whl s
```

```
! pip install lvis
```

```
Collecting lvis
  Downloading https://files.pythonhosted.org/packages/72/b6/1992240ab48310b53
Requirement already satisfied: opencv-python>=4.1.0.25 in /usr/local/lib/pyth
Requirement already satisfied: matplotlib>=3.1.1 in /usr/local/lib/python3.7/
Requirement already satisfied: kiwisolver>=1.1.0 in /usr/local/lib/python3.7/
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.7/dist-p
```

```
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/pytho
Requirement already satisfied: Cython>=0.29.12 in /usr/local/lib/python3.7/di
Requirement already satisfied: numpy>=1.18.2 in /usr/local/lib/python3.7/dist
Requirement already satisfied: pyparsing>=2.4.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: cycler>=0.10.0 in /usr/local/lib/python3.7/dis
Installing collected packages: lvis
Successfully installed lvis-0.5.3
```

```
%cd

!git clone --quiet https://github.com/tensorflow/models.git

!apt-get install -qq protobuf-compiler python-tk

!pip install -q Cython contextlib2 pillow lxml matplotlib PyDrive

!pip install -q pycocotools

%cd ~/models/research
!protoc object_detection/protos/*.proto --python_out=.

import os
os.environ['PYTHONPATH'] += ':/root/models/research/:/root/models/research/slim/'

!python object_detection/builders/model_builder_test.py
```

```
/root
/root/models/research
2021-04-14 22:32:40.835733: I tensorflow/stream_executor/platform/default/dso
```

Import Libraries

```
import glob
import io
import os
from collections import namedtuple
from xml.etree import ElementTree as tree

import pandas as pd
import tensorflow.compat.v1 as tf
from PIL import Image
from object_detection.utils import dataset_util
```

Create Helper Functions

```
def encode_class(row_label):
    class_mapping = {'apple': 1, 'orange': 2, 'banana': 3}

    return class_mapping.get(row_label, None)
```

```
def split(df, group):
    Data = namedtuple('data', ['filename', 'object'])
    groups = df.groupby(group)
    return [Data(filename, groups.get_group(x))
            for filename, x
            in zip(groups.groups.keys(), groups.groups)]


def create_tf_example(group, path):
    groups_path = os.path.join(path, f'{group.filename}')
    with tf.gfile.GFile(groups_path, 'rb') as f:
        encoded_jpg = f.read()

    image = Image.open(io.BytesIO(encoded_jpg))
    width, height = image.size

    filename = group.filename.encode('utf8')
    image_format = b'jpg'

    xmins = []
    xmaxs = []
    ymins = []
    ymaxs = []
    classes_text = []
    classes = []

    for index, row in group.object.iterrows():
        xmins.append(row['xmin'] / width)
        xmaxs.append(row['xmax'] / width)
        ymins.append(row['ymin'] / height)
        ymaxs.append(row['ymax'] / height)
        classes_text.append(row['class'].encode('utf8'))
        classes.append(encode_class(row['class']))

    features = tf.train.Features(feature={
        'image/height':
            dataset_util.int64_feature(height),
        'image/width':
            dataset_util.int64_feature(width),
        'image/filename':
            dataset_util.bytes_feature(filename),
        'image/source_id':
            dataset_util.bytes_feature(filename),
        'image/encoded':
            dataset_util.bytes_feature(encoded_jpg),
        'image/format':
            dataset_util.bytes_feature(image_format),
        'image/object/bbox/xmin':
            dataset_util.float_list_feature(xmins),
        'image/object/bbox/xmax':
            dataset_util.float_list_feature(xmaxs),
        'image/object/bbox/ymin':
            dataset_util.float_list_feature(ymins),
```

```
        'image/object/bbox/ymax':
            dataset_util.float_list_feature(ymaxs),
        'image/object/class/text':
            dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label':
            dataset_util.int64_list_feature(classes)
    })

    return tf.train.Example(features=features)


def bboxes_to_csv(path):
    xml_list = []

    bboxes_pattern = os.path.sep.join([path, '*.xml'])
    for xml_file in glob.glob(bboxes_pattern):
        t = tree.parse(xml_file)
        root = t.getroot()

        for member in root.findall('object'):
            value = (root.find('filename').text,
                     int(root.find('size')[0].text),
                     int(root.find('size')[1].text),
                     member[0].text,
                     int(member[4][0].text),
                     int(member[4][1].text),
                     int(member[4][2].text),
                     int(member[4][3].text))
            xml_list.append(value)

    column_names = ['filename', 'width', 'height', 'class',
                    'xmin', 'ymin', 'xmax', 'ymax']
    df = pd.DataFrame(xml_list, columns=column_names)
    return df
```

Add Dataset - Fruits

```
%cd /content
```

```
/content
```

```
! mkdir fruits
```

```
! ls
```

```
After_0.jpg   cat.jpg                   detections_orange_93.jpg  fruits.zip
banana.jpg    detections_apple_87.jpg   dog_example.jpg           me_dogs.jpg
Before_0.jpg  detections_banana_81.jpg  dog.jpg                   resources
cafe.jpg      detections_banana_88.jpg  drive                     sample_data
cars.jpg      detections_banana_91.jpg  elephants.jpg             test_images
catdog.jpg    detections_mixed_22.jpg   fruits
```

```
! mv fruits.zip ./fruits
```

```
! ls fruits
```

```
fruits.zip
```

```
%cd fruits
! unzip fruits.zip
! ls /content/fruits
```

```
/content/fruits
Archive:  fruits.zip
  inflating: test_zip/test/apple_77.jpg
  inflating: test_zip/test/apple_77.xml
  inflating: test_zip/test/apple_78.jpg
  inflating: test_zip/test/apple_78.xml
  inflating: test_zip/test/apple_79.jpg
  inflating: test_zip/test/apple_79.xml
  inflating: test_zip/test/apple_80.jpg
  inflating: test_zip/test/apple_80.xml
  inflating: test_zip/test/apple_81.jpg
  inflating: test_zip/test/apple_81.xml
  inflating: test_zip/test/apple_82.jpg
  inflating: test_zip/test/apple_82.xml
  inflating: test_zip/test/apple_83.jpg
  inflating: test_zip/test/apple_83.xml
  inflating: test_zip/test/apple_84.jpg
  inflating: test_zip/test/apple_84.xml
  inflating: test_zip/test/apple_85.jpg
  inflating: test_zip/test/apple_85.xml
  inflating: test_zip/test/apple_86.jpg
  inflating: test_zip/test/apple_86.xml
  inflating: test_zip/test/apple_87.jpg
  inflating: test_zip/test/apple_87.xml
  inflating: test_zip/test/apple_88.jpg
  inflating: test_zip/test/apple_88.xml
  inflating: test_zip/test/apple_89.jpg
  inflating: test_zip/test/apple_89.xml
  inflating: test_zip/test/apple_90.jpg
  inflating: test_zip/test/apple_90.xml
  inflating: test_zip/test/apple_91.jpg
  inflating: test_zip/test/apple_91.xml
  inflating: test_zip/test/apple_92.jpg
  inflating: test_zip/test/apple_92.xml
  inflating: test_zip/test/apple_93.jpg
  inflating: test_zip/test/apple_93.xml
  inflating: test_zip/test/apple_94.jpg
  inflating: test_zip/test/apple_94.xml
  inflating: test_zip/test/apple_95.jpg
  inflating: test_zip/test/apple_95.xml
  inflating: test_zip/test/banana_77.jpg
  inflating: test_zip/test/banana_77.xml
  inflating: test_zip/test/banana_78.jpg
  inflating: test_zip/test/banana_78.xml
  inflating: test_zip/test/banana_79.jpg
  inflating: test_zip/test/banana_79.xml
```

```
  inflating: test_zip/test/banana_80.jpg
  inflating: test_zip/test/banana_80.xml
  inflating: test_zip/test/banana_81.jpg
  inflating: test_zip/test/banana_81.xml
  inflating: test_zip/test/banana_82.jpg
  inflating: test_zip/test/banana_82.xml
  inflating: test_zip/test/banana_83.jpg
  inflating: test_zip/test/banana_83.xml
  inflating: test_zip/test/banana_84.jpg
  inflating: test_zip/test/banana_84.xml
  inflating: test_zip/test/banana_85.jpg
  inflating: test_zip/test/banana_85.xml
```

```
#base = 'fruits'
base = '/content/fruits'
for subset in ['test', 'train']:
    folder = os.path.sep.join([base, f'{subset}_zip', subset])
    print(folder)

    labels_path = os.path.sep.join([base,
                                    f'{subset}_labels.csv'])
    print(labels_path)

    bboxes_df = bboxes_to_csv(folder)
    bboxes_df.to_csv(labels_path, index=None)

    writer = (tf.python_io.
              TFRecordWriter(f'/content/resources/{subset}.record'))
    examples = pd.read_csv(f'/content/fruits/{subset}_labels.csv')
    grouped = split(examples, 'filename')

    path = os.path.join(f'/content/fruits/{subset}_zip/{subset}')
    for group in grouped:
        tf_example = create_tf_example(group, path)
        writer.write(tf_example.SerializeToString())

    writer.close()
```

```
/content/fruits/test_zip/test
/content/fruits/test_labels.csv
/content/fruits/train_zip/train
/content/fruits/train_labels.csv
```

```
%cd /content
```

```
/content
```

```
! wget http://download.tensorflow.org/models/object_detection/tf2/20200711/efficie
```

```
--2021-04-14 22:34:13--  http://download.tensorflow.org/models/object_detecti
Resolving download.tensorflow.org (download.tensorflow.org)... 142.250.73.208
Connecting to download.tensorflow.org (download.tensorflow.org)|142.250.73.20
HTTP request sent, awaiting response... 200 OK
Length: 30736482 (29M) [application/x-tar]
Saving to: ‘efficientdet_d0_coco17_tpu-32.tar.gz’
```

```
efficientdet_d0_coc 100%[===================>]  29.31M  --.-KB/s    in 0.1s

2021-04-14 22:34:14 (235 MB/s) - ‘efficientdet_d0_coco17_tpu-32.tar.gz’ saved
```

```
! tar -xf efficientdet_d0_coco17_tpu-32.tar.gz
```

```
! ls /content/efficientdet_d0_coco17_tpu-32/checkpoint
```

```
checkpoint  ckpt-0.data-00000-of-00001  ckpt-0.index
```

```
%cd /root/models/research/object_detection
```

```
/root/models/research/object_detection
```

```
! mkdir /content/training
```

```
! mkdir /content/checkpoint
```

Check GPU card

```
! nvidia-smi
```

```
Wed Apr 14 22:35:05 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.67       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   40C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

VGG training

```
#%%time
```

```
# Tesla K80 - VGG
# num_train_steps=10 -> Wall time: 1min 53s - NOTE: BAD, not enough for successful
# num_train_steps=100 -> Wall time: 3min 35s - NOTE: BAD, not enough for successfu
# num_train_steps=1000 -> Wall time: 20min 42s - NOTE: OK!!! ... enough for succes
# num_train_steps=10000 -> ... let's try yourself :)

#! python model_main_tf2.py --pipeline_config_path=/content/resources/ssd_efficien
```

```
2021-04-13 14:14:55.777306: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.727407: I tensorflow/compiler/jit/xla_cpu_device.cc:41]
2021-04-13 14:14:58.728305: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.734327: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:58.734993: I tensorflow/core/common_runtime/gpu/gpu_device
pciBusID: 0000:00:04.0 name: Tesla K80 computeCapability: 3.7
coreClock: 0.8235GHz coreCount: 13 deviceMemorySize: 11.17GiB deviceMemoryBa
2021-04-13 14:14:58.735045: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.737978: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.738081: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.740337: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.740764: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.742963: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.743811: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.744105: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.744272: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:58.744917: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:58.745505: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-13 14:14:58.745962: I tensorflow/compiler/jit/xla_gpu_device.cc:99]
2021-04-13 14:14:58.746110: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:58.746711: I tensorflow/core/common_runtime/gpu/gpu_device
pciBusID: 0000:00:04.0 name: Tesla K80 computeCapability: 3.7
coreClock: 0.8235GHz coreCount: 13 deviceMemorySize: 11.17GiB deviceMemoryBa
2021-04-13 14:14:58.746747: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.746820: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.746882: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.746925: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.746984: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.747041: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.747094: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.747151: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:58.747287: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:58.747940: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:58.748553: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-13 14:14:58.748609: I tensorflow/stream_executor/platform/default/ds
2021-04-13 14:14:59.197967: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-13 14:14:59.198042: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-13 14:14:59.198075: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-13 14:14:59.198301: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:59.198939: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:59.199677: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-13 14:14:59.200375: W tensorflow/core/common_runtime/gpu/gpu_bfc_al
2021-04-13 14:14:59.200434: I tensorflow/core/common_runtime/gpu/gpu_device
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica
I0413 14:14:59.202437 140614863771520 mirrored_strategy.py:350] Using Mirro
INFO:tensorflow:Maybe overwriting train_steps: 1000
I0413 14:14:59.207508 140614863771520 config_util.py:552] Maybe overwriting
INFO:tensorflow:Maybe overwriting use_bfloat16: False
I0413 14:14:59.207687 140614863771520 config_util.py:552] Maybe overwriting
I0413 14:14:59.221871 140614863771520 ssd_efficientnet_bifpn_feature_extract
I0413 14:14:59.222006 140614863771520 ssd_efficientnet_bifpn_feature_extract
```

```
I0413 14:14:59.222135 140614863771520 ssd_efficientnet_bifpn_feature_extract
I0413 14:14:59.229565 140614863771520 efficientnet_model.py:147] round_filte
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then
I0413 14:14:59.251369 140614863771520 cross_device_ops.py:565] Reduce to /jo
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then
I0413 14:14:59.252799 140614863771520 cross_device_ops.py:565] Reduce to /jo
```

```
! cp /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/resou
```

```
! ls -all /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/
```

```
total 7
-rw------- 1 root root   84 Feb 11 12:28 anchors.json
-rw------- 1 root root  625 Feb 11 12:28 coco_labels.txt
-rw------- 1 root root  112 Feb 11 12:28 label_map.txt
-rw------- 1 root root 4758 Apr 13 12:18 ssd_efficientdet_d0_512x512_coco17_t
```

```
! ls -all /content/resources/
```

```
total 30364
drwx------ 2 root root     4096 Apr 13 12:03 .
drwxr-xr-x 1 root root     4096 Apr 13 12:07 ..
-rw------- 1 root root       84 Apr 13 11:42 anchors.json
-rw------- 1 root root      625 Apr 13 12:07 coco_labels.txt
-rw------- 1 root root      112 Apr 13 12:07 label_map.txt
-rw------- 1 root root     4728 Apr 13 12:07 ssd_efficientdet_d0_512x512_coco
-rw-r--r-- 1 root root  7081578 Apr 13 11:49 test.record
-rw-r--r-- 1 root root 23981776 Apr 13 11:49 train.record
```

```
!pwd
```

```
/root/models/research/object_detection
```

```
! ls -all /content/checkpoint
```

```
total 8
drwxr-xr-x 2 root root 4096 Apr 13 12:07 .
drwxr-xr-x 1 root root 4096 Apr 13 12:07 ..
```

MobileNetV2 training

```
%%time
# Tesla T4 - MobileNetV2
# num_train_steps=1000 -> Wall time: 12 min 39s - NOTE: OK!!! ... enough for succes
# num_train_steps=10000 -> ... let's try yourself :)

! python model_main_tf2.py --pipeline_config_path=/content/resources/ssd_efficient
```

```
2021-04-14 22:36:57.973294: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.360838: I tensorflow/compiler/jit/xla_cpu_device.cc:41]
2021-04-14 22:37:00.361698: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.399980: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.400575: I tensorflow/core/common_runtime/gpu/gpu_device
pciBusID: 0000:00:04.0 name: Tesla T4 computeCapability: 7.5
coreClock: 1.59GHz coreCount: 40 deviceMemorySize: 14.75GiB deviceMemoryBand
2021-04-14 22:37:00.400619: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.415790: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.415939: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.423211: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.423845: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.462855: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.463560: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.463975: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.464118: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.464743: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.465286: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:37:00.465757: I tensorflow/compiler/jit/xla_gpu_device.cc:99]
2021-04-14 22:37:00.465889: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.466455: I tensorflow/core/common_runtime/gpu/gpu_device
pciBusID: 0000:00:04.0 name: Tesla T4 computeCapability: 7.5
coreClock: 1.59GHz coreCount: 40 deviceMemorySize: 14.75GiB deviceMemoryBand
2021-04-14 22:37:00.466483: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466527: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466548: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466569: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466592: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466615: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466634: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466653: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.466721: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.467337: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.467843: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:37:00.467894: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:37:00.974493: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:37:00.974544: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:37:00.974557: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:37:00.974753: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.975418: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.975982: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:37:00.976534: W tensorflow/core/common_runtime/gpu/gpu_bfc_al
2021-04-14 22:37:00.976582: I tensorflow/core/common_runtime/gpu/gpu_device
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica
I0414 22:37:00.978274 139786865235840 mirrored_strategy.py:350] Using Mirro
INFO:tensorflow:Maybe overwriting train_steps: 1000
I0414 22:37:00.983815 139786865235840 config_util.py:552] Maybe overwriting
INFO:tensorflow:Maybe overwriting use_bfloat16: False
I0414 22:37:00.983990 139786865235840 config_util.py:552] Maybe overwriting
I0414 22:37:00.997574 139786865235840 ssd_efficientnet_bifpn_feature_extract
I0414 22:37:00.997685 139786865235840 ssd_efficientnet_bifpn_feature_extract
I0414 22:37:00.997754 139786865235840 ssd_efficientnet_bifpn_feature_extract
I0414 22:37:01.004395 139786865235840 efficientnet_model.py:147] round_filte
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then
I0414 22:37:01.019604 139786865235840 cross_device_ops.py:565] Reduce to /jo
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then
I0414 22:37:01.025848 139786865235840 cross_device_ops.py:565] Reduce to /jo
```

Export Trained Model for Future Prediction (Inference)

Once the network has been fine-tuned, we must export it as a frozen graph in order to use it for inference.

For that matter, use `cd` command again to the `object_detection` folder in the TensorFlow Object Detection API.

```
%cd /root/models/research/object_detection
```

```
/root/models/research/object_detection
```

```
%%time

! python exporter_main_v2.py --trained_checkpoint_dir=/content/training/ --pipelin
```

```
2021-04-14 22:50:43.427411: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.541202: I tensorflow/compiler/jit/xla_cpu_device.cc:41]
2021-04-14 22:50:45.542016: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.575902: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:45.576482: I tensorflow/core/common_runtime/gpu/gpu_device
pciBusID: 0000:00:04.0 name: Tesla T4 computeCapability: 7.5
coreClock: 1.59GHz coreCount: 40 deviceMemorySize: 14.75GiB deviceMemoryBand
2021-04-14 22:50:45.576518: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.581578: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.581655: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.583439: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.583787: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.585556: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.586146: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.586329: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.586433: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:45.587033: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:45.587594: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:50:45.588000: I tensorflow/compiler/jit/xla_gpu_device.cc:99]
2021-04-14 22:50:45.588119: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:45.588656: I tensorflow/core/common_runtime/gpu/gpu_device
pciBusID: 0000:00:04.0 name: Tesla T4 computeCapability: 7.5
coreClock: 1.59GHz coreCount: 40 deviceMemorySize: 14.75GiB deviceMemoryBand
2021-04-14 22:50:45.588695: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588725: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588746: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588764: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588783: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588802: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588821: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588840: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:45.588918: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:45.589474: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:45.589998: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:50:45.590045: I tensorflow/stream_executor/platform/default/ds
2021-04-14 22:50:46.086597: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:50:46.086658: I tensorflow/core/common_runtime/gpu/gpu_device
```

```
2021-04-14 22:50:46.086673: I tensorflow/core/common_runtime/gpu/gpu_device
2021-04-14 22:50:46.086875: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:46.087491: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:46.088083: I tensorflow/stream_executor/cuda/cuda_gpu_execu
2021-04-14 22:50:46.088595: W tensorflow/core/common_runtime/gpu/gpu_bfc_al
2021-04-14 22:50:46.088645: I tensorflow/core/common_runtime/gpu/gpu_device
I0414 22:50:46.095656 140350676047744 ssd_efficientnet_bifpn_feature_extract
I0414 22:50:46.095843 140350676047744 ssd_efficientnet_bifpn_feature_extract
I0414 22:50:46.095929 140350676047744 ssd_efficientnet_bifpn_feature_extract
I0414 22:50:46.101149 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.119569 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.119671 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.173674 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.173789 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.307396 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.307505 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.445173 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.445297 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.643217 140350676047744 efficientnet_model.py:147] round_filte
I0414 22:50:46.643334 140350676047744 efficientnet_model.py:147] round_filte
```

```
import glob
import random
from io import BytesIO

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from PIL import Image
from object_detection.utils import ops
from object_detection.utils import visualization_utils as viz
from object_detection.utils.label_map_util import \
    create_category_index_from_labelmap
```

```
def load_image(path):
    image_data = tf.io.gfile.GFile(path, 'rb').read()
    image = Image.open(BytesIO(image_data))

    width, height = image.size
    shape = (height, width, 3)

    image = np.array(image.getdata())
    image = image.reshape(shape).astype('uint8')

    return image


def infer_image(net, image):
    image = np.asarray(image)
    input_tensor = tf.convert_to_tensor(image)
    input_tensor = input_tensor[tf.newaxis, ...]

    model = net.signatures['serving_default']
    result = model(input_tensor)
```

```
    num_detections = int(result.pop('num_detections'))
    result = {key: value[0, :num_detections].numpy()
              for key, value in result.items()}
    result['num_detections'] = num_detections

    result['detection_classes'] = \
        result['detection_classes'].astype('int64')

    if 'detection_masks' in result:
        detection_masks_reframed = \
            ops.reframe_box_masks_to_image_masks(
                result['detection_masks'],
                result['detection_boxes'],
                image.shape[0],
                image.shape[1])

        detection_masks_reframed = \
            tf.cast(detection_masks_reframed > 0.5, tf.uint8)

        result['detection_masks_reframed'] = \
            detection_masks_reframed.numpy()

    return result
```

```
%cd /content
```

```
/content
```

```
%%time

labels_path = 'resources/label_map.txt'
CATEGORY_IDX = \
    create_category_index_from_labelmap(labels_path,
                                        use_display_name=True)
model_path = 'resources/inference_graph/saved_model'
model = tf.saved_model.load(model_path)

test_images = list(glob.glob('fruits/test_zip/test/*.jpg'))
random.shuffle(test_images)
test_images = test_images[:3]
```

```
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_ar
```

```
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_EfficientDet-D0_layer_call_a
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
WARNING:absl:Importing a function (__inference_bifpn_layer_call_and_return_
```

MobileNetV2 Prediction

```
%%time
from tqdm import tqdm
from IPython.display import clear_output

for image_path in tqdm(test_images):
    image = load_image(image_path)
    result = infer_image(model, image)

    masks = result.get('detection_masks_reframed', None)
    viz.visualize_boxes_and_labels_on_image_array(
```

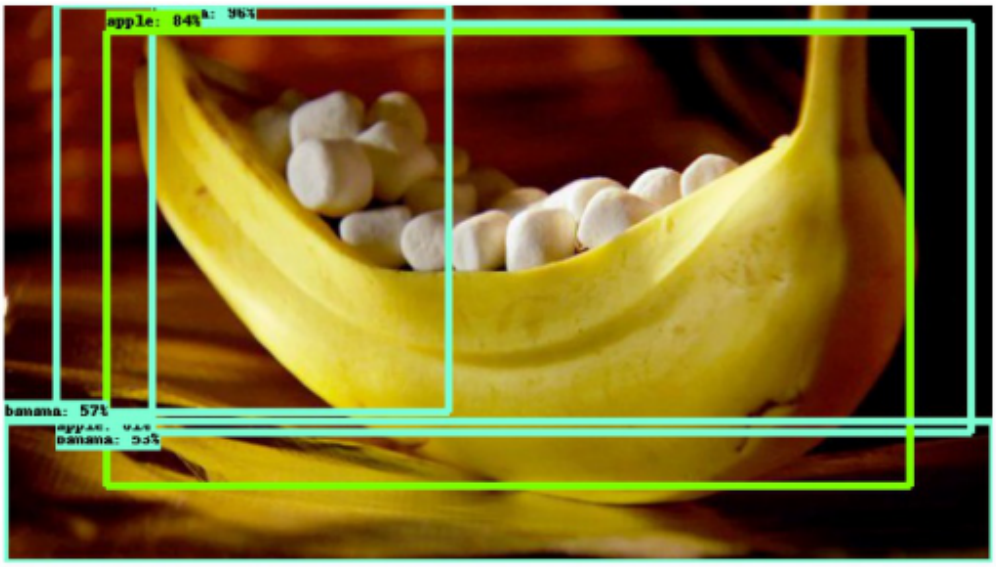
```
        image,
        result['detection_boxes'],
        result['detection_classes'],
        result['detection_scores'],
        CATEGORY_IDX,
        instance_masks=masks,
        use_normalized_coordinates=True,
        line_thickness=5)

    clear_output()
    plt.figure(figsize=(12, 8))
    plt.axis("off")
    plt.imshow(image);
    plt.savefig(f'detections_{image_path.split("/")[-1]}')
```

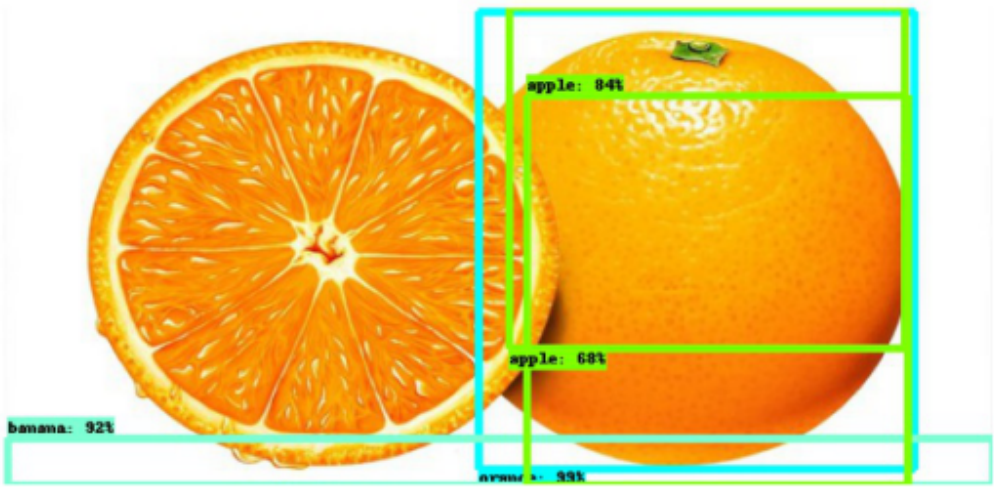
```
100%|██████████| 3/3 [00:05<00:00,  1.89s/it]CPU times: user 4.26 s, sys: 1.1
Wall time: 5.7 s
```

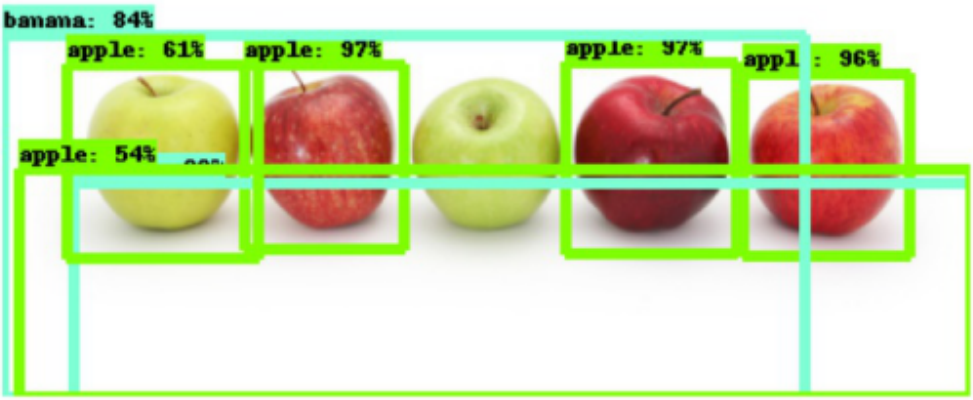
```
! ls -all detections*
```

```
-rw-r--r-- 1 root root  60237 Apr 14 22:53 detections_apple_86.jpg
-rw------- 1 root root  49142 Apr 14 18:45 detections_apple_87.jpg
-rw------- 1 root root  44233 Apr 14 18:45 detections_banana_81.jpg
-rw-r--r-- 1 root root  90256 Apr 14 22:53 detections_banana_83.jpg
-rw------- 1 root root  55359 Apr 14 18:45 detections_banana_88.jpg
-rw------- 1 root root 217628 Apr 14 18:45 detections_banana_91.jpg
-rw------- 1 root root 291011 Apr 14 18:45 detections_mixed_22.jpg
-rw-r--r-- 1 root root  88203 Apr 14 22:53 detections_orange_78.jpg
-rw------- 1 root root 219106 Apr 14 18:45 detections_orange_93.jpg
```

```
%matplotlib inline

import matplotlib.pyplot as plt
plt.rcParams.update({'figure.max_open_warning': 0})

plt.figure(figsize=(12,8))
img = plt.imread('/content/detections_banana_83.jpg')
plt.axis("off")
plt.imshow(img);
#plt.show()

plt.figure(figsize=(12,8))
img = plt.imread('/content/detections_orange_78.jpg')
plt.axis("off")
plt.imshow(img);
#plt.show()

plt.figure(figsize=(12,8))
img = plt.imread('/content/detections_apple_86.jpg')
plt.axis("off")
plt.imshow(img);
#plt.show()
```


apple: 84%
banana: 57%
banana: 53%

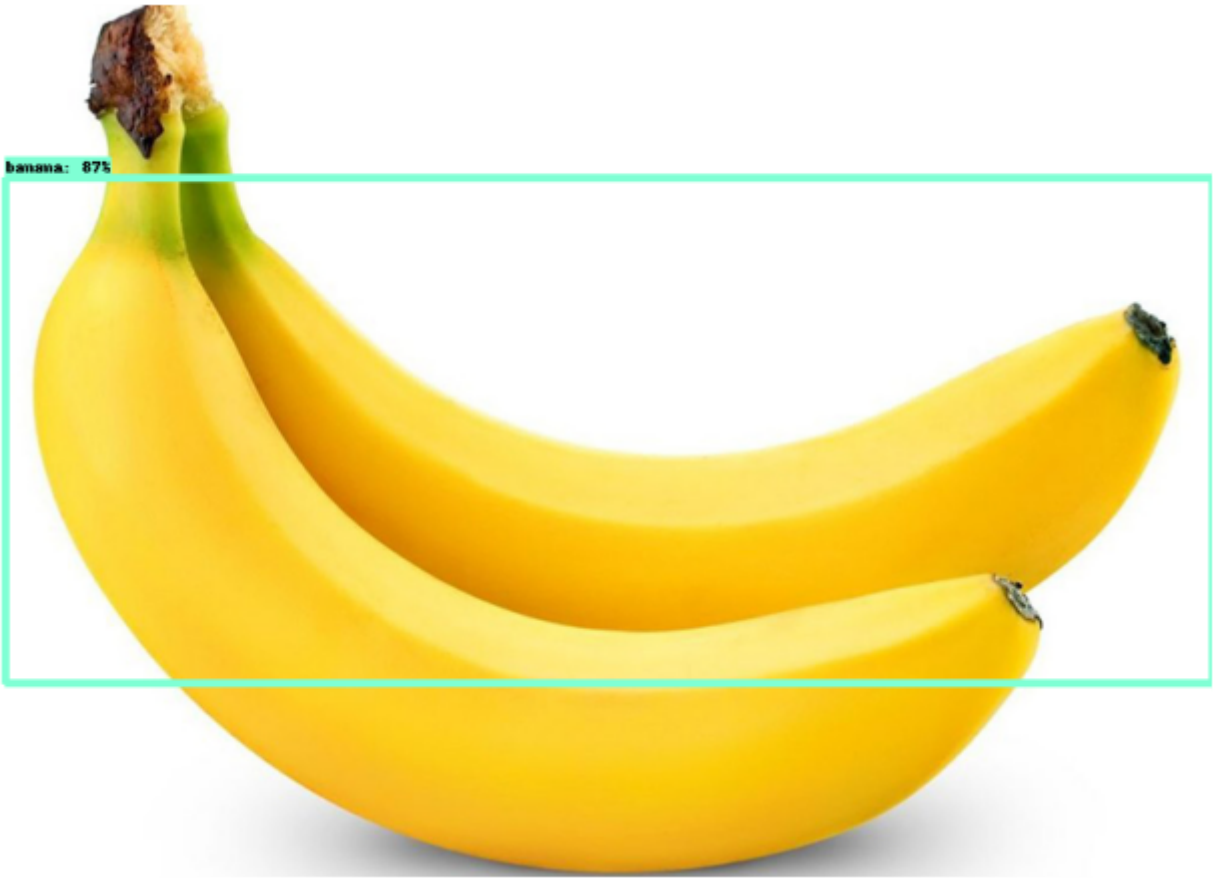

apple: 84%
apple: 68%
banana: 92%

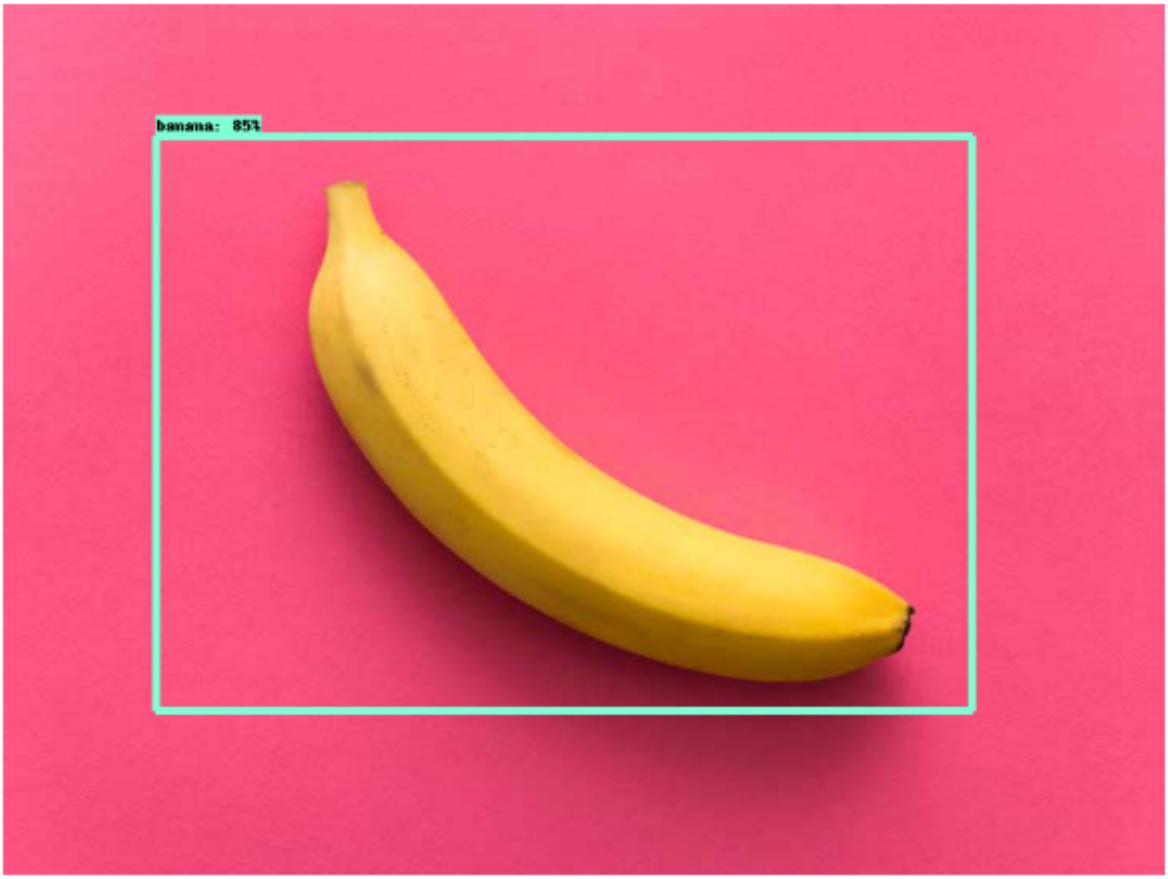


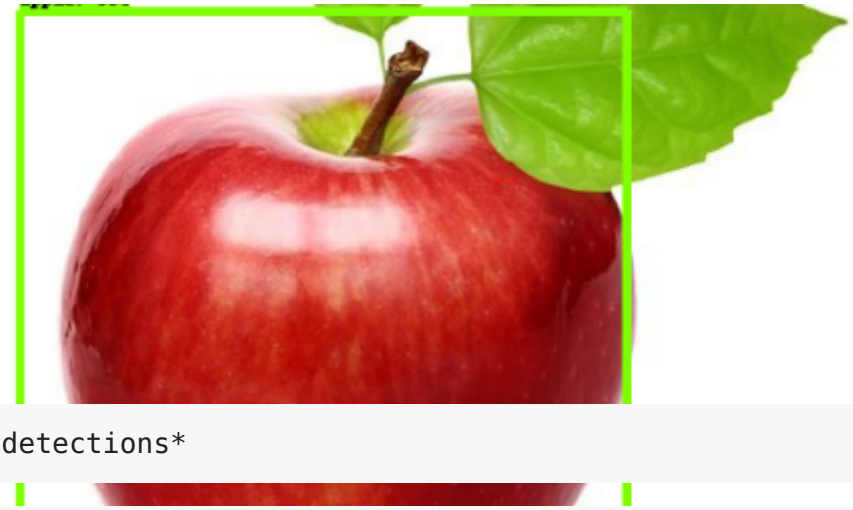
VGG Prediction

```
%%time
from tqdm import tqdm
from IPython.display import clear_output

for image_path in tqdm(test_images):
    image = load_image(image_path)
    result = infer_image(model, image)

    masks = result.get('detection_masks_reframed', None)
    viz.visualize_boxes_and_labels_on_image_array(
        image,
        result['detection_boxes'],
        result['detection_classes'],
        result['detection_scores'],
        CATEGORY_IDX,
        instance_masks=masks,
        use_normalized_coordinates=True,
        line_thickness=5)

    clear_output()
    plt.figure(figsize=(12, 8))
    plt.axis("off")
    plt.imshow(image);
    plt.savefig(f'detections_{image_path.split("/")[-1]}')
```

```
100%|██████████| 3/3 [00:02<00:00,  1.32it/s]
CPU times: user 2.2 s, sys: 79.2 ms, total: 2.28 s
Wall time: 2.28 s
```





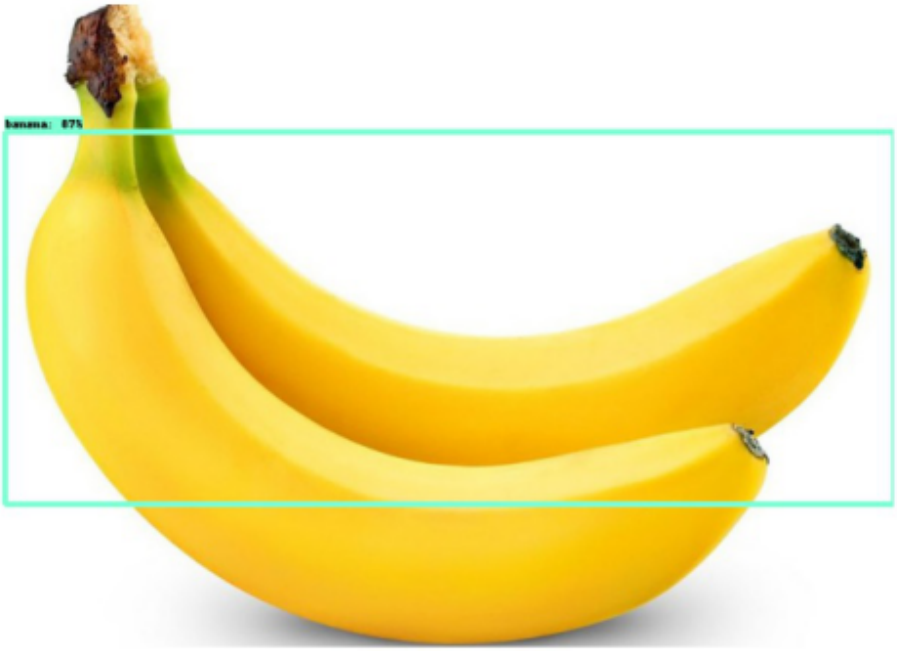


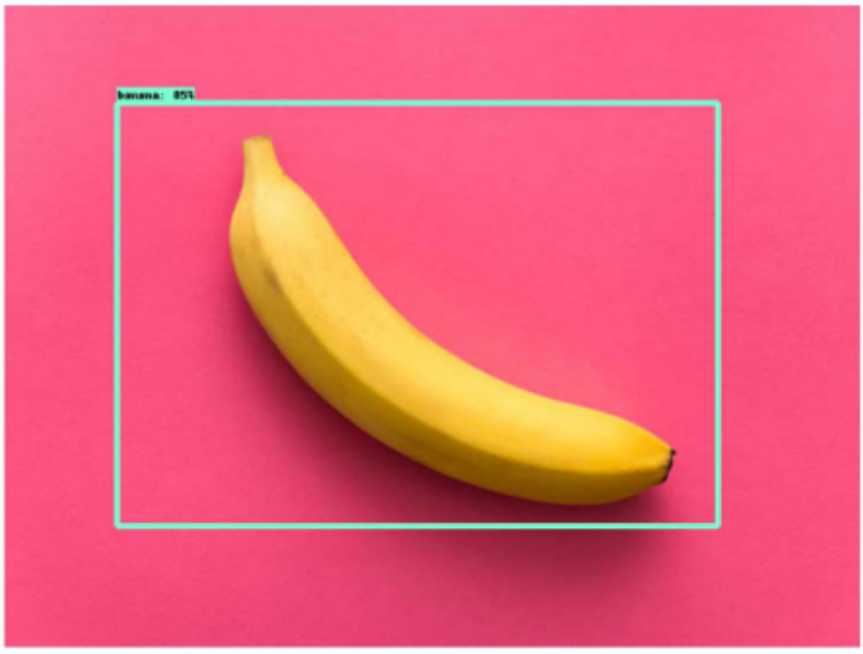
```
#! rm detections*
```

```
! ls -all detections*
```

```
-rw-r--r-- 1 root root  60237 Apr 14 22:53 detections_apple_86.jpg
-rw------- 1 root root  49142 Apr 14 18:45 detections_apple_87.jpg
-rw------- 1 root root  44233 Apr 14 18:45 detections_banana_81.jpg
-rw-r--r-- 1 root root  90256 Apr 14 22:53 detections_banana_83.jpg
-rw------- 1 root root  55359 Apr 14 18:45 detections_banana_88.jpg
-rw------- 1 root root 217628 Apr 14 18:45 detections_banana_91.jpg
-rw------- 1 root root 291011 Apr 14 18:45 detections_mixed_22.jpg
-rw-r--r-- 1 root root  88203 Apr 14 22:53 detections_orange_78.jpg
-rw------- 1 root root 219106 Apr 14 18:45 detections_orange_93.jpg
```

```
%matplotlib inline

import matplotlib.pyplot as plt
plt.rcParams.update({'figure.max_open_warning': 0})

plt.figure(figsize=(12,8))
img = plt.imread('/content/detections_banana_81.jpg')
plt.axis("off")
plt.imshow(img);
#plt.show()

plt.figure(figsize=(12,8))
img = plt.imread('/content/detections_banana_88.jpg')
plt.axis("off")
plt.imshow(img);
#plt.show()

plt.figure(figsize=(12,8))
img = plt.imread('/content/detections_apple_87.jpg')
plt.axis("off")
plt.imshow(img);
#plt.show()
```


banana: 87%


banana: 85%


apple: 99%

Summary

In this section, we discovered that training an object detector is a hard and challenging feat. The good news, however, is that we have the **TensorFlow Object Detection API** at our disposal to train a wide range of vanguardist networks.

- Because the TensorFlow Object Detection API is **an experimental tool**, it uses different conventions than regular TensorFlow, and therefore in order to use it, we need to perform a little bit of processing work on the input data to put it into a shape that the API understands. This is done by converting the labels in the Fruits for Object Detection dataset (originally in XML format) to CSV and then into serialized `tf.train.Example objects`.
- Then, to use the trained model, we exported it as an inference graph using the exporter_main_v2.py script and leveraged some of the visualization tools in the API to display the detections on the sample test images.
- Training is arguably the easiest part, entailing three major steps:
    - Creating a mapping from text labels to integers (Step 12)
    - Modifying the configuration file corresponding to the model to fine-tune it in all the relevant places (Step 13)
    - Running `model_main_tf2.py` file to train the network, passing it the proper parameters (Step 14)

This section provides you with a template you can tweak and adapt to train virtually any modern object detector (supported by the API) on any dataset of your choosing.

NOTE:

You can learn more about the TensorFlow Object Detection API here: https://github.com/tensorflow/models/tree/master/research/ object_detection

Also, read this great article to learn more about EfficientDet : https://towardsdatascience.com/a-thorough-breakdown-of- efficientdet-for-object-detection-dc6a15788b73

If you want to learn a great deal about the Pascal VOC format, then you must watch this video: https://www.youtube.com/watch?v=-f6TJpHcAeM

Part 5.Object Detection - Use Detectors from TFHub

TFHub is a collection of state-of-the-art (SOTA) models when it comes to object detection.

Using them to spot elements of interest in our images is a fairly straightforward task, especially considering they've been trained on the gigantic COCO dataset, which make them an excellent choice for out-of-the-box object detection.

```
import glob
from io import BytesIO

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
from PIL import Image
from object_detection.utils import visualization_utils as viz
from object_detection.utils.label_map_util import create_category_index_from_labelı
```

```
def load_image(path):
    image_data = tf.io.gfile.GFile(path, 'rb').read()
    image = Image.open(BytesIO(image_data))

    width, height = image.size
    shape = (1, height, width, 3)

    image = np.array(image.getdata())
    image = image.reshape(shape).astype('uint8')

    return image


def get_and_save_predictions(model, image_path):
    image = load_image(image_path)
    results = model(image)

    model_output = {k: v.numpy() for k, v in results.items()}

    boxes = model_output['detection_boxes'][0]
    classes = \
        model_output['detection_classes'][0].astype('int')
    scores = model_output['detection_scores'][0]

    clone = image.copy()
    viz.visualize_boxes_and_labels_on_image_array(
        image=clone[0],
        boxes=boxes,
        classes=classes,
        scores=scores,
```

```
        category_index=CATEGORY_IDX,
        use_normalized_coordinates=True,
        max_boxes_to_draw=200,
        min_score_thresh=0.30,
        agnostic_mode=False,
        line_thickness=5
    )

    plt.figure(figsize=(24, 32))
    plt.imshow(clone[0])

    plt.savefig(f'output/{image_path.split("/")[-1]}')
```

```
! cp -r /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/r
! cp -r /content/drive/MyDrive/COLAB_NN/Lecture_11_Object_Detection_Segmentation/t
! echo In test_images:
! ls /content/test_images
! echo In resources:
! ls /content/resources
```

```
In test_images:
cafe.jpg  dog.jpg        me_dogs.jpg      result_dog.png
cars.jpg  elephants.jpg  result_cars.png
In resources:
anchors.json     label_map.txt                                  test.record
coco_labels.txt  mscoco_label_map.pbtxt                         train.record
inference_graph  ssd_efficientdet_d0_512x512_coco17_tpu8.txt
```

Inception_ResNet_v2

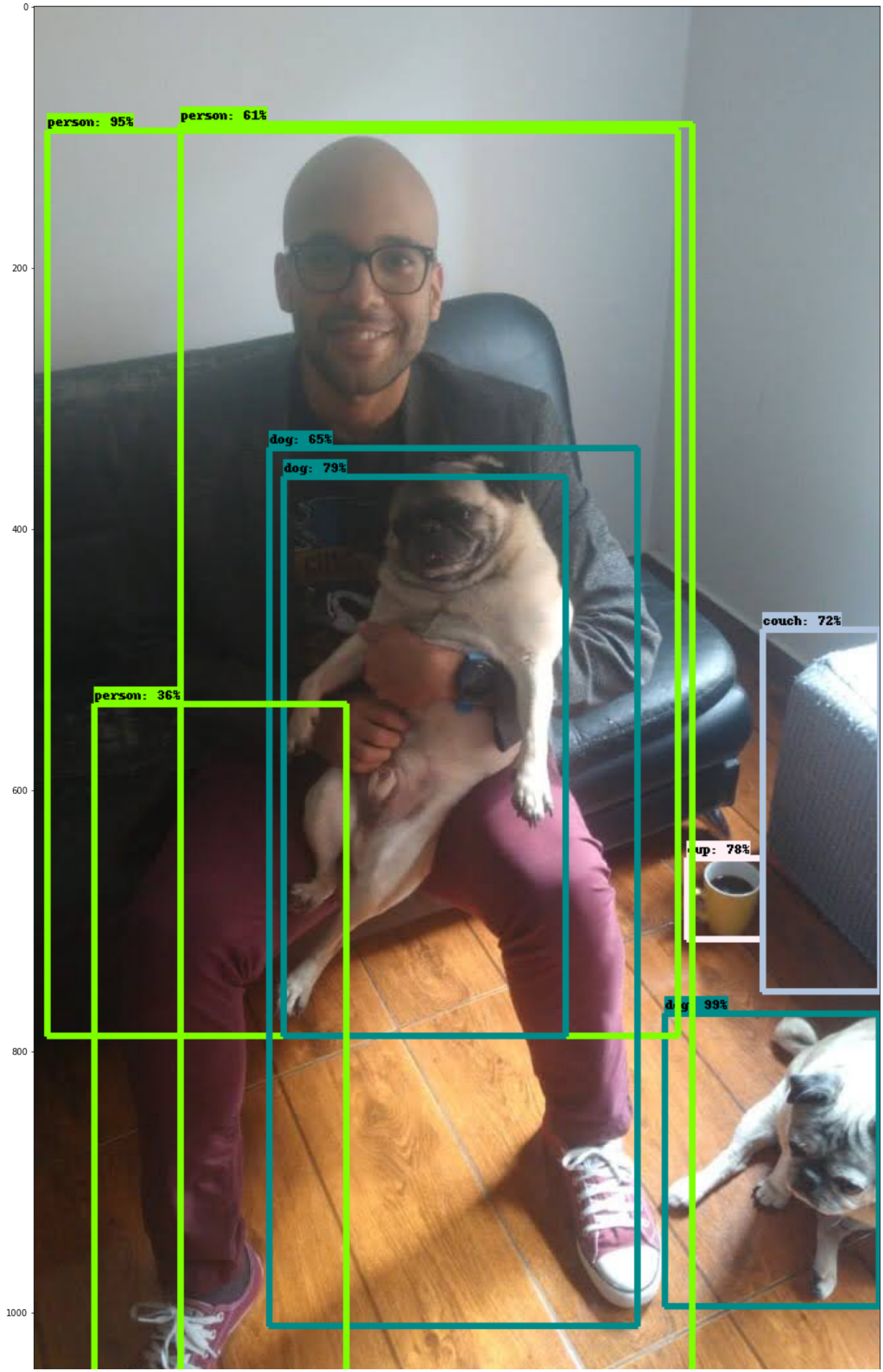
```
%%time

labels_path = 'resources/mscoco_label_map.pbtxt'
CATEGORY_IDX = create_category_index_from_labelmap(labels_path)

MODEL_PATH = ('https://tfhub.dev/tensorflow/faster_rcnn/inception_resnet_v2_1024x1
model = hub.load(MODEL_PATH)
```

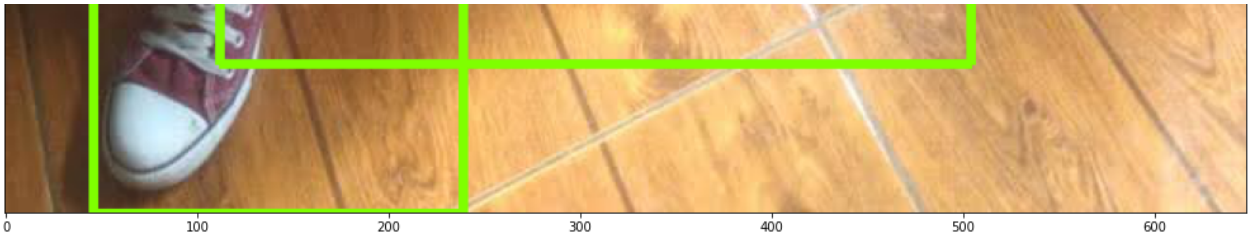
```
CPU times: user 28 s, sys: 2.43 s, total: 30.4 s
Wall time: 30.2 s
```

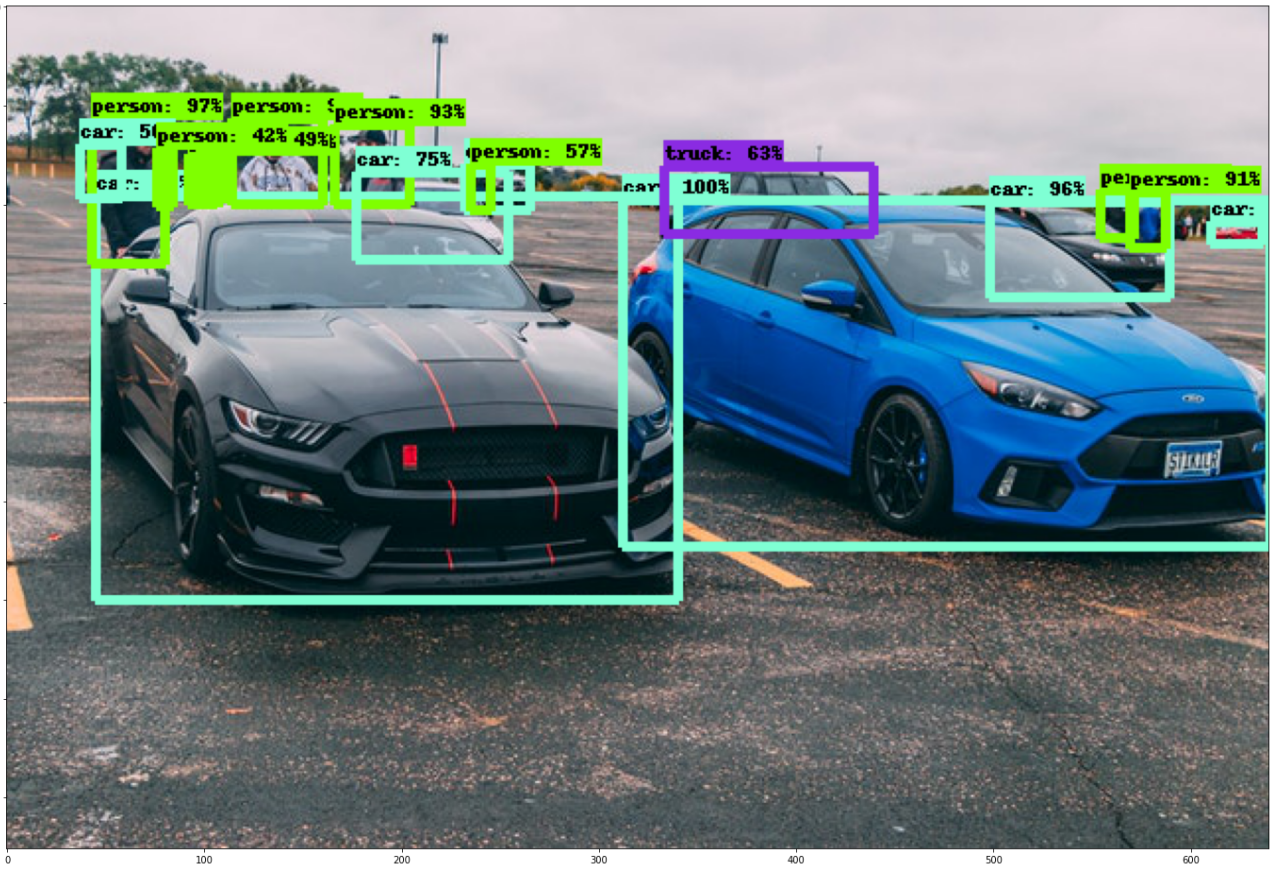
```
! mkdir output
```

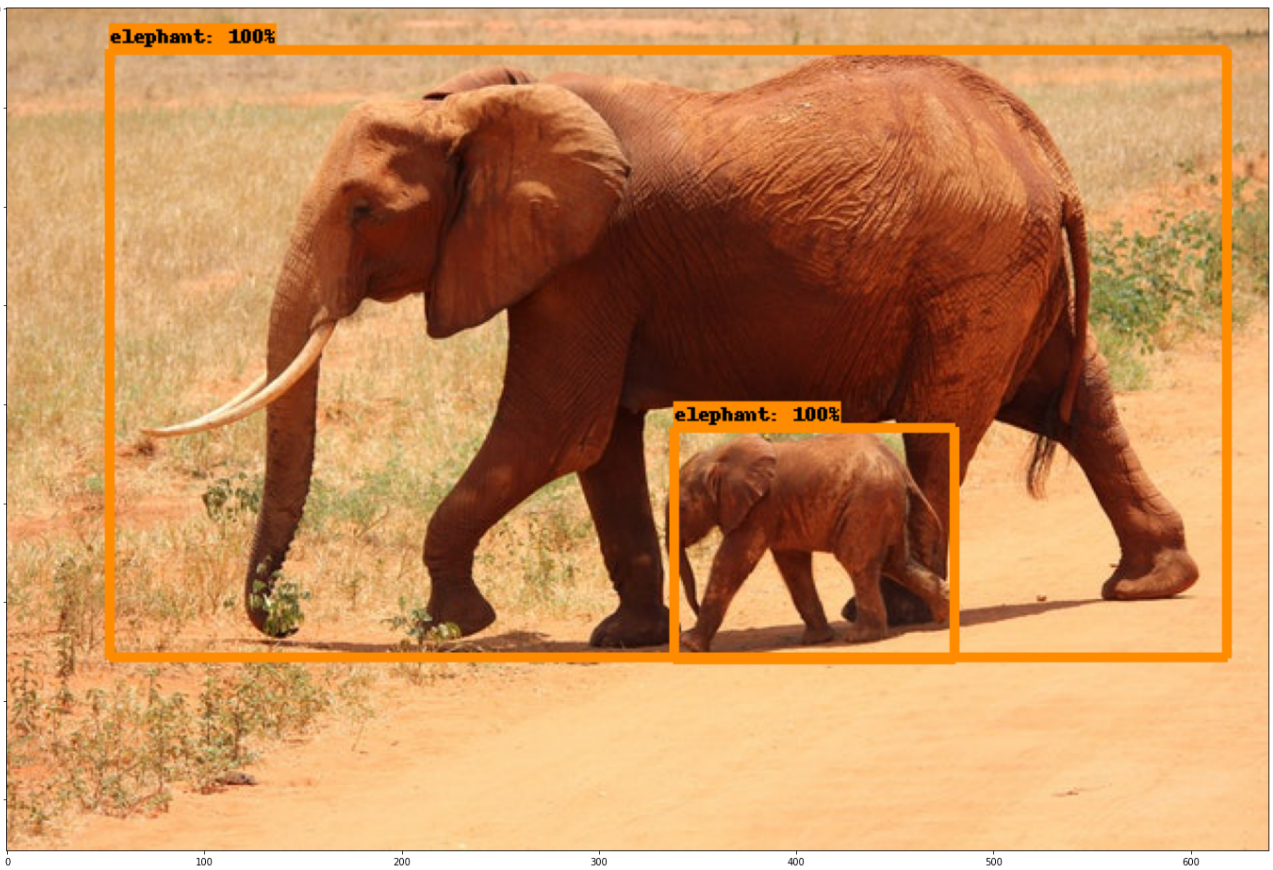
```
%%time

test_images_paths = glob.glob('test_images/*')
for image_path in test_images_paths:
    get_and_save_predictions(model, image_path)
```

```
CPU times: user 18.3 s, sys: 3.23 s, total: 21.5 s
Wall time: 23.2 s
```

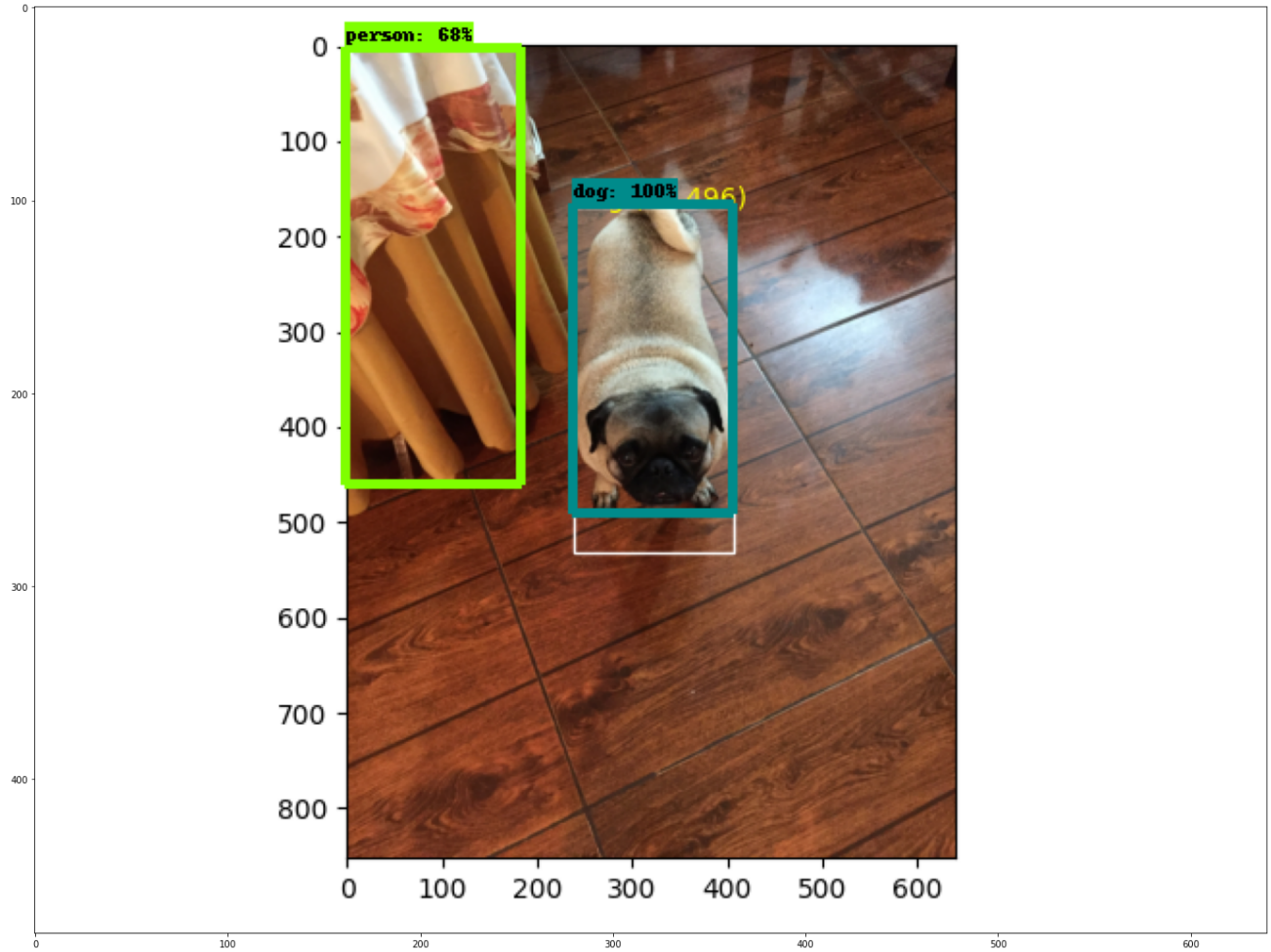


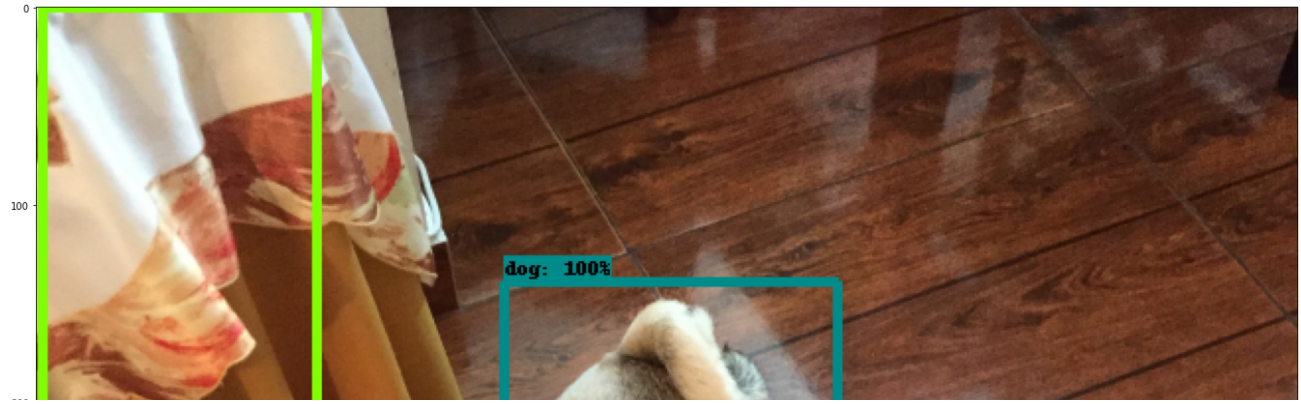


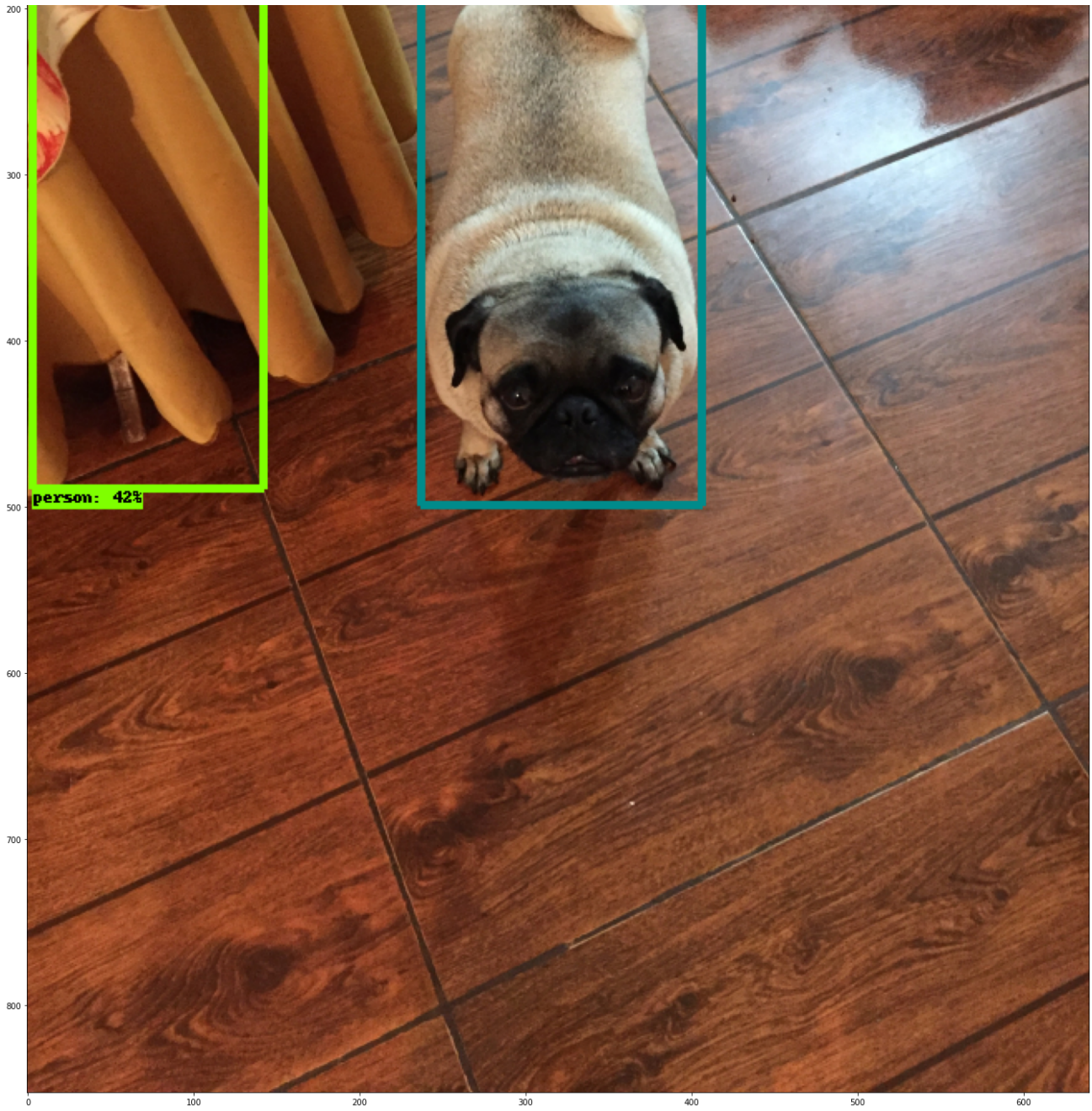

person: 97%
person: 93%
person: 42%
car: 75%
person: 57%
truck: 63%
100%
car: 96%
person: 91%

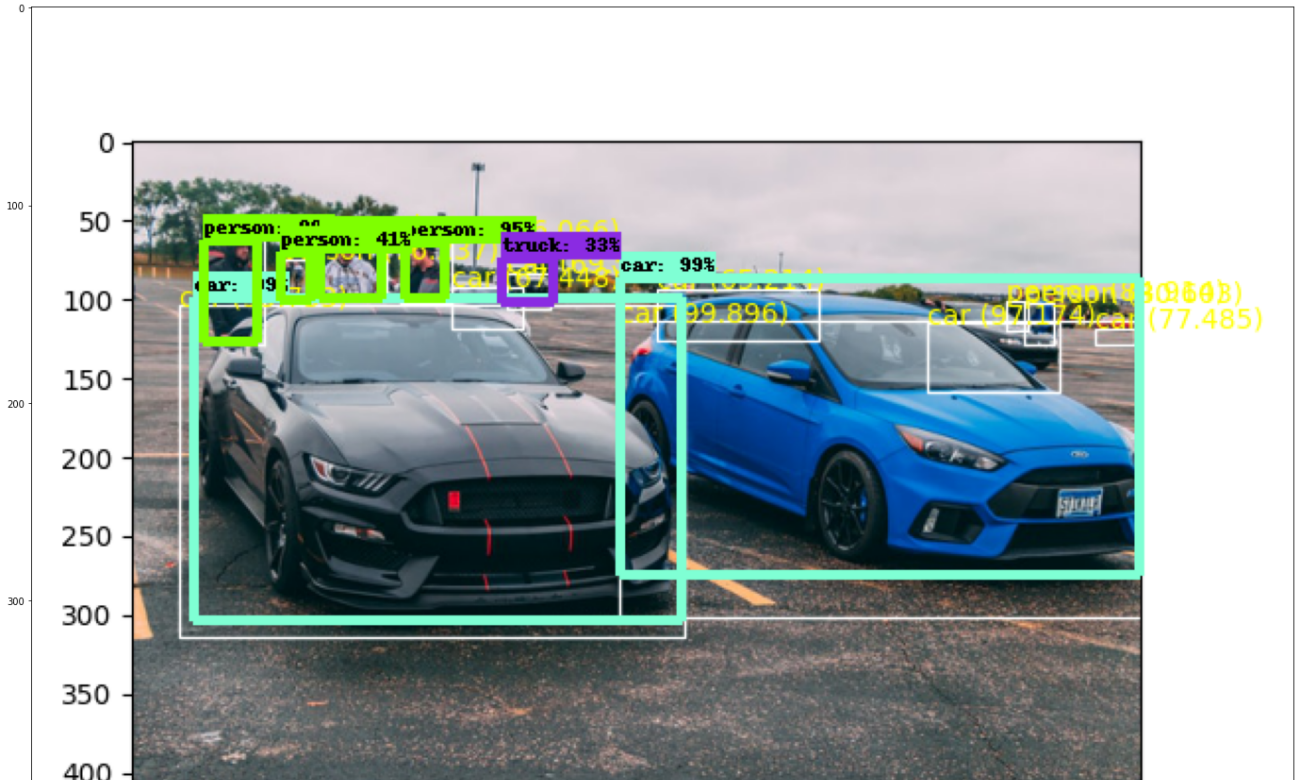

elephant: 100%
elephant: 100%

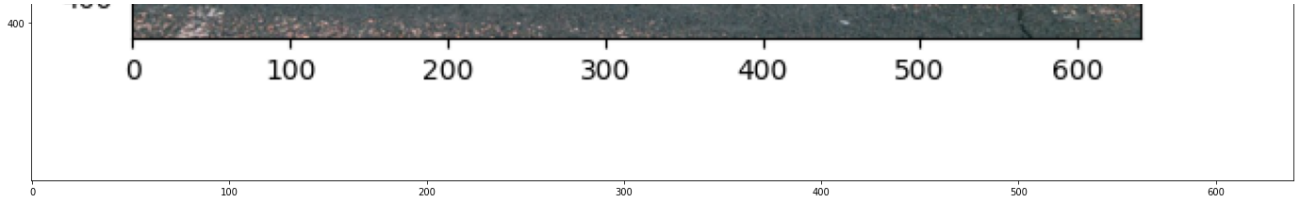

bottle: 44%
potted plant: 83%
bottle: 34%
chair: 56%
chair: 97%
person: 35%
bottle: 33%
dining table: 66%
chair: 99%
chair: 88%
chair: 64%


person: 68%
dog: 100%


dog: 100%


person: 42%


person: 41%
truck: 33%
car: 99%


400
0
100
200
300
400
500
600

SSD - MobileNet_v2

Helper functions for downloading images and for visualization.

Visualization code adapted from [TF object detection API] for the simplest required functionality.

```
# For drawing onto the image.
import numpy as np
from PIL import Image
from PIL import ImageColor
from PIL import ImageDraw
from PIL import ImageFont
from PIL import ImageOps
```

```
def display_image(image):
    fig = plt.figure(figsize=(20, 15))
    plt.rcParams.update({'font.size': 30})
    plt.grid(False)
    plt.imshow(image)


def download_and_resize_image(url, new_width=256, new_height=256,
                              display=False):
  _, filename = tempfile.mkstemp(suffix=".jpg")
  response = urlopen(url)
  image_data = response.read()
  image_data = BytesIO(image_data)
  pil_image = Image.open(image_data)
  pil_image = ImageOps.fit(pil_image, (new_width, new_height), Image.ANTIALIAS)
  pil_image_rgb = pil_image.convert("RGB")
  pil_image_rgb.save(filename, format="JPEG", quality=90)
  print("Image downloaded to %s." % filename)
  if display:
    display_image(pil_image)
  return filename


def draw_bounding_box_on_image(image,
                               ymin,
                               xmin,
                               ymax,
                               xmax,
                               color,
                               font,
                               thickness=4,
                               display_str_list=()):
  """Adds a bounding box to an image."""
```

```
  draw = ImageDraw.Draw(image)
  im_width, im_height = image.size
  (left, right, top, bottom) = (xmin * im_width, xmax * im_width,
                                ymin * im_height, ymax * im_height)
  draw.line([(left, top), (left, bottom), (right, bottom), (right, top),
             (left, top)],
            width=thickness,
            fill=color)

  # If the total height of the display strings added to the top of the bounding
  # box exceeds the top of the image, stack the strings below the bounding box
  # instead of above.
  display_str_heights = [font.getsize(ds)[1] for ds in display_str_list]
  # Each display_str has a top and bottom margin of 0.05x.
  total_display_str_height = (1 + 2 * 0.05) * sum(display_str_heights)

  if top > total_display_str_height:
    text_bottom = top
  else:
    text_bottom = bottom + total_display_str_height
  # Reverse list and print from bottom to top.
  for display_str in display_str_list[::-1]:
    text_width, text_height = font.getsize(display_str)
    margin = np.ceil(0.05 * text_height)
    draw.rectangle([(left, text_bottom - text_height - 2 * margin),
                    (left + text_width, text_bottom)],
                   fill=color)
    draw.text((left + margin, text_bottom - text_height - margin),
              display_str,
              fill="black",
              font=font)
    text_bottom -= text_height - 2 * margin


def draw_boxes(image, boxes, class_names, scores, max_boxes=10, min_score=0.1):
  """Overlay labeled boxes on an image with formatted scores and label names."""
  colors = list(ImageColor.colormap.values())

  try:
    font = ImageFont.truetype("/usr/share/fonts/truetype/liberation/LiberationSansI
                              25)
  except IOError:
    print("Font not found, using default font.")
    font = ImageFont.load_default()

  for i in range(min(boxes.shape[0], max_boxes)):
    if scores[i] >= min_score:
      ymin, xmin, ymax, xmax = tuple(boxes[i])
      display_str = "{}: {}%".format(class_names[i].decode("ascii"),
                                     int(100 * scores[i]))
      color = colors[hash(class_names[i]) % len(colors)]
      image_pil = Image.fromarray(np.uint8(image)).convert("RGB")
      draw_bounding_box_on_image(
          image_pil,
          ymin,
```

```
          xmin,
          ymax,
          xmax,
          color,
          font,
          display_str_list=[display_str])
      np.copyto(image, np.array(image_pil))
  return image
```

```
%%time

labels_path = 'resources/mscoco_label_map.pbtxt'
CATEGORY_IDX = create_category_index_from_labelmap(labels_path)

MODEL_PATH = ('https://tfhub.dev/google/openimages_v4/ssd/mobilenet_v2/1')
model = hub.load(MODEL_PATH).signatures['default']
```

```
INFO:tensorflow:Saver not created because there are no variables in the graph
INFO:tensorflow:Saver not created because there are no variables in the graph
CPU times: user 16.8 s, sys: 983 ms, total: 17.8 s
Wall time: 17.5 s
```

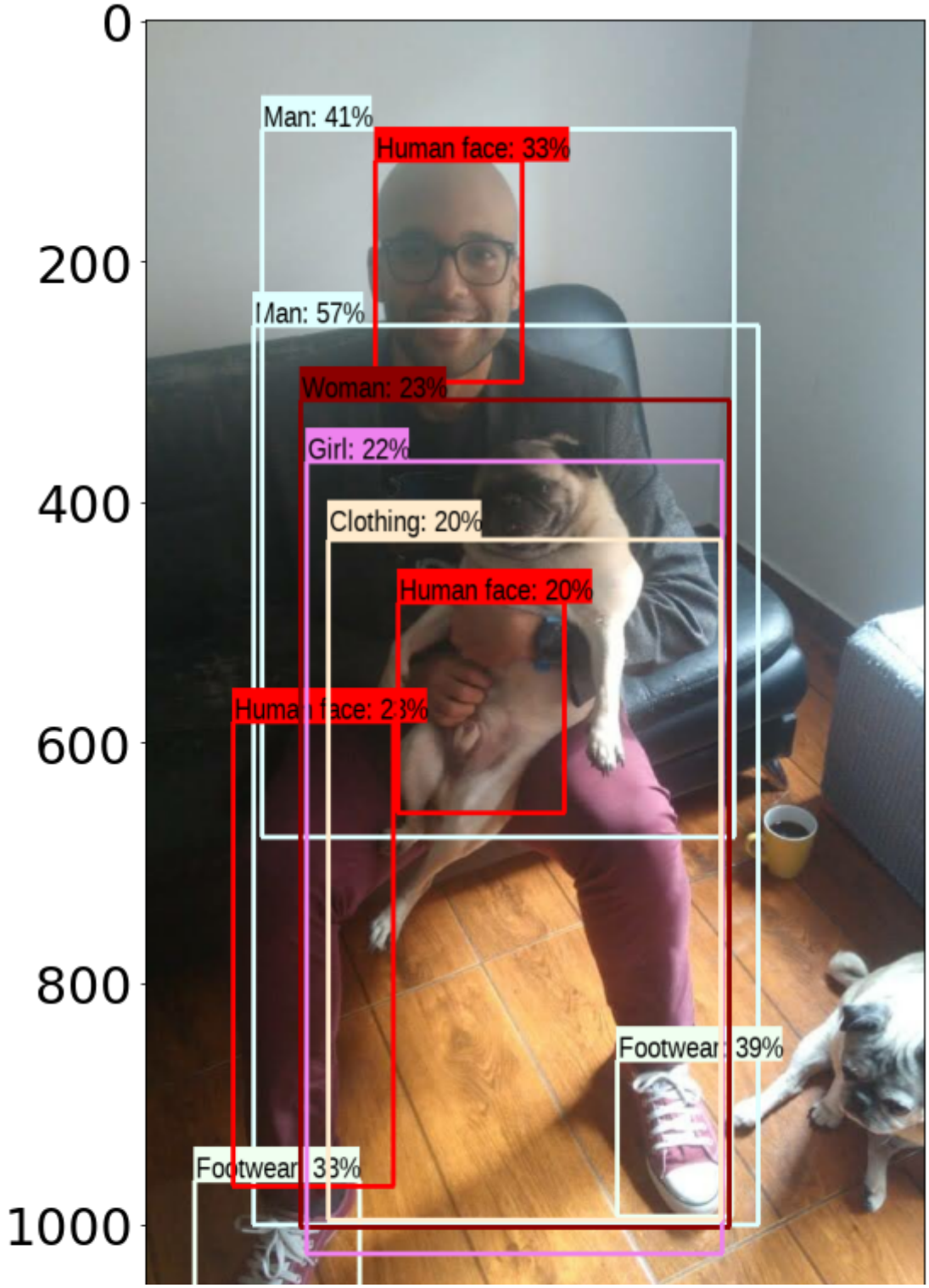
```
def load_img(path):
  img = tf.io.read_file(path)
  img = tf.image.decode_jpeg(img, channels=3)
  return img
```

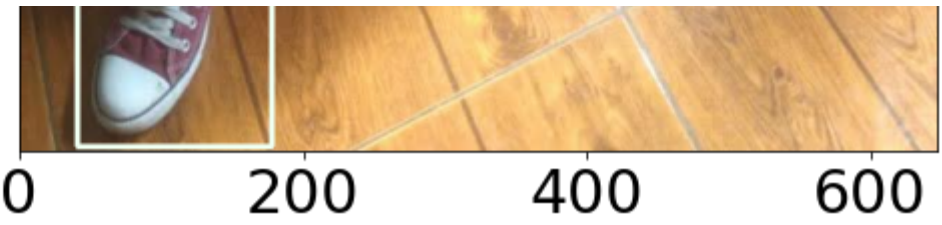
```
def run_detector(detector, path):
  #img = np.array(load_img(path))
  img = np.array(image_load)
  converted_img  = tf.image.convert_image_dtype(img, tf.float32)[tf.newaxis, ...]
  start_time = time.time()
  result = detector(converted_img)
  end_time = time.time()

  result = {key:value.numpy() for key,value in result.items()}

  print("Found %d objects." % len(result["detection_scores"]))
  print("Inference time: ", end_time-start_time)

  image_with_boxes = draw_boxes(
      #img.numpy(), result["detection_boxes"],
      img, result["detection_boxes"],
      result["detection_class_entities"], result["detection_scores"])

  display_image(image_with_boxes)
```

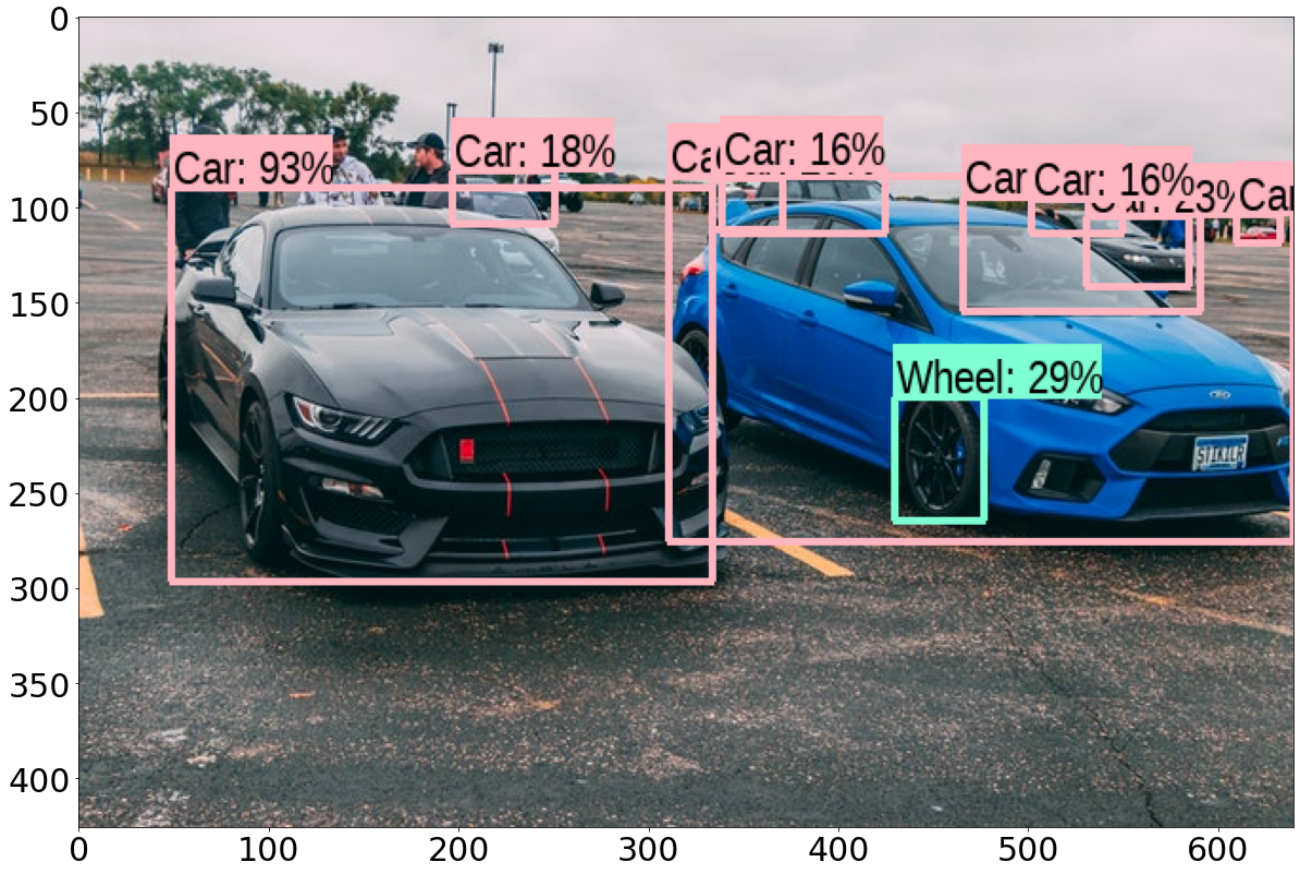
```
%%time
import time

test_images_paths = glob.glob('test_images/*')
```

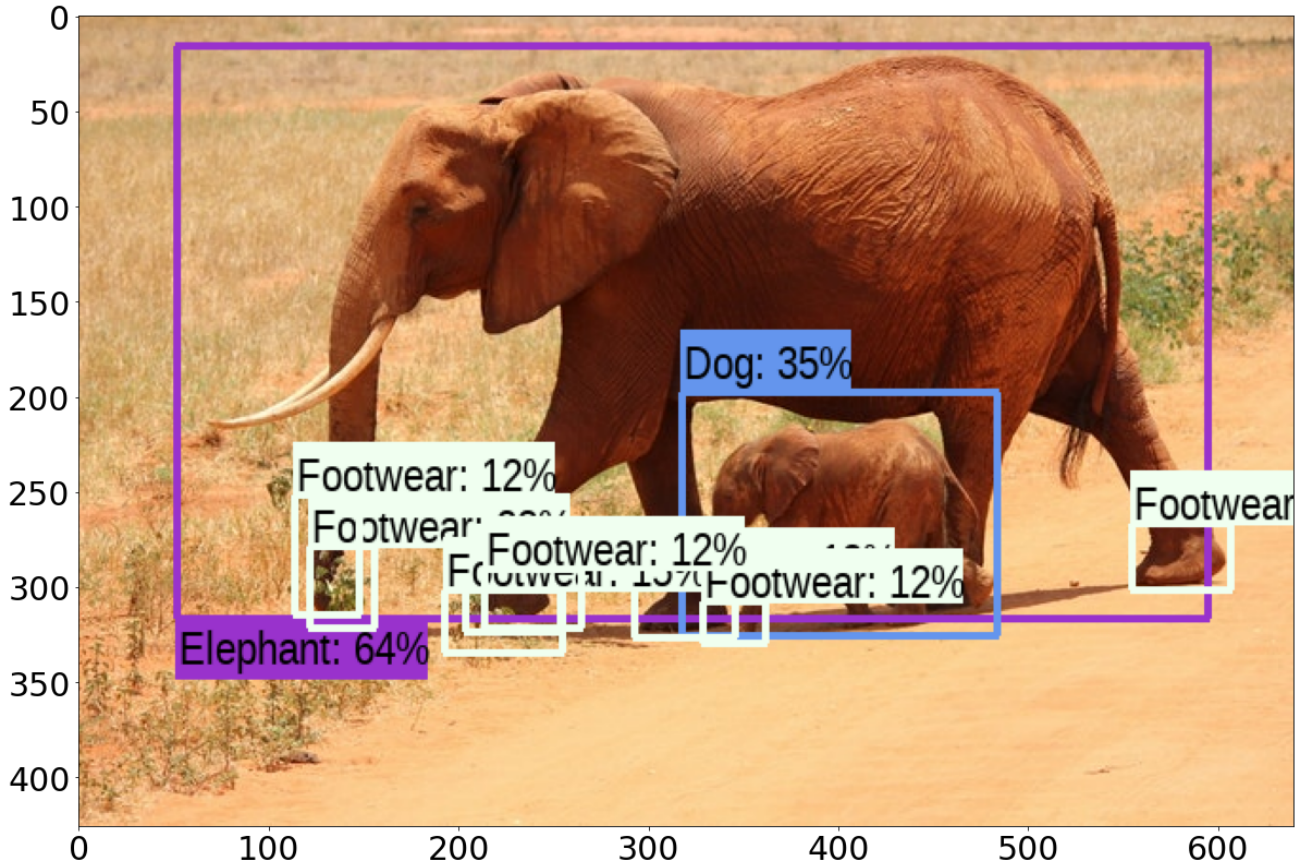
```
for image_path in test_images_paths:
    image_load = load_img(image_path)
    #get_and_save_predictions(model, image_path)
    run_detector(model, image_load)
```

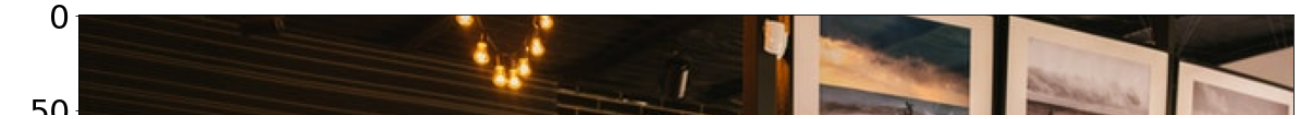
```
Found 100 objects.
Inference time:  0.2592189311981201
Found 100 objects.
Inference time:  0.15335679054260254
Found 100 objects.
Inference time:  0.15506243705749512
Found 100 objects.
Inference time:  0.1459641456604004
Found 100 objects.
Inference time:  0.13845443725585938
Found 100 objects.
Inference time:  0.14296507835388184
Found 100 objects.
Inference time:  0.13794898986816406
CPU times: user 1.3 s, sys: 304 ms, total: 1.6 s
Wall time: 1.56 s
```

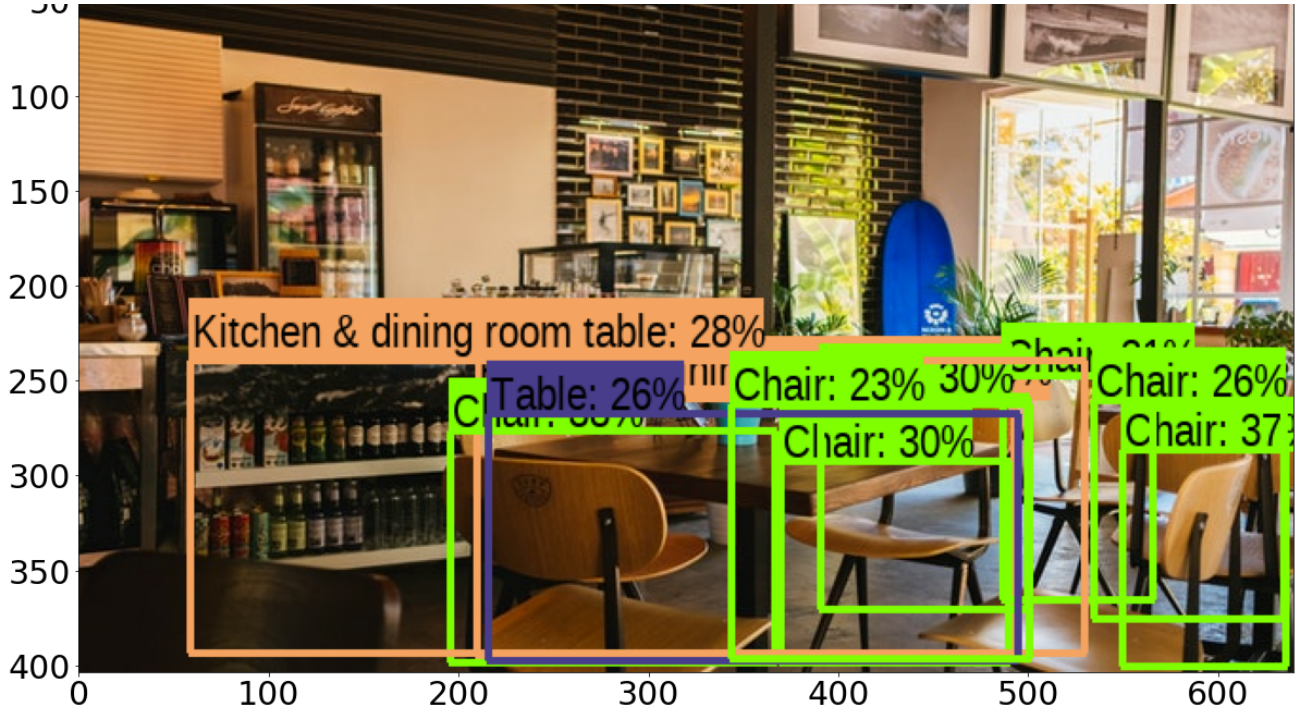


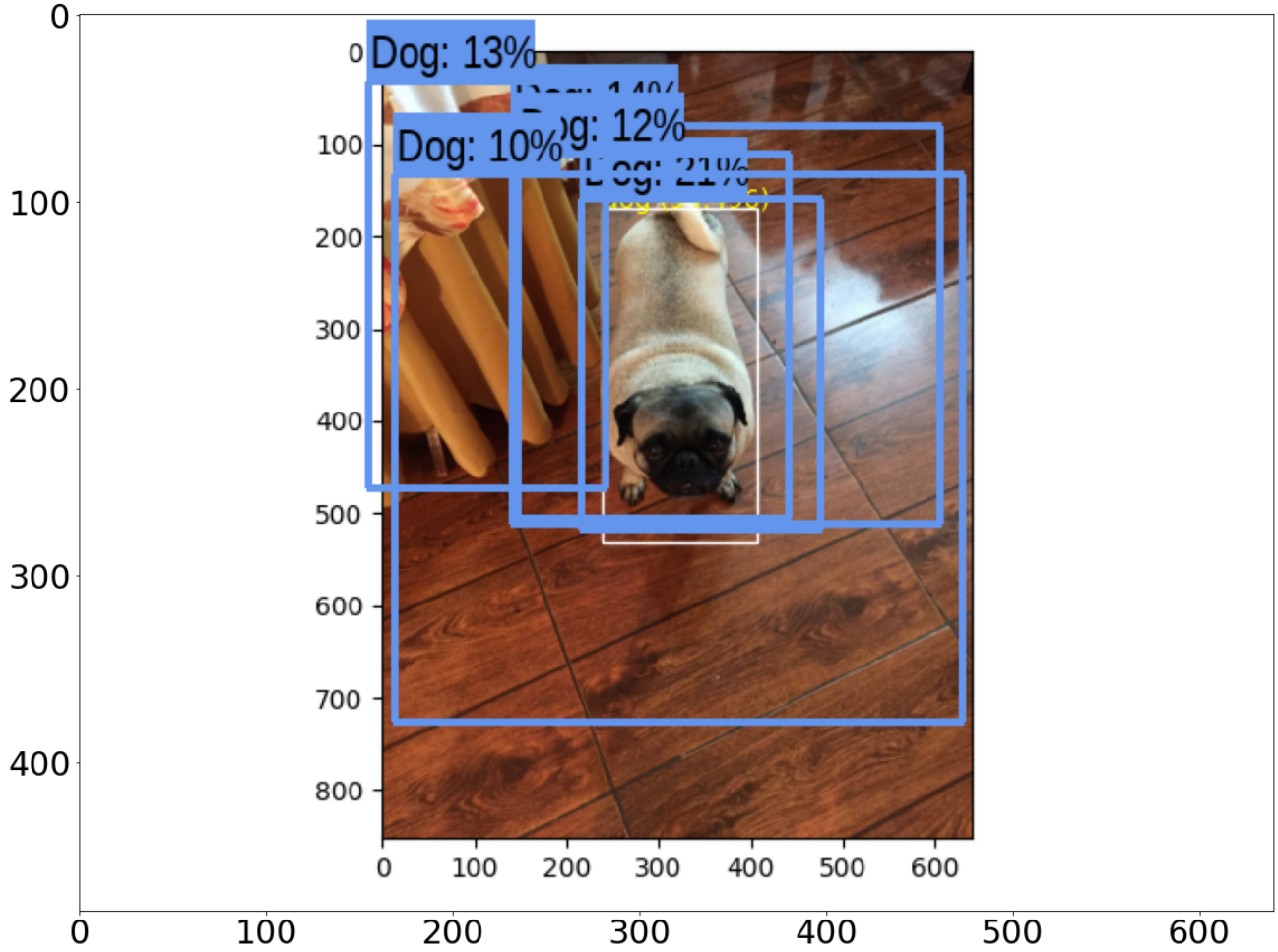

0
200
400
600


Car: 93%
Car: 18%
Car: 16%
Car
Car: 16%
Wheel: 29%
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

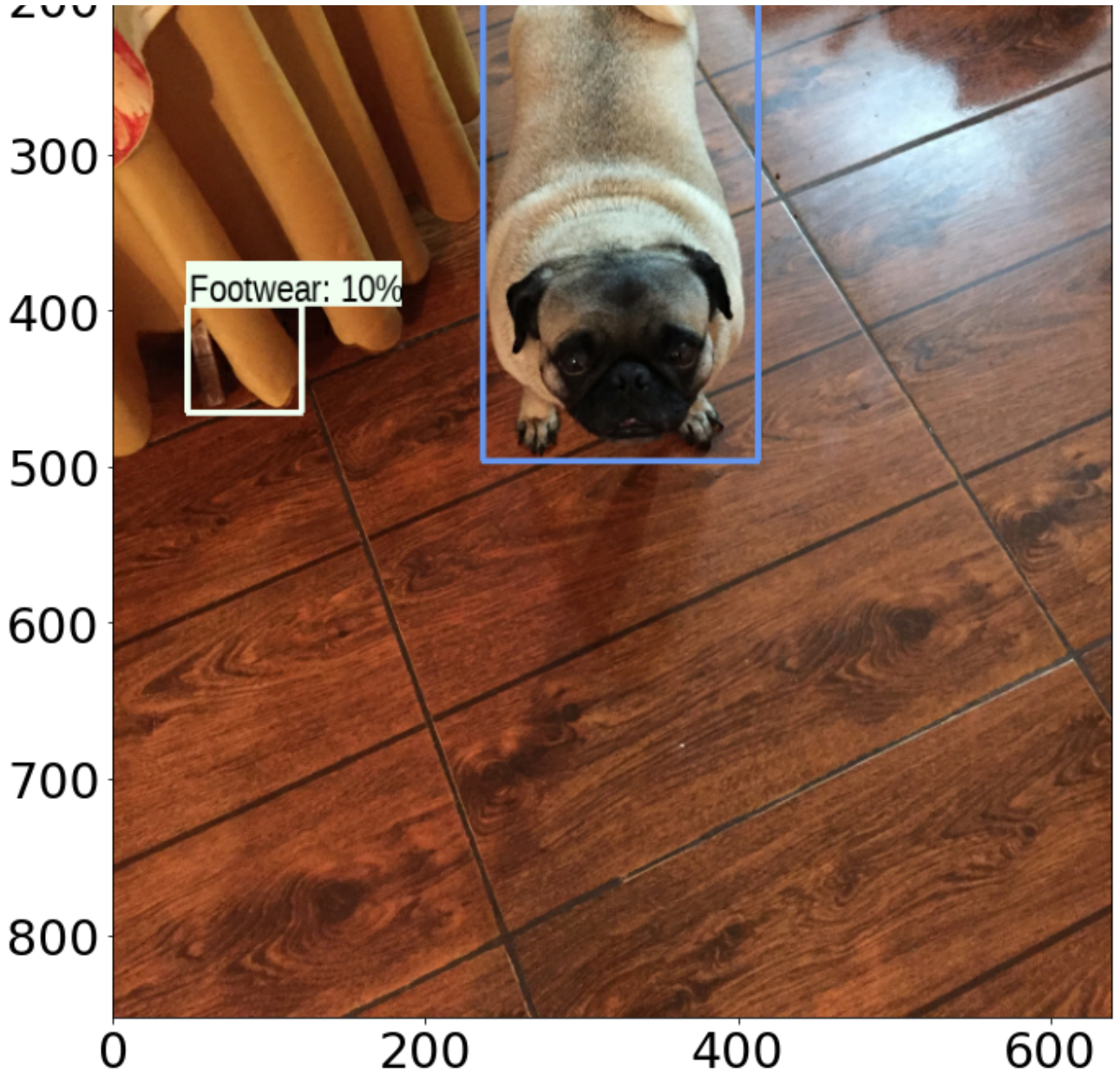

Dog: 35%
Footwear: 12%
Footwear: 12%
Footwear: 12%
Footwear
Elephant: 64%
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

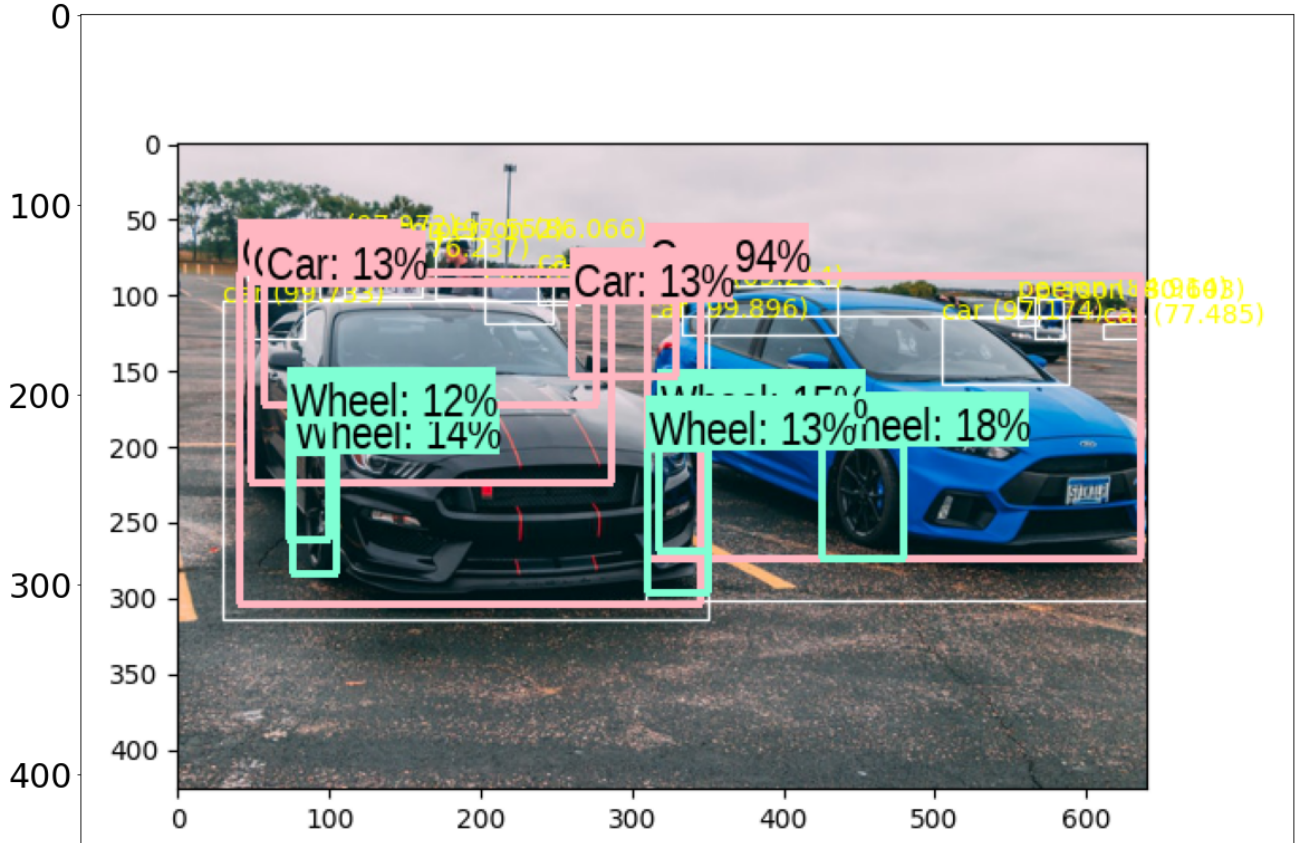

0
50

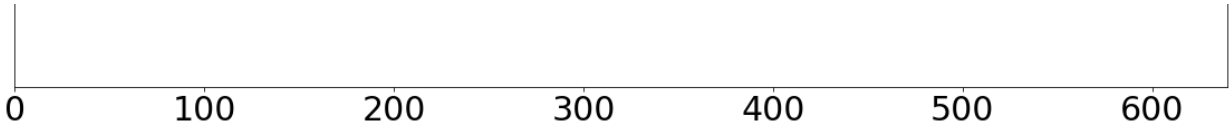

Kitchen & dining room table: 28%
Table: 26%
Chair: 23%
Chair: 30%
Chair: 26%


Dog: 13%
Dog: 12%
Dog: 10%


Dog: 25%


Footwear: 10%
300
400
500
600
700
800
0
200
400
600


Car: 13%
Car: 13%
94%
Wheel: 12%
Wheel: 14%
Wheel: 13%
Wheel: 18%
0
100
200
300
400
50
150
250
350
500
600


0
100
200
300
400
500
600

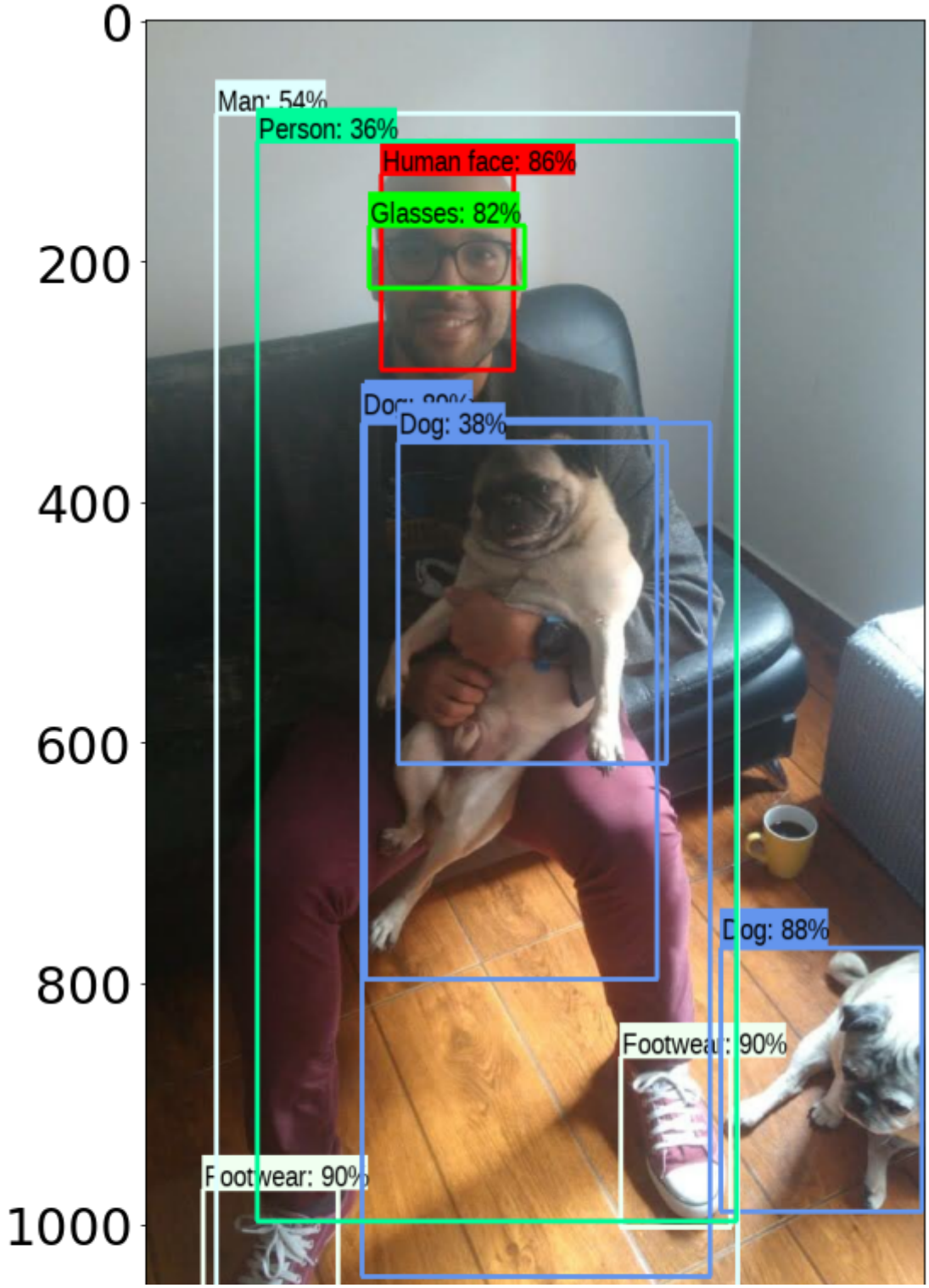
Fast R-CNN - Inception_ResNet_v2

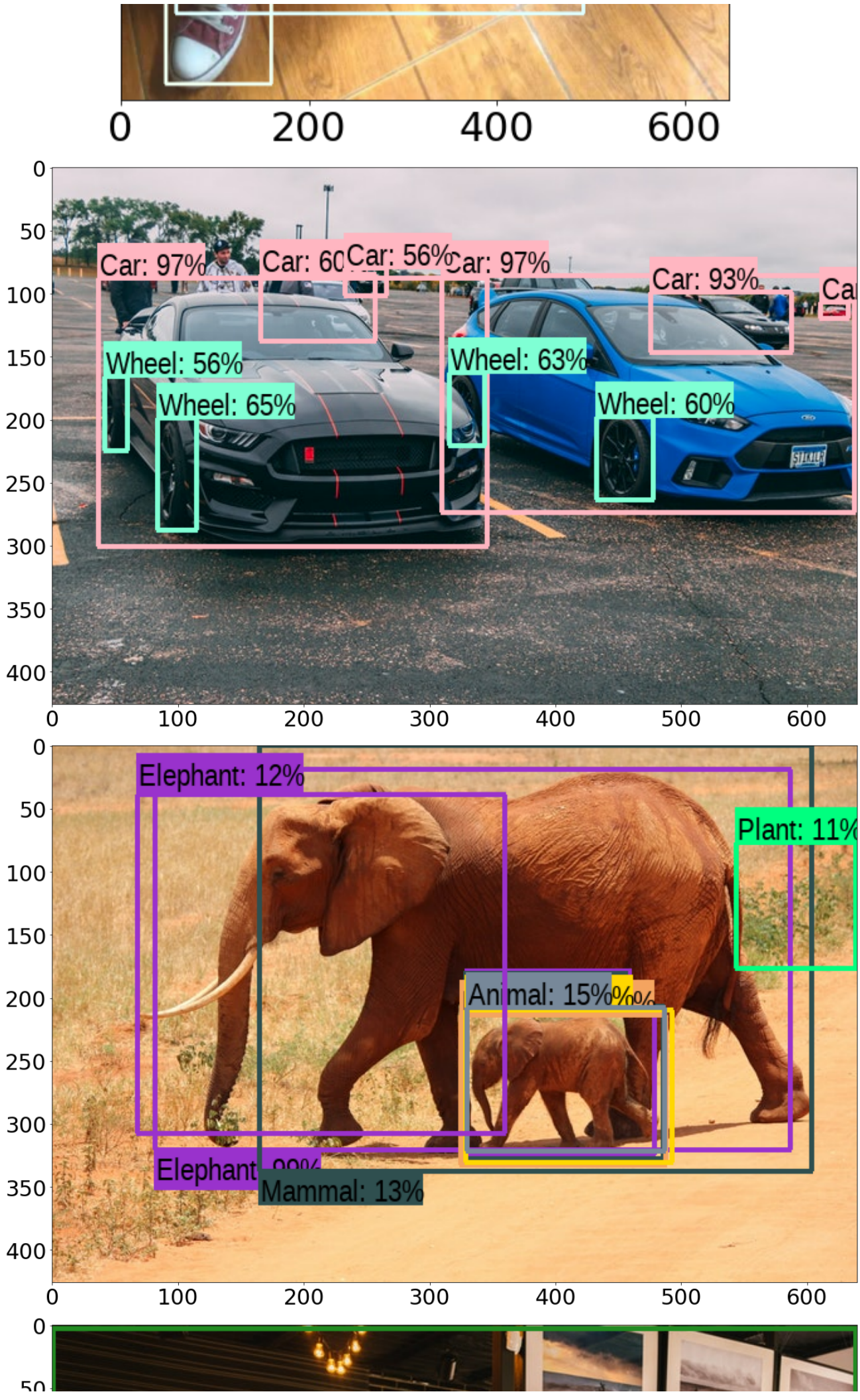
```
%%time

MODEL_PATH = ('https://tfhub.dev/google/faster_rcnn/openimages_v4/inception_resnet_
model = hub.load(MODEL_PATH).signatures['default']
```

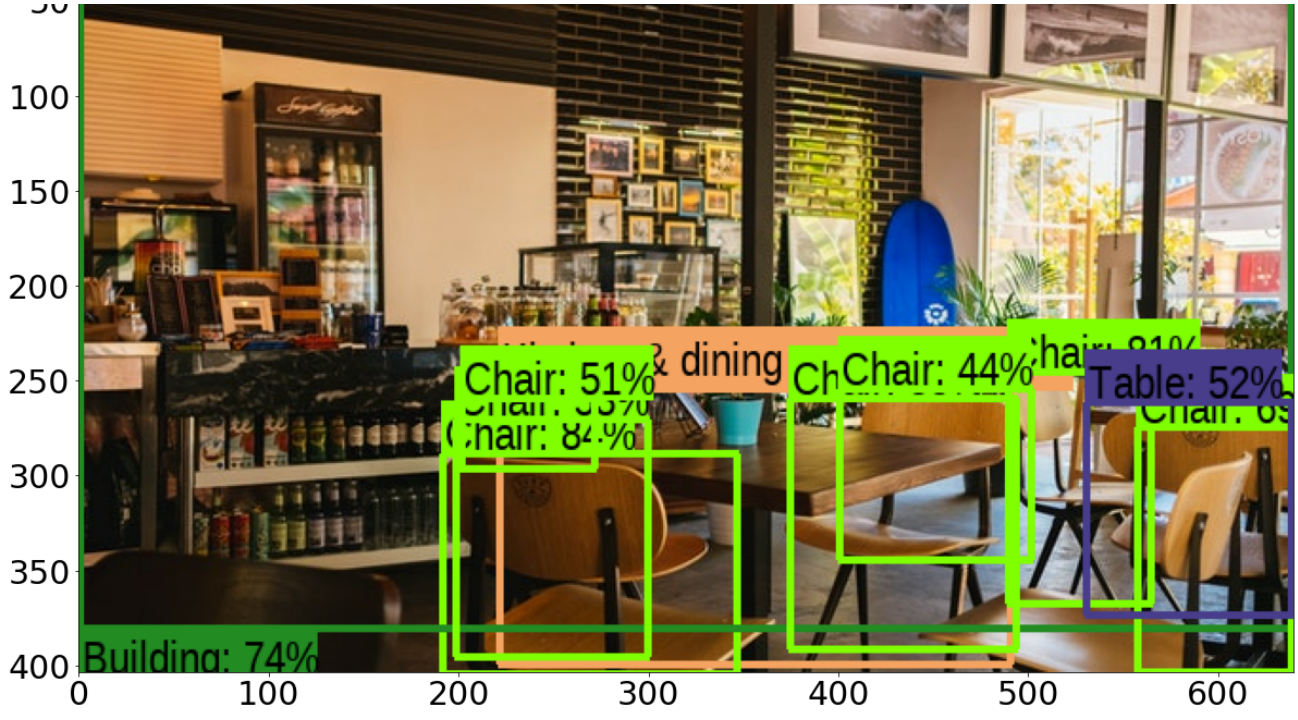
```
INFO:tensorflow:Saver not created because there are no variables in the graph
INFO:tensorflow:Saver not created because there are no variables in the graph
CPU times: user 1min 9s, sys: 3.2 s, total: 1min 12s
Wall time: 1min 11s
```

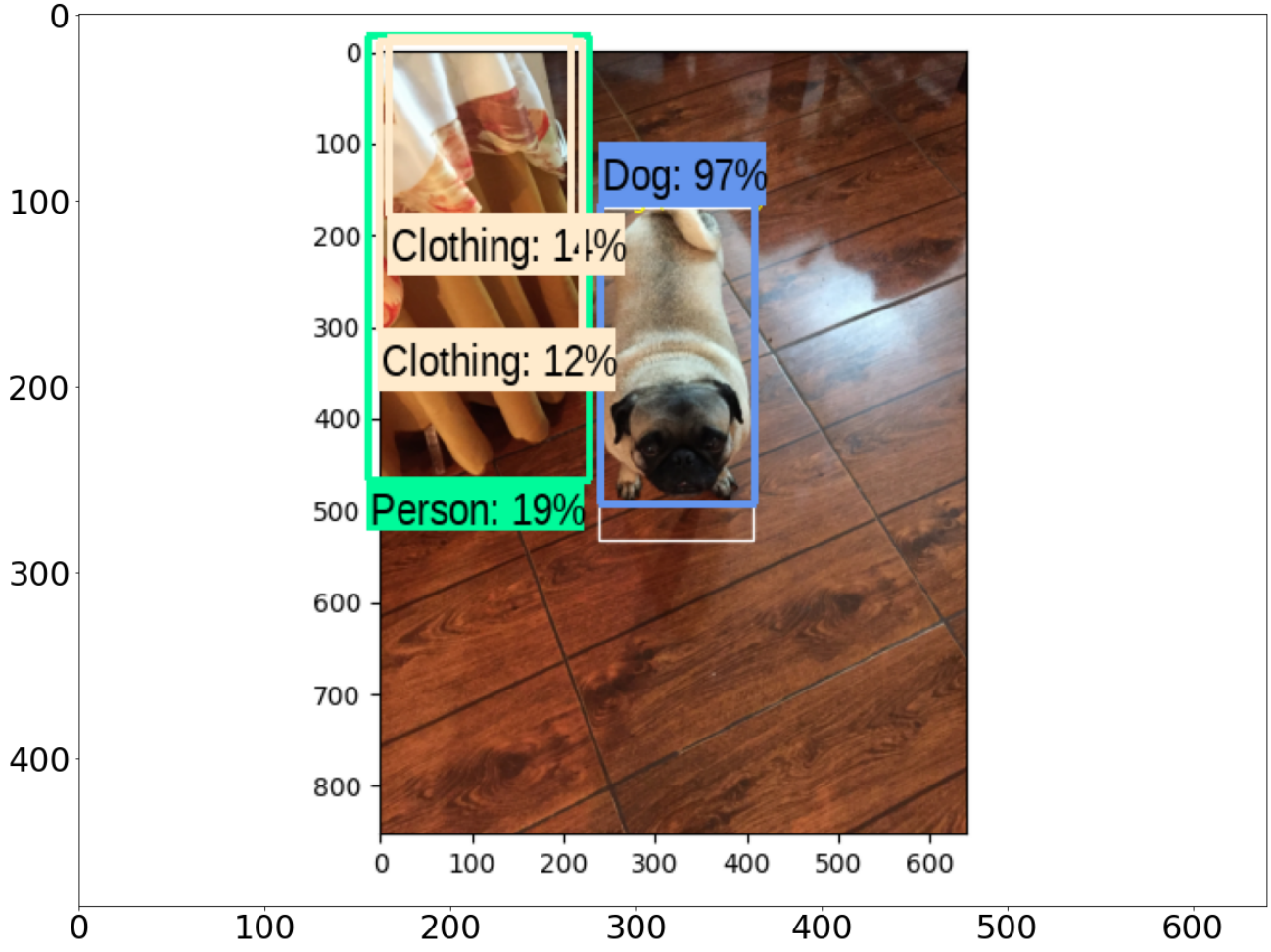
```
%%time
import time

test_images_paths = glob.glob('test_images/*')
for image_path in test_images_paths:
    image_load = load_img(image_path)
    #get_and_save_predictions(model, image_path)
    run_detector(model, image_load)
```

```
Found 100 objects.
Inference time:  29.08570909500122
Found 100 objects.
Inference time:  3.0380373001098633
Found 100 objects.
Inference time:  1.2008917331695557
Found 100 objects.
Inference time:  3.158496141433716
Found 100 objects.
Inference time:  2.903021812438965
Found 100 objects.
Inference time:  2.9254162311553955
Found 100 objects.
Inference time:  1.1527056694030762
CPU times: user 37.2 s, sys: 7.36 s, total: 44.5 s
Wall time: 43.9 s
```




0
200
400
600
0
50
100
150
200
250
300
350
400
Car: 97%
Car: 60
Car: 56%
Car: 97%
Car: 93%
Ca
Wheel: 56%
Wheel: 65%
Wheel: 63%
Wheel: 60%
0
100
200
300
400
500
600
Elephant: 12%
Plant: 11%
Animal: 15%
Elephant
Mammal: 13%


Chair: 51%
Chair: 44%
Table: 52%
Building: 74%


Dog: 97%
Clothing: 12%
Person: 19%


Dog: 99%

Нейронні мережі

Лекція_12

********** **********

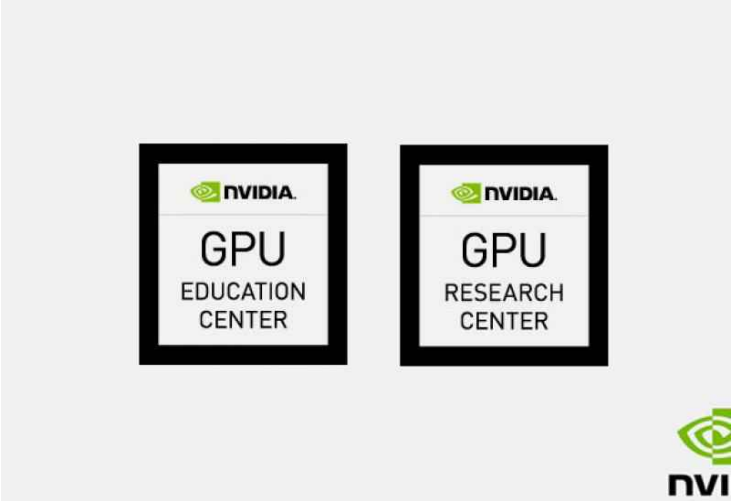
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 12 - Нейронні мережі - Сучасні CNN – Супер роздільна здатність зображення

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1
Супер роздільна здатність зображення
https://drive.google.com/file/d/1KWe2MgZuTuuJDbwzToB3oogU99d2qDMg/view?usp=sharing

Lecture 12 - Image Super Resolution

(C) partially based on the works by d2l Open Source book authors, TF Datasets contributors, Krasser, Birla, ...

Beside classification and object detection tasks, the other very important and canonical task is **Image Super Resolution (ISR)**.

This DEMO is an introduction to ISR implemented by Tensorflow 2.0.

This DEMO will introduce techniques used for ISR with some attention to introduction of:

- specialized residual network architectures for ISR,
- generative adversarial networks (GANs) for fine-tuning super-resolution models.

Part 0.Preparatory Actions

```
! nvidia-smi
```

```
Wed Apr 21 16:53:57 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.67       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   44C    P0    58W / 149W |    124MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

```
! pwd
```

```
/content
```

```
! git clone https://github.com/krasserm/super-resolution.git
```

```
fatal: destination path 'super-resolution' already exists and is not an empty
```

```
! cp -r ./super-resolution/* ./
```

```
! ls super-resolution
```

```
article.ipynb  docs                 example-srgan.ipynb  model      utils.py
data.py        environment.yml      example-wdsr.ipynb   README.md
demo           example-edsr.ipynb   LICENSE              train.py
```

Part 1.Intro to ISR by DNNs

Super-resolution is the process of recovering a high-resolution (HR) image from a low-resolution (LR) image. We will refer to a recovered HR image as *super-resolved image* or *SR image*. Super-resolution is an ill-posed problem since a large number of solutions exist for a single pixel in an LR image. Simple approaches like bilinear or bicubic interpolation use only local information in an LR image to compute pixel values in the corresponding SR image.

Supervised ML approaches, on the other hand, learn mapping functions from LR images to HR images from a large number of examples. Super-resolution models are trained with LR images as input and HR images as target. The mapping function learned by these models is the inverse of a downgrade function that transforms HR images to LR images. Downgrade functions can be known or unknown.

Known downgrade functions are used in image processing pipelines, for example, like bicubic downsampling. With known downgrade functions, LR images can be automatically obtained from HR images. This allows the creation of large training datasets from a vast amount of freely available HR images which enables self-supervised learning.

If the downgrade function is unknown, supervised model training requires existing LR and HR image pairs to be available which can be difficult to collect. Alternatively, unsupervised learning methods can be used that learn to approximate the downgrade function from unpaired LR and HR images. In this article though, we will use a known downgrade function (bicubic downsampling) and follow a supervised learning approach.

A more detailed overview on single image super-resolution is given in these papers. A higher-level training API for the example code in this article is implemented in this repository.

High-level architecture

Many state-of-the-art super-resolution models learn most of the mapping function in LR space followed by one or more upsampling layers at the end of the network. This is called *post-upsampling SR* in Fig. 1. Upsampling layers are learnable and trained together with the preceding convolution layers in an end-to-end manner.

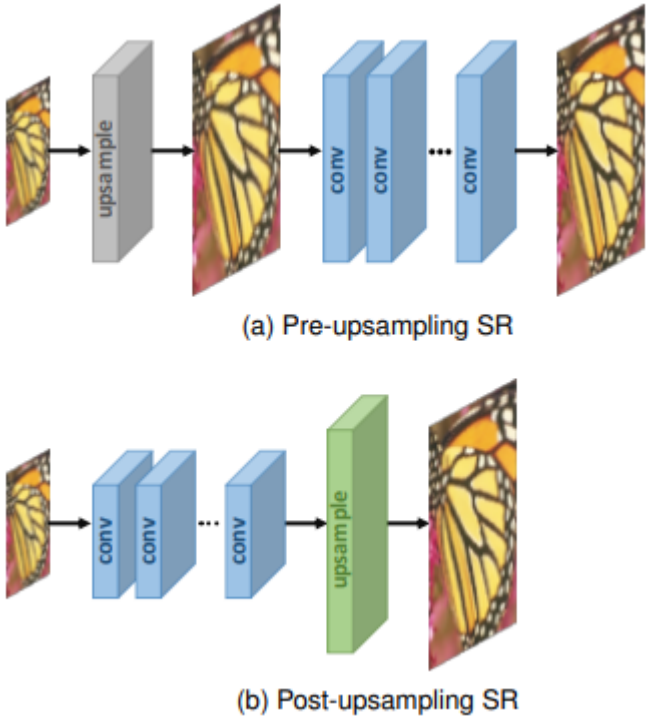


Fig. 1. Upsampling layers positions.

Earlier approaches first upsampled the LR image with a pre-defined upsampling operation and then learned the mapping in HR space (*pre-upsampling SR*). A disadvantage of this approach is that more parameters per layer are required which leads to higher computational costs and limits the construction of deeper neural networks.

Residual design

Super-resolution requires that most of the information contained in an LR image must be preserved in the SR image. Super-resolution models therefore mainly learn the residuals between LR and HR images. Residual network designs are therefore of high importance: identity information is conveyed via skip connections whereas reconstruction of high frequency content is done on the main path of the network.

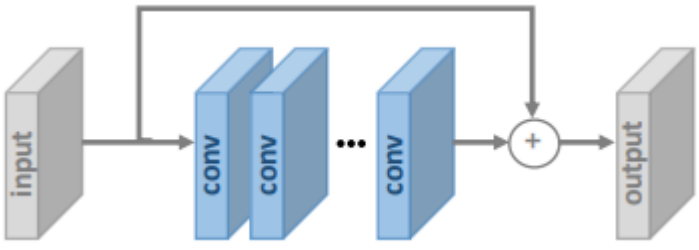


Fig. 2. Global skip connection.

Fig. 2. shows a global skip connection over several layers. These layers are often residual blocks as in ResNet or specialized variants (see sections EDSR and WDSR). Local skip connections in residual blocks make the network easier to optimize and therefore support the construction of deeper networks.

Upsampling layer

The upsampling layer used in this article is a sub-pixel convolution layer. Given an input of size $H \times W \times C$ and an upsampling factor s, the sub-pixel convolution layer first creates a representation of size $H \times W \times s^2 C$ via a convolution operation and then reshapes it to $sH \times sW \times C$, completing the upsampling operation. The result is an output spatially scaled by factor s.

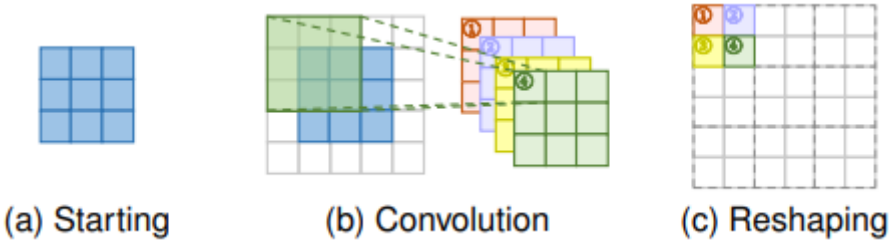


Fig. 3. Sub-pixel convolution.

An alternative are transposed convolution layers. Transposed convolutions can be learned too but have the disadvantage that they have a smaller receptive field than sub-pixel convolutions and can therefore process less contextual information which often results in less accurate predictions.

Super-resolution models

EDSR

One super-resolution model that follows this high-level architecture is described in the paper Enhanced Deep Residual Networks for Single Image Super-Resolution (EDSR). It is a winner of the NTIRE 2017 super-resolution challenge. Here's an overview of the EDSR architecture:

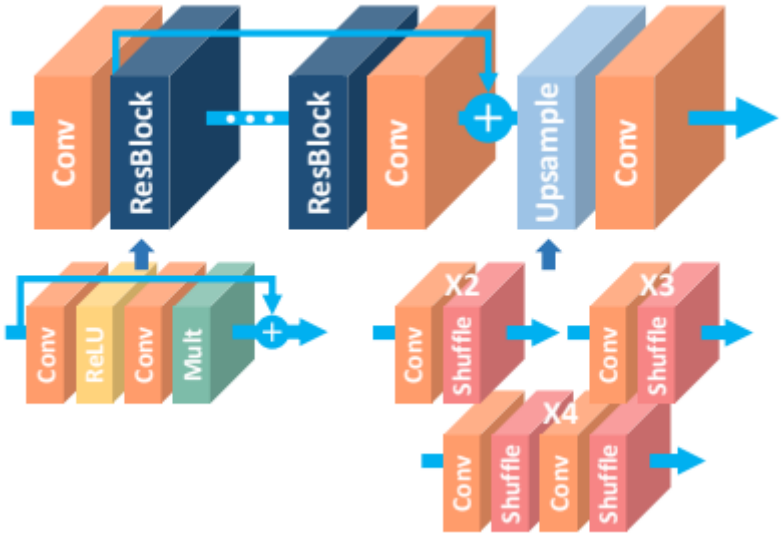


Fig. 4. EDSR architecture.

Its residual block design differs from that of ResNet. Batch normalization layers have been removed together with the final ReLU activation as shown on the right side of Fig. 5.

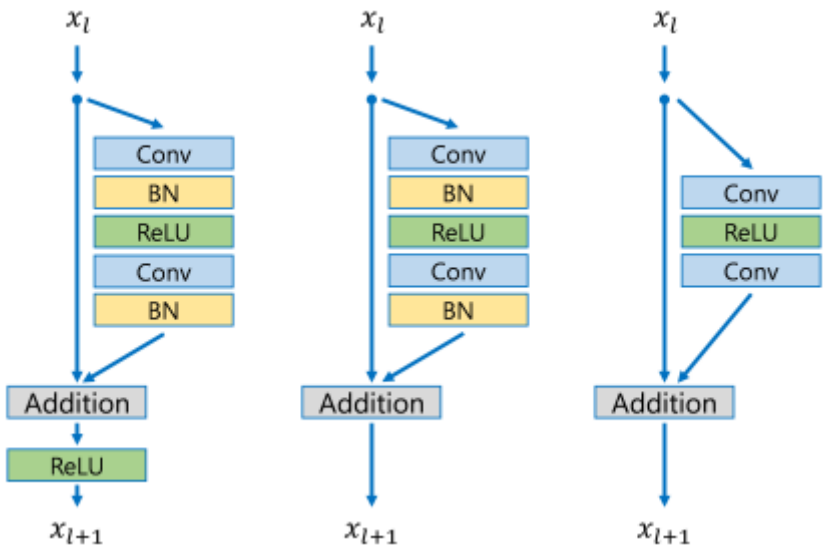


Fig. 5. Residual block design in ResNet (left) and in EDSR (right).

The EDSR authors argue that batch normalization loses scale information of images and reduces the range flexibility of activations. Removal of batch normalization layers not only increases super-resolution performance but also reduces GPU memory up to 40% so that significantly larger models can be trained.

EDSR uses a single sub-pixel upsampling layer for super-resolution scales (i.e. upsampling factors) $\times 2$ and $\times 3$ and two upsampling layers for scale $\times 4$. The following `edsr` function implements the EDSR model with Tensorflow 2.0. The default arguments correspond to the

```
import numpy as np
import tensorflow as tf

from tensorflow.keras.layers import Add, Conv2D, Input, Lambda
from tensorflow.keras.models import Model

DIV2K_RGB_MEAN = np.array([0.4488, 0.4371, 0.4040]) * 255


def edsr(scale, num_filters=64, num_res_blocks=8, res_block_scaling=None):
    """Creates an EDSR model."""
    x_in = Input(shape=(None, None, 3))
    x = Lambda(normalize)(x_in)

    x = b = Conv2D(num_filters, 3, padding='same')(x)
    for i in range(num_res_blocks):
        b = res_block(b, num_filters, res_block_scaling)
    b = Conv2D(num_filters, 3, padding='same')(b)
    x = Add()([x, b])

    x = upsample(x, scale, num_filters)
    x = Conv2D(3, 3, padding='same')(x)
```

```
    x = Lambda(denormalize)(x)
    return Model(x_in, x, name="edsr")


def res_block(x_in, filters, scaling):
    """Creates an EDSR residual block."""
    x = Conv2D(filters, 3, padding='same', activation='relu')(x_in)
    x = Conv2D(filters, 3, padding='same')(x)
    if scaling:
        x = Lambda(lambda t: t * scaling)(x)
    x = Add()([x_in, x])
    return x


def upsample(x, scale, num_filters):
    def upsample_1(x, factor, **kwargs):
        """Sub-pixel convolution."""
        x = Conv2D(num_filters * (factor ** 2), 3, padding='same', **kwargs)(x)
        return Lambda(pixel_shuffle(scale=factor))(x)

    if scale == 2:
        x = upsample_1(x, 2, name='conv2d_1_scale_2')
    elif scale == 3:
        x = upsample_1(x, 3, name='conv2d_1_scale_3')
    elif scale == 4:
        x = upsample_1(x, 2, name='conv2d_1_scale_2')
        x = upsample_1(x, 2, name='conv2d_2_scale_2')

    return x


def pixel_shuffle(scale):
    return lambda x: tf.nn.depth_to_space(x, scale)


def normalize(x):
    return (x - DIV2K_RGB_MEAN) / 127.5


def denormalize(x):
    return x * 127.5 + DIV2K_RGB_MEAN
```

WDSR

Another super-resolution model is a derivative of EDSR and is described in the paper [Wide Activation for Efficient and Accurate Image Super-Resolution](), a winner in the realistic tracks of the [NTIRE 2018]() super-resolution challenge. It makes further changes to the residual block design by reducing the number of channels on the identity mapping path and increasing the number of channels in each residual block without increasing the total number of parameters. The residual block design of their WDSR-A and WDSR-B models is shown in Fig. 6, a Tensorflow 2.0 implementation is available [here]().

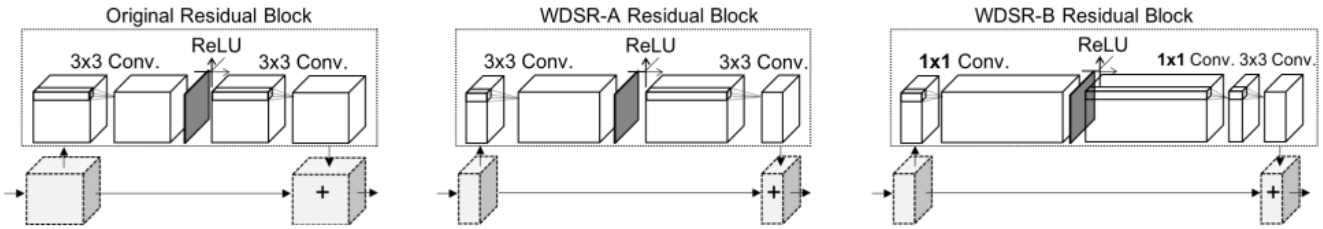


Fig. 6. Residual block design in EDSR (left), WDSR-A (middle) and WDSR-B (right).

The authors conjecture that increasing the number of channels before ReLU in residual blocks allows more information to pass through the activation function which further increases model performance. They also found that the implementation of weight normalization further eases training and convergence of deeper models so that they could use learning rates that are an order of magnitude higher compared to those used in EDSR training.

This is not surprising as the removal of weight normalization layers in EDSR makes it more difficult to train deeper models. Weight normalization is just a reparameterization of neural network weights that decouples the direction of weight vectors from their magnitude which improves the conditioning of the optimization problem and speeds up convergence.

Data-dependent initialization of weight normalization layer parameters is not done though. It would rescale features similar to batch normalization which would decrease model performance as has been shown in the EDSR paper and confirmed in the WDSR paper. On the other hand, using weight normalization alone without data-dependent initialization leads to better accuracy of deeper WDSR models.

Part 2.Training

If you want to skip running trained code in this section, you can download pre-trained models here and use in the next section for generating SR images from LR images.

Data

For training EDSR and WDSR models we will use the DIV2K dataset. It is a dataset of LR and HR image pairs with a large diversity of contents. LR images are available for different downgrade functions. We will use `bicubic` downsampling here. There are 800 training HR images and 100 validation HR images.

For data augmentation, random crops, flips and rotations are made to get a large number of different training images. A `DIV2K` data loader automatically downloads DIV2K images for given scale and downgrade function and provides LR and HR image pairs as `tf.data.Dataset`.

```
from data import DIV2K
```

```
%%time

train = DIV2K(scale=4, downgrade='bicubic', subset='train')
train_ds = train.dataset(batch_size=16, random_transform=True)
```

```
Downloading data from http://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_train_LR_
246915072/246914039 [==============================] - 28s 0us/step
Caching decoded images in .div2k/caches/DIV2K_train_LR_bicubic_X4.cache ...
Cached decoded images in .div2k/caches/DIV2K_train_LR_bicubic_X4.cache.
Downloading data from http://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_train_HR.
3530604544/3530603713 [==============================] - 369s 0us/step
Caching decoded images in .div2k/caches/DIV2K_train_HR.cache ...
Cached decoded images in .div2k/caches/DIV2K_train_HR.cache.
CPU times: user 2min 22s, sys: 27.2 s, total: 2min 49s
Wall time: 10min 33s
```

```
%%time

div2k_valid = DIV2K(scale=4, subset='valid', downgrade='bicubic')
valid_ds = div2k_valid.dataset(batch_size=16, random_transform=True, repeat_count=
```

```
CPU times: user 270 ms, sys: 4.92 ms, total: 275 ms
Wall time: 276 ms
```

```
%%time
# Alternative way to dataset - VERY LONG for the 1st attempt - it is used here for
import tensorflow_datasets as tfds

valid_ds, valid_info = tfds.load('Div2k', split='validation', with_info=True)
```

```
CPU times: user 40.4 ms, sys: 1.16 ms, total: 41.5 ms
Wall time: 55.1 ms
```

```
# Show examples of images
import matplotlib.pyplot as plt

tfds.as_dataframe(valid_ds.take(4), valid_info)
```

Metrics + Models

Pixel-wise losses

The pixel-wise L^2 loss and the pixel-wise L^1 loss are frequently used loss functions for training super-resolution models. They measure the pixel-wise mean squared error and the pixel-wise mean absolute error, respectively, between an HR image I^{HR} and an SR image I^{SR}:

$$\mathcal{L}_{pixel,L^2}(I^{HR}, I^{SR}) = \frac{1}{HWC}\|I^{HR} - I^{SR}\|_2^2 \tag{1}$$

$$\mathcal{L}_{pixel,L^1}(I^{HR}, I^{SR}) = \frac{1}{HWC}\|I^{HR} - I^{SR}\|_1 \tag{2}$$

where H, W and C are the height, width and number of channels of the image, respectively. The pixel-wise L^2 loss directly optimizes PSNR, an evaluation metric often used in super-resolution competitions. Experiments have shown that the pixel-wise L^1 loss can sometimes achieve even better performance and is therefore used for EDSR and WDSR training.

PSNR

Peak signal-to-noise ratio (PSNR) is an engineering term for the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. Because many signals have a very wide dynamic range, PSNR is usually expressed as a logarithmic quantity using the decibel scale.

PSNR is most easily defined via **the mean squared error (MSE)**.

Given a noise-free $m \times n$ monochrome image I and its noisy approximation K, MSE is defined as:

$$\boldsymbol{MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2}$$

PSNR (in dB) is defined as:

$$
\begin{aligned}
PSNR &= 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \\
&= 20 \cdot \log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \\
&= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE)
\end{aligned}
$$

Here, MAX_I is the maximum possible pixel value of the image.

If the pixels are represented using 8 bits per sample $MAX_I = 255$.

More generally, when samples are represented using B bits per sample, then $MAX_I = 2^B - 1$.

PSNR is implemented in TensorFlow 2.0 as:
https://www.tensorflow.org/api_docs/python/tf/image/psnr

SSIM

Structural Similarity Index Measure (SSIM) is a method for predicting the perceived quality of digital television and cinematic pictures, as well as other kinds of digital images and videos.

SSIM is used for **measuring the similarity** between two images.

The SSIM index is a full reference metric; in other words, the measurement or prediction of image quality is based on an initial uncompressed or distortion-free image as reference.

The SSIM index is calculated on various windows of an image. The measure between two windows x and y of common size $N \times N$ is:[4]

$$
\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}
$$

where:

μ_x the average of x;

μ_y the average of y;

σ_x^2 the variance of x;

σ_y^2 the variance of y;

σ_{xy} the covariance of x and y;

$c_1 = (k_1 L)^2, c_2 = (k_2 L)^2$;

$L = MAX_I = 2^B - 1$ the dynamic range of the pixel-values;

$k_1 = 0.01, k_2 = 0.03$ by default.

Finally, **SSIM** is a decimal value between 0 and 1:

- value 1 is only reachable in the case of two identical sets of data and therefore indicates **perfect structural similarity**,
- value of 0 indicates **no structural similarity**.

SSIM is typically calculated using a **sliding Gaussian window** of size 11x11 or a **block window** of size 8×8. The window can be displaced pixel-by-pixel on the image to create an SSIM quality map of the image. In the case of video quality assessment, the authors propose to use only a subgroup of the possible windows to reduce the complexity of the calculation.

SSIM is implemented in TensorFlow 2.0 as:
https://www.tensorflow.org/api_docs/python/tf/image/ssim

MS-SSIM

Multiscale SSIM (MS-SSIM) is the more advanced form of SSIM.

MS-SSIM is conducted over multiple scales through a process of multiple stages of sub-sampling, reminiscent of multiscale processing in the early vision system.

MS-SSIM has been shown to perform equally well or better than SSIM on different subjective image and video databases.

MS-SSIM is implemented in TensorFlow 2.0 as:
https://www.tensorflow.org/api_docs/python/tf/image/ssim_multiscale

Pixel Loss -> Models

EDSR

```
%%time

import os

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers.schedules import PiecewiseConstantDecay

# Create directory for saving model weights
weights_dir = 'weights/article'
os.makedirs(weights_dir, exist_ok=True)

# EDSR baseline as described in the EDSR paper (1.52M parameters)
model_edsr = edsr(scale=4, num_res_blocks=16)

# Adam optimizer with a scheduler that halfs learning rate after 200,000 steps
optim_edsr = Adam(learning_rate=PiecewiseConstantDecay(boundaries=[200000], values

# Compile and train model for 300,000 steps with L1 pixel loss
```

```
model_edsr.compile(optimizer=optim_edsr, loss='mean_absolute_error')
#model_edsr.fit(train_ds, epochs=300, steps_per_epoch=1000)
model_edsr.fit(train_ds, epochs=30, steps_per_epoch=1)

# Save model weights
model_edsr.save_weights(os.path.join(weights_dir, 'weights-edsr-16-x4.h5'))
```

```
Epoch 1/30
1/1 [==============================] - 44s 44s/step - loss: 159.5664
Epoch 2/30
1/1 [==============================] - 0s 114ms/step - loss: 80.2392
Epoch 3/30
1/1 [==============================] - 0s 100ms/step - loss: 85.2447
Epoch 4/30
1/1 [==============================] - 3s 3s/step - loss: 68.2489
Epoch 5/30
1/1 [==============================] - 4s 4s/step - loss: 61.8405
Epoch 6/30
1/1 [==============================] - 4s 4s/step - loss: 65.9680
Epoch 7/30
1/1 [==============================] - 4s 4s/step - loss: 58.9226
Epoch 8/30
1/1 [==============================] - 4s 4s/step - loss: 55.1938
Epoch 9/30
1/1 [==============================] - 4s 4s/step - loss: 52.9230
Epoch 10/30
1/1 [==============================] - 4s 4s/step - loss: 46.9493
Epoch 11/30
1/1 [==============================] - 4s 4s/step - loss: 38.1564
Epoch 12/30
1/1 [==============================] - 4s 4s/step - loss: 42.2474
Epoch 13/30
1/1 [==============================] - 4s 4s/step - loss: 36.4787
Epoch 14/30
1/1 [==============================] - 4s 4s/step - loss: 33.6342
Epoch 15/30
1/1 [==============================] - 4s 4s/step - loss: 31.6794
Epoch 16/30
1/1 [==============================] - 4s 4s/step - loss: 38.5694
Epoch 17/30
1/1 [==============================] - 4s 4s/step - loss: 35.7862
Epoch 18/30
1/1 [==============================] - 4s 4s/step - loss: 33.2851
Epoch 19/30
1/1 [==============================] - 4s 4s/step - loss: 32.0955
Epoch 20/30
1/1 [==============================] - 4s 4s/step - loss: 35.3857
Epoch 21/30
1/1 [==============================] - 4s 4s/step - loss: 31.4447
Epoch 22/30
1/1 [==============================] - 4s 4s/step - loss: 32.9607
Epoch 23/30
1/1 [==============================] - 4s 4s/step - loss: 32.0649
Epoch 24/30
1/1 [==============================] - 4s 4s/step - loss: 28.2578
Epoch 25/30
1/1 [==============================] - 4s 4s/step - loss: 30.2251
Epoch 26/30
1/1 [==============================] - 4s 4s/step - loss: 25.9892
```

```
Epoch 27/30
1/1 [==============================] - 4s 4s/step - loss: 24.6861
Epoch 28/30
1/1 [==============================] - 4s 4s/step - loss: 28.8908
Epoch 29/30
1/1 [==============================] - 4s 4s/step - loss: 26.6187
```

WDSR

```
! pip install tensorflow_addons
```

```
Collecting tensorflow_addons
  Downloading https://files.pythonhosted.org/packages/74/e3/56d2fe76f0bb7c88e
     |████████████████████████████████| 706kB 5.9MB/s
Requirement already satisfied: typeguard>=2.7 in /usr/local/lib/python3.7/dis
Installing collected packages: tensorflow-addons
Successfully installed tensorflow-addons-0.12.1
```

```
%%time

from model.wdsr import wdsr_b

# Custom WDSR B model (0.62M parameters)
model_wdsr = wdsr_b(scale=4, num_res_blocks=32)

# Adam optimizer with a scheduler that halfs learning rate after 200,000 steps
optim_wdsr = Adam(learning_rate=PiecewiseConstantDecay(boundaries=[200000], values

# Compile and train model for 300,000 steps with L1 pixel loss
model_wdsr.compile(optimizer=optim_wdsr, loss='mean_absolute_error')
#model_wdsr.fit(train_ds, epochs=300, steps_per_epoch=1000)
model_wdsr.fit(train_ds, epochs=30, steps_per_epoch=1)

# Save weights
model_wdsr.save_weights(os.path.join(weights_dir, 'weights-wdsr-b-32-x4.h5'))
```

```
Epoch 3/30
1/1 [==============================] - 4s 4s/step - loss: 255.4819
Epoch 4/30
1/1 [==============================] - 1s 874ms/step - loss: 182.1632
Epoch 5/30
1/1 [==============================] - 0s 223ms/step - loss: 99.2515
Epoch 6/30
1/1 [==============================] - 0s 220ms/step - loss: 106.8622
Epoch 7/30
1/1 [==============================] - 0s 194ms/step - loss: 104.8311
Epoch 8/30
1/1 [==============================] - 0s 218ms/step - loss: 103.6704
Epoch 9/30
1/1 [==============================] - 0s 195ms/step - loss: 81.7083
Epoch 10/30
1/1 [==============================] - 0s 202ms/step - loss: 81.7624
Epoch 11/30
1/1 [==============================] - 0s 189ms/step - loss: 78.7481
Epoch 12/30
```

```
1/1 [==============================] - 0s 169ms/step - loss: 68.7312
Epoch 13/30
1/1 [==============================] - 0s 196ms/step - loss: 59.6304
Epoch 14/30
1/1 [==============================] - 0s 192ms/step - loss: 61.5376
Epoch 15/30
1/1 [==============================] - 0s 219ms/step - loss: 48.9806
Epoch 16/30
1/1 [==============================] - 0s 179ms/step - loss: 50.9072
Epoch 17/30
1/1 [==============================] - 0s 190ms/step - loss: 51.9613
Epoch 18/30
1/1 [==============================] - 0s 198ms/step - loss: 42.8601
Epoch 19/30
1/1 [==============================] - 0s 184ms/step - loss: 44.6315
Epoch 20/30
1/1 [==============================] - 0s 196ms/step - loss: 48.2145
Epoch 21/30
1/1 [==============================] - 0s 211ms/step - loss: 44.6723
Epoch 22/30
1/1 [==============================] - 0s 180ms/step - loss: 41.9233
Epoch 23/30
1/1 [==============================] - 0s 194ms/step - loss: 35.9083
Epoch 24/30
1/1 [==============================] - 0s 180ms/step - loss: 40.1410
Epoch 25/30
1/1 [==============================] - 0s 170ms/step - loss: 34.3915
Epoch 26/30
1/1 [==============================] - 0s 181ms/step - loss: 32.9996
Epoch 27/30
1/1 [==============================] - 0s 152ms/step - loss: 30.9171
Epoch 28/30
1/1 [==============================] - 0s 157ms/step - loss: 27.5731
Epoch 29/30
1/1 [==============================] - 0s 164ms/step - loss: 33.2289
Epoch 30/30
1/1 [==============================] - 0s 173ms/step - loss: 31.5661
CPU times: user 44.9 s, sys: 3.91 s, total: 48.8 s
Wall time: 56.3 s
```

A major problem with pixel-wise loss functions is that they lead to poor perceptual quality. Generated SR images often lack high-frequency content, realistic textures and are perceived blurry. This problem is addressed with perceptual loss functions.

Content, Generator, Discriminator, and Perceptual losses

A milestone paper for generating SR images with better perceived quality is Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network (SRGAN). The authors use a perceptual loss function composed of a *content loss* and an *adversarial loss*. The content loss compares deep features extracted from SR and HR images with a pre-trained VGG network ϕ

$$\mathcal{L}_{content}(I^{HR}, I^{SR}; \phi, l) = \frac{1}{H_l W_l C_l} \|\phi_l(I^{HR}) - \phi_l(I^{SR})\|_2^2 \quad (3)$$

were $\phi_l(I)$ is the feature map at layer l and H_l, W_l and C_l are the height, width and number of channels of that feature map, respectively. They also train their super-resolution model as generator G in a generative adversarial network (GAN). The GAN descriminator D is optimized for descriminating SR from HR images whereas the generator is optimized for generating more realistic SR images in order to fool the discriminator. They combine the generator loss

$$\mathcal{L}_{generator}(I^{LR}; G, D) = -\log D(G(I^{LR})) \tag{4}$$

with the content loss to a perceptual loss which is used as optimization target for super-resolution model training:

$$\mathcal{L}_{perceptual} = \mathcal{L}_{content} + 10^{-3}\mathcal{L}_{generator} \tag{5}$$

Instead of training the super-resolution model i.e. the generator from scratch in a GAN, they pre-train it with a pixel-wise loss and fine-tune the model with a perceptual loss. The SRGAN paper uses *SRResNet* as super-resolution model, a predecessor of EDSR.

I found in experiments that the SRGAN approach also works very well for fine-tuning EDSR and WDSR models. The following example fine-tunes an EDSR baseline model that was pre-trained with a pixel-wise L^1 loss. The definition of the SRGAN discriminator is here.

NOTE: Content, Generator, Discriminator, and Perceptual losses are **NOT** implemented in TensorFlow 2.0 yet, but :) who knows **maybe Francois Chollet works on them right now?**

SRGAN

GANs (Generative Adversarial Networks) are class of AI algorithms used in Unsupervised Machine Learning.

GANs are DNN architectures comprised of two networks (Generator and Discriminator) pitting one against the other (thus the “adversarial”).

GANs is about creating, like drawing a portrait or composing a symphony. The main focus for GANs is to generate data from scratch.

In ML, there are two main classes of models:

- generative and
- discriminative.

A discriminative model is one that discriminates between two (or more) different classes of data — for example a convolutional neural network that is trained to output 1 given an image of a car and 0 otherwise. A generative model on the other hand doesn’t know anything about classes of data. Instead, its purpose is to generate new data which fits the distribution of the training data.

GANs consist of a Generator and Discriminator. Think it like a game where Generator tries to produce some data from probability distribution and Discriminator acts like a judge. Discriminator decides whether input is coming from true training data set of fake generated data. Generator tries to optimize data so that it can match true training data. Or we can say

discriminator is guiding generator to produce realistic data. They just work like encoder and decoder.

Let make it easy by taking an example-

Lets say we want to generate animated characters face. So we will provide training data which consists of images of anime character faces. And fake data consists of some random noise. Now Generator will try to produce image from noise which will be judged by discriminator. Both will keep training so that generator can generate images which can match true training data. One interesting thing is, images generated by generator will have features from original training data images but may or may not be the same. So like this we can generate some anime faces with heterogeneous mix of features from training data.

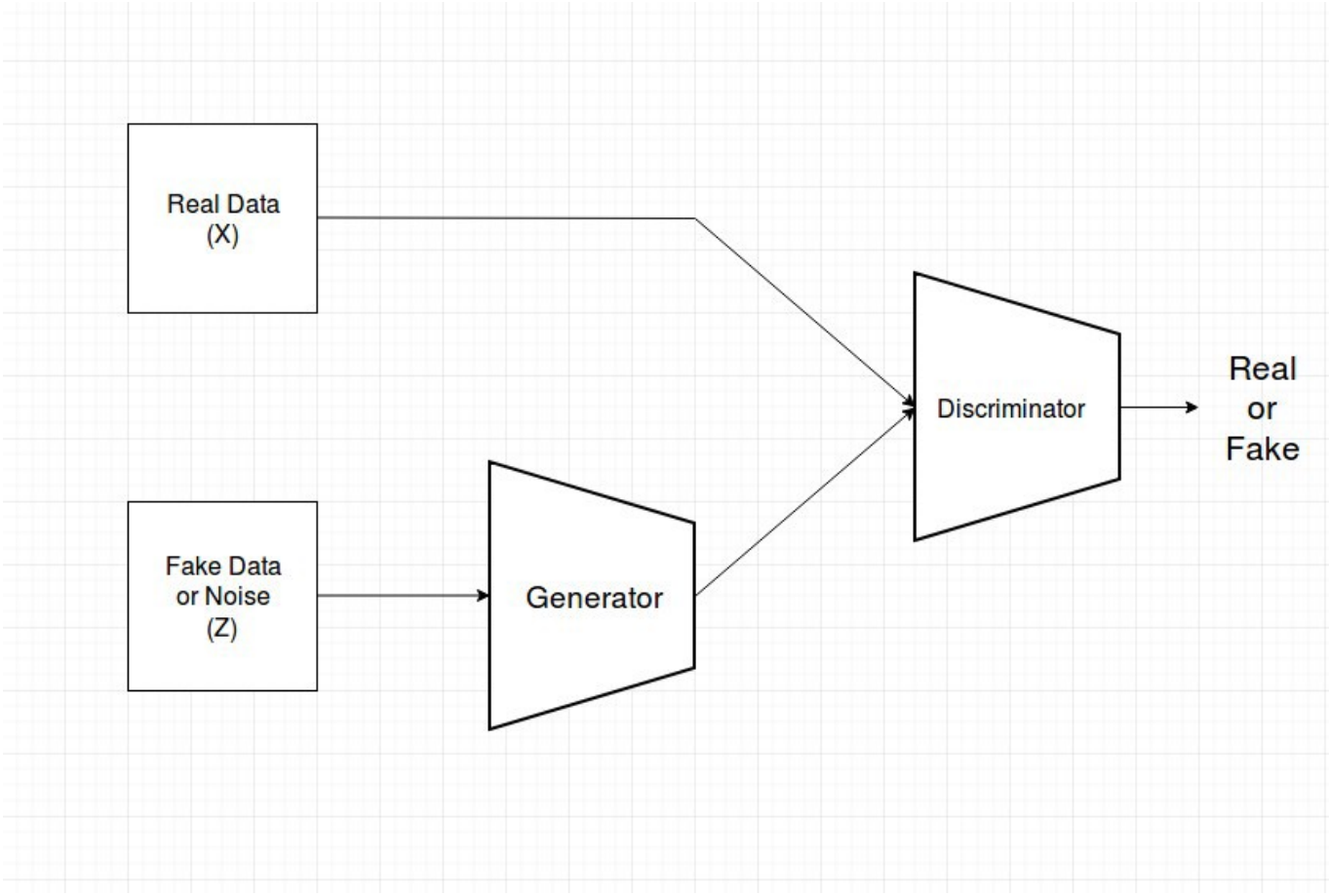


Figure 7: GANs basic architecture

Discriminator and Generator are both learning at the same time, and once Generator is trained it knows enough about the distribution of the training samples so that it can now generate new samples which share very similar properties.

SRGAN is the **Super Resolution Generative Adversarial Network.**

Idea Behind SRGAN : We have seen various ways for Single image super resolution. Those ways are fast and accurate as well. But still there is one problem which is not solved. That is, how can we recover finer texture details from low resolution image so that image is not distorted. Recent work has largely focused on minimizing the mean squared reconstruction error. The results have high peak signal-to-noise ratios(PSNR) means we have good image quality results, but

they are often lacking high-frequency details and are perceptually unsatisfying as they are not able to match the fidelity expected in high resolution images. Previous ways try to see similarity in pixel space which led to perceptually unsatisfying results or they produce blurry images. So we need a stable model which can capture the perceptual differences between the model's output and the ground truth image.

To achieve this we will use Perceptual loss function which comprise of Content and Adversarial loss. Other then that SRGAN uses residual blocks for deep neural network.

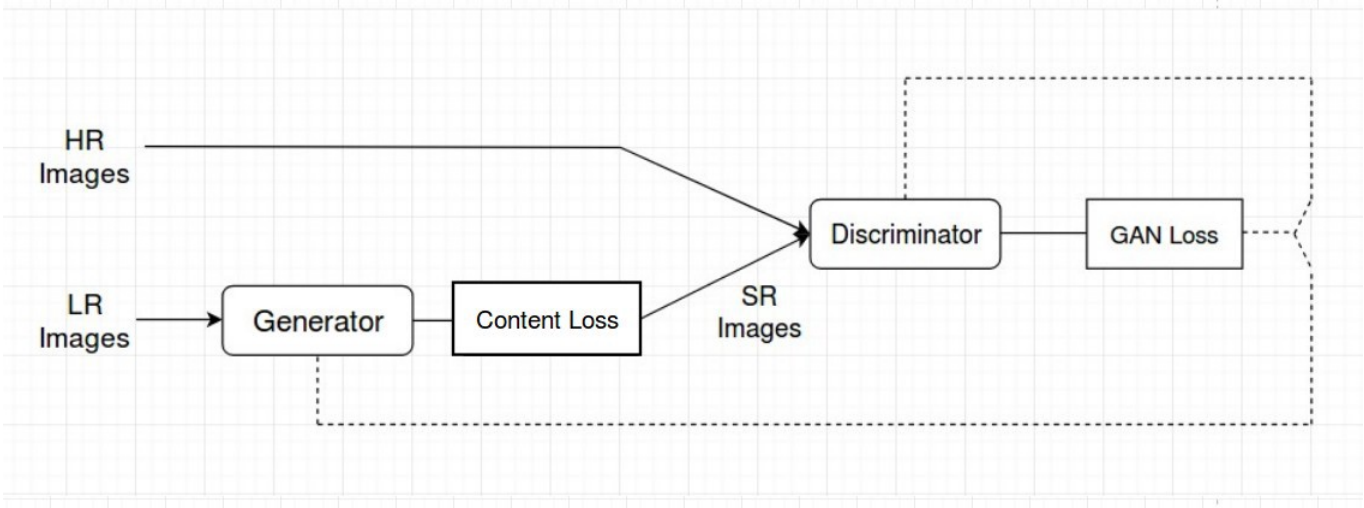


Figure 8. SRGAN Architecture

Now lets go further into details about SRGAN : Super-resolution GAN applies a deep network in combination with an adversary network to produce higher resolution images.

Training procedure is shown in following steps:

- process the HR(High Resolution) images to get down-sampled LR(Low Resolution) images. Now we have both HR and LR images for training data set.
- pass LR images through Generator which up-samples and gives SR(Super Resolution) images.
- use a discriminator to distinguish the HR images and back-propagate the GAN loss to train the discriminator and the generator.

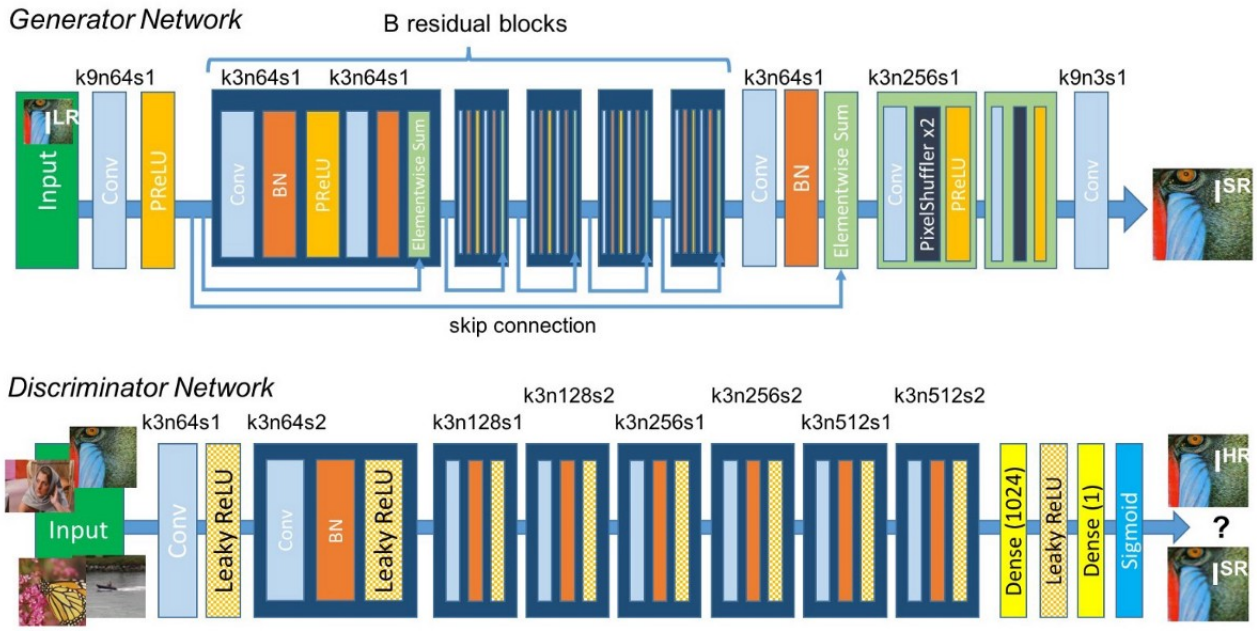


Figure 9. Generator and Discriminator Network

Above is the network design for the generator and the discriminator. It mostly composes of convolution layers, batch normalization and parameterized ReLU (PRelU). The generator also implements skip connections similar to ResNet.

Few things to note from Network architecture:

- Residual blocks: Since deeper networks are more difficult to train. The residual learning framework eases the training of these networks, and enables them to be substantially deeper, leading to improved performance. More about Residual blocks and Deep Residual learning can be found in paper given below. 16 residual blocks are used in Generator.
- PixelShuffler x2: This is feature map upscaling. 2 sub-pixel CNN are used in Generator. Upscaling or Upsampling are same. There are various ways to do that. In code keras inbuilt function has been used.
- PRelu(Parameterized Relu): We are using PRelu in place of Relu or LeakyRelu. It introduces learn-able parameter that makes it possible to adaptively learn the negative part coefficient.
- k3n64s1 this means kernel 3, channels 64 and strides 1.

Loss Functions: This is most important part:

- Perceptual loss includes Content (Reconstruction) loss and Adversarial loss (Figure 10).

$$l^{SR} = \underbrace{l_{\mathrm{X}}^{SR}}_{\text{content loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{adversarial loss}}$$

perceptual loss (for VGG based content losses)

Figure 10. Perceptual loss.

- Adversarial loss pushes the solution to the natural image manifold using a discriminator network that is trained to differentiate between the super-resolved images and original photo-realistic images.

$$l_{Gen}^{SR} = \sum_{n=1}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR}))$$

Figure 11. Adversarial loss.

- Content Loss is used so to keep perceptual similarity instead of pixel wise similarity. This will allow us to recover photo-realistic textures from heavily down sampled images. Instead of relying on pixel-wise losses we will and use a loss function that is closer to perceptual similarity.

$$l_{VGG/i.j}^{SR} = \frac{1}{W_{i,j}H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

Figure 12. Content loss.

NOTE: Here we define the VGG loss based on the ReLU activation layers of the per-trained 19 layer VGG network. VGG loss is defined as the euclidean distance between the feature representations of a reconstructed image and the reference image.

SRGAN uses a perceptual loss measuring the MSE of features extracted by a VGG-19 network. For a specific layer within VGG-19, we want their features to be matched (Minimum MSE for features).

NOTE. More about perceptual loss:

- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network, https://arxiv.org/pdf/1609.04802.pdf
- Perceptual Losses for Real-Time Style Transferand Super-Resolution, https://arxiv.org/pdf/1603.08155.pdf
- Deep Residual Learning for Image Recognition, https://arxiv.org/pdf/1512.03385.pdf

SRGAN with EDSR as a Generator

```
%%time

from model import srgan

# Used in content_loss
mean_squared_error = tf.keras.losses.MeanSquaredError()

# Used in generator_loss and discriminator_loss
binary_cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=False)

# Model that computes the feature map after the 4th convolution
# before the 5th max-pooling layer in VGG19. This is layer 20 in
# the corresponding Keras model.
vgg = srgan.vgg_54()

# EDSR model used as generator in SRGAN
generator = edsr(scale=4, num_res_blocks=16)
generator.load_weights(os.path.join(weights_dir, 'weights-edsr-16-x4.h5'))

# SRGAN discriminator
discriminator = srgan.discriminator()

# Optmizers for generator and discriminator. SRGAN will be trained for
# 200,000 steps and learning rate is reduced from 1e-4 to 1e-5 after
# 100,000 steps
schedule = PiecewiseConstantDecay(boundaries=[100000], values=[1e-4, 1e-5])
generator_optimizer = Adam(learning_rate=schedule)
discriminator_optimizer = Adam(learning_rate=schedule)

def generator_loss(sr_out):
    return binary_cross_entropy(tf.ones_like(sr_out), sr_out)

def discriminator_loss(hr_out, sr_out):
    hr_loss = binary_cross_entropy(tf.ones_like(hr_out), hr_out)
    sr_loss = binary_cross_entropy(tf.zeros_like(sr_out), sr_out)
    return hr_loss + sr_loss

@tf.function
def content_loss(hr, sr):
    sr = tf.keras.applications.vgg19.preprocess_input(sr)
    hr = tf.keras.applications.vgg19.preprocess_input(hr)
    sr_features = vgg(sr) / 12.75
    hr_features = vgg(hr) / 12.75
    return mean_squared_error(hr_features, sr_features)

@tf.function
def train_step(lr, hr):
    """SRGAN training step.

    Takes an LR and an HR image batch as input and returns
    the computed perceptual loss and discriminator loss.
    """
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
```

```
        lr = tf.cast(lr, tf.float32)
        hr = tf.cast(hr, tf.float32)

        # Forward pass
        sr = generator(lr, training=True)
        hr_output = discriminator(hr, training=True)
        sr_output = discriminator(sr, training=True)

        # Compute losses
        con_loss = content_loss(hr, sr)
        gen_loss = generator_loss(sr_output)
        perc_loss = con_loss + 0.001 * gen_loss
        disc_loss = discriminator_loss(hr_output, sr_output)

    # Compute gradient of perceptual loss w.r.t. generator weights
    gradients_of_generator = gen_tape.gradient(perc_loss, generator.trainable_vari
    # Compute gradient of discriminator loss w.r.t. discriminator weights
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.train

    # Update weights of generator and discriminator
    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trai
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discri

    return perc_loss, disc_loss

pls_metric = tf.keras.metrics.Mean()
dls_metric = tf.keras.metrics.Mean()

#steps = 200000
steps = 200
step = 0

# Train SRGAN for 200,000 steps.
for lr, hr in train_ds.take(steps):
    step += 1

    pl, dl = train_step(lr, hr)
    pls_metric(pl)
    dls_metric(dl)

    if step % 50 == 0:
        print(f'{step}/{steps}, perceptual loss = {pls_metric.result():.4f}, discr
        pls_metric.reset_states()
        dls_metric.reset_states()

generator.save_weights(os.path.join(weights_dir, 'weights-edsr-16-x4-fine-tuned.h5
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applica
80142336/80134624 [==============================] - 2s 0us/step
WARNING:tensorflow:AutoGraph could not transform <function train_step at 0x7f
Please report this to the TensorFlow team. When filing the bug, set the verbo
Cause: Unable to locate the source code of <function train_step at 0x7f119161
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function train_step at 0x7f
Please report this to the TensorFlow team. When filing the bug, set the verbo
```

```
Cause: Unable to locate the source code of <function train_step at 0x7f11916f
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function train_step at 0x7f11916f2950
Please report this to the TensorFlow team. When filing the bug, set the verbo
Cause: Unable to locate the source code of <function train_step at 0x7f11916f
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function content_loss at 0x
Please report this to the TensorFlow team. When filing the bug, set the verbo
Cause: Unable to locate the source code of <function content_loss at 0x7f1191
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING:tensorflow:AutoGraph could not transform <function content_loss at 0x
Please report this to the TensorFlow team. When filing the bug, set the verbo
Cause: Unable to locate the source code of <function content_loss at 0x7f1191
To silence this warning, decorate the function with @tf.autograph.experimenta
WARNING: AutoGraph could not transform <function content_loss at 0x7f11916f28
Please report this to the TensorFlow team. When filing the bug, set the verbo
Cause: Unable to locate the source code of <function content_loss at 0x7f1191
To silence this warning, decorate the function with @tf.autograph.experimenta
50/200, perceptual loss = 0.1957, discriminator loss = 0.3814
100/200, perceptual loss = 0.1761, discriminator loss = 0.3172
150/200, perceptual loss = 0.1959, discriminator loss = 0.4148
200/200, perceptual loss = 0.1742, discriminator loss = 0.8254
CPU times: user 47.5 s, sys: 23 s, total: 1min 10s
Wall time: 9min 34s
```

SRGAN with WDSR-B as a Generator ... NOT IMPLEMENTED HERE!

Let's try it as a self-guided learning exercise.

It's full implementation can be found at https://github.com/krasserm/super-resolution

```
%%time
# This is just TEMPLATE - edit the code below by the previous example of EDSR as a

# WDSR B model used as generator in SRGAN
generator = wdsr_b(scale=4, num_res_blocks=32)
generator.load_weights(os.path.join(weights_dir, 'weights-wdsr-b-32-x4.h5'))
# Run SRGAN training ...
generator.save_weights(os.path.join(weights_dir, 'weights-wdsr-b-32-x4-fine-tuned.
```

Part 3.Validation of Pre-Trained Models

If you didn't run training code in the previous section, download model weights here and extract the downloaded archive. The trained EDSR model can now be used to create SR images from LR images. One can clearly see how fine-tuning with a perceptual loss creates more realistic textures in SR images compared to training with a pixel-wise loss alone.

Dataset - Validation

```
%%time
from data import DIV2K
div2k_valid = DIV2K(scale=4, subset='valid', downgrade='bicubic')

# Version of validation dataset with 100 FULL-size images + WITHOUT data augmentat
# NOTE: batch = 1 (only! because images are different by sizes), repeat_count=1 (o
valid_ds = div2k_valid.dataset(batch_size=1, random_transform=False, repeat_count=
```

```
Downloading data from http://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_valid_LR_
31506432/31505881 [==============================] - 5s 0us/step
Caching decoded images in .div2k/caches/DIV2K_valid_LR_bicubic_X4.cache ...
Cached decoded images in .div2k/caches/DIV2K_valid_LR_bicubic_X4.cache.
Downloading data from http://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_valid_HR.
448995328/448993893 [==============================] - 55s 0us/step
Caching decoded images in .div2k/caches/DIV2K_valid_HR.cache ...
Cached decoded images in .div2k/caches/DIV2K_valid_HR.cache.
CPU times: user 21.2 s, sys: 4.06 s, total: 25.2 s
Wall time: 1min 23s
```

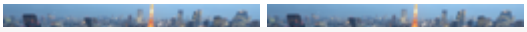
```
%%time
# Alternative way to dataset - VERY LONG for the 1st attempt - it is used here for
import tensorflow_datasets as tfds

valid_ds, valid_info = tfds.load('Div2k', split='validation', with_info=True)
```

```
CPU times: user 38 ms, sys: 1.34 ms, total: 39.3 ms
Wall time: 39.4 ms
```

```
# Show examples of images
import matplotlib.pyplot as plt

tfds.as_dataframe(valid_ds.take(4), valid_info)
```


```
%%time
# Version of validation dataset with 96x96 segments + their data augmentation
# NOTE: batch = 16, repeat_count=1600
valid_ds_transformed = div2k_valid.dataset(batch_size=16, random_transform=True, r
```

Import Libraries



```
%%time

from model import srgan
import numpy as np
import tensorflow as tf
import os
import pandas as pd
import matplotlib.pyplot as plt
```

```
CPU times: user 1.55 s, sys: 264 ms, total: 1.82 s
Wall time: 1.72 s
```

Download Model Weights

```
! wget https://martin-krasser.de/sisr/weights-srgan.tar.gz
```

```
--2021-04-21 06:44:20--  https://martin-krasser.de/sisr/weights-srgan.tar.gz
Resolving martin-krasser.de (martin-krasser.de)... 217.160.0.142, 2001:8d8:10
Connecting to martin-krasser.de (martin-krasser.de)|217.160.0.142|:443... cor
HTTP request sent, awaiting response... 200 OK
Length: 99033507 (94M) [application/gzip]
Saving to: ‘weights-srgan.tar.gz’

weights-srgan.tar.g 100%[===================>]  94.45M  6.97MB/s    in 14s

2021-04-21 06:44:35 (6.74 MB/s) - ‘weights-srgan.tar.gz’ saved [99033507/9903
```

```
! tar -zxvf weights-srgan.tar.gz
```

```
weights/srgan/gan_discriminator.h5
weights/srgan/gan_generator.h5
weights/srgan/pre_generator.h5
```

```
! ls -all weights/srgan/
```

```
total 104884
```

```
drwxr-xr-x 2 root root     4096 Apr 21 06:44 .
drwxr-xr-x 3 root root     4096 Apr 21 06:44 ..
-rw-r--r-- 1 1000 1000 94349016 Jun 13  2019 gan_discriminator.h5
-rw-r--r-- 1 1000 1000  6519624 Jun 13  2019 gan_generator.h5
-rw-r--r-- 1 1000 1000  6519624 Jun 13  2019 pre_generator.h5
```

```
from model.srgan import generator, discriminator
from train import SrganTrainer, SrganGeneratorTrainer

weights_dir = 'weights/srgan/'
```

```
pre_trainer = SrganGeneratorTrainer(model=generator(), checkpoint_dir=f'.ckpt/pre_
pre_trainer.model.load_weights(os.path.join(weights_dir, 'pre_generator.h5'))
```

```
ft_gan_trainer = SrganGeneratorTrainer(model=generator(), checkpoint_dir=f'.ckpt/p
ft_gan_trainer.model.load_weights(os.path.join(weights_dir, 'gan_generator.h5'))
```

```
gan_trainer = SrganTrainer(generator=generator(), discriminator=discriminator())

gan_trainer.generator.load_weights(os.path.join(weights_dir, 'gan_generator.h5'))
gan_trainer.discriminator.load_weights(os.path.join(weights_dir, 'gan_discriminato
```

Metrics (PSNR, SSIM, ...) -> Definiton

Old Metrics - Classic

```
def resolve(model, lr_batch):
    lr_batch = tf.cast(lr_batch, tf.float32)
    sr_batch = model(lr_batch)
    sr_batch = tf.clip_by_value(sr_batch, 0, 255)
    sr_batch = tf.round(sr_batch)
    sr_batch = tf.cast(sr_batch, tf.uint8)
    return sr_batch


def evaluate_psnr(model, dataset):
    psnr_values = []
    for lr, hr in dataset:
        sr = resolve(model, lr)
        psnr_value = psnr(hr, sr)[0]
        psnr_values.append(psnr_value)
    return tf.reduce_mean(psnr_values)

def psnr(x1, x2):
    return tf.image.psnr(x1, x2, max_val=255)


def evaluate_ssim(model, dataset):
```

```
    ssim_values = []
    for lr, hr in dataset:
        sr = resolve(model, lr)
        ssim_value = ssim(hr, sr)[0]
        ssim_values.append(ssim_value)
    return tf.reduce_mean(ssim_values)

def ssim(x1, x2):
    return tf.image.ssim(x1, x2, max_val=255)



def evaluate_ssim_multiscale(model, dataset):
    ssim_multiscale_values = []
    for lr, hr in dataset:
        sr = resolve(model, lr)
        ssim_multiscale_value = ssim_multiscale(hr, sr)[0]
        ssim_multiscale_values.append(ssim_multiscale_value)
    return tf.reduce_mean(ssim_multiscale_values)

# Use filter_size<7 because of cropped segments 96x96!
def ssim_multiscale(x1, x2):
    return tf.image.ssim_multiscale(x1, x2, max_val=255, filter_size=6)
```

Stastical Data - Functions

```
def evaluate_psnr_stat(dataset, model, dataset_label, model_label, samples_number)
  metric_list = []
  for i in range(samples_number):
    dataset_shuffled = dataset.shuffle(samples_number)
    metric_value = evaluate_psnr(model, dataset_shuffled.take(1))
    metric_list.append(metric_value)

  metric_mean = np.mean(metric_list)
  metric_stdev = np.std(metric_list)
  print(f'{model_label} on {dataset_label} -> PSNR = {metric_mean:3f}(+-{metric_st
  return metric_mean, metric_stdev
```

```
def evaluate_ssim_stat(dataset, model, dataset_label, model_label, samples_number)
  metric_list = []
  for i in range(samples_number):
    dataset_shuffled = dataset.shuffle(samples_number)
    metric_value = evaluate_ssim(model, dataset_shuffled.take(1))
    metric_list.append(metric_value)

  metric_mean = np.mean(metric_list)
  metric_stdev = np.std(metric_list)
  print(f'{model_label} on {dataset_label} -> SSIM = {metric_mean:3f}(+-{metric_st
  return metric_mean, metric_stdev
```

```
def evaluate_ssim_multiscale_stat(dataset, model, dataset_label, model_label, samp
  metric_list = []
  for i in range(samples_number):
    dataset_shuffled = dataset.shuffle(samples_number)
    metric_value = evaluate_ssim_multiscale(model, dataset_shuffled.take(1))
    metric_list.append(metric_value)

  metric_mean = np.mean(metric_list)
  metric_stdev = np.std(metric_list)
  print(f'{model_label} on {dataset_label} -> MS-SSIM = {metric_mean:3f}(+-{metric

  return metric_mean, metric_stdev
```

New Metrics -> Perception (and other) losses

```
from model import srgan

# Used in content_loss
mean_squared_error = tf.keras.losses.MeanSquaredError()

# Used in generator_loss and discriminator_loss
binary_cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=False)

# Model that computes the feature map after the 4th convolution
# before the 5th max-pooling layer in VGG19. This is layer 20 in
# the corresponding Keras model.
vgg = srgan.vgg_54()

def generator_loss(sr_out):
    return binary_cross_entropy(tf.ones_like(sr_out), sr_out)

def discriminator_loss(hr_out, sr_out):
    hr_loss = binary_cross_entropy(tf.ones_like(hr_out), hr_out)
    sr_loss = binary_cross_entropy(tf.zeros_like(sr_out), sr_out)
    return hr_loss + sr_loss

@tf.function
def content_loss(hr, sr):
    sr = tf.keras.applications.vgg19.preprocess_input(sr)
    hr = tf.keras.applications.vgg19.preprocess_input(hr)
    sr_features = vgg(sr) / 12.75
    hr_features = vgg(hr) / 12.75
    return mean_squared_error(hr_features, sr_features)
```

```
from tqdm import tqdm

def evaluate_perc_loss(generator, discriminator, dataset):
    con_loss_values = []
    gen_loss_values = []
    disc_loss_values = []
    perc_loss_values = []
```

```
    for lr, hr in dataset:
        lr = tf.cast(lr, tf.float32)
        hr = tf.cast(hr, tf.float32)

        # Forward pass
        sr = generator(lr)
        hr_output = discriminator(hr)
        sr_output = discriminator(sr)

        # Compute losses
        con_loss = content_loss(hr, sr)
        gen_loss = generator_loss(sr_output)
        perc_loss = con_loss + 0.001 * gen_loss
        disc_loss = discriminator_loss(hr_output, sr_output)

        con_loss_values.append(con_loss)
        gen_loss_values.append(gen_loss)
        disc_loss_values.append(disc_loss)
        perc_loss_values.append(perc_loss)

    return np.mean(con_loss_values), np.std(con_loss_values), np.mean(gen_loss_val
```

Pre-Trained Version

Metrics -> Run

PSNR - GPU Tesla K80 - 1st ("heating") iteration

Just to see the difference between 1st ("heating") and 2nd iterations.

```
%%time
# CPU
# Pre-Trained -> PSNR = 29.406034
# CPU times: user 43.4 s, sys: 1.37 s, total: 44.8 s
# Wall time: 24 s

# GPU - Tesla K80 - 1st iteration
# Pre-Trained -> PSNR = 29.406033
# CPU times: user 3.96 s, sys: 3.32 s, total: 7.29 s
# Wall time: 34.2 s

# Evaluate model on full validation set
pt_psnrv = pre_trainer.evaluate(valid_ds.take(1))
print(f'Pre-Trained -> PSNR = {pt_psnrv.numpy():3f}')
```

```
Pre-Trained -> PSNR = 29.406033
CPU times: user 3.96 s, sys: 3.32 s, total: 7.29 s
Wall time: 34.2 s
```

PSNR - GPU Tesla K80 - 2nd iteration

FULL-size images - valid_ds

```
%%time
# GPU - 100 FULL images - NOT RUN YET! - NO available GPUs :)

psnr_full_pre_mean, psnr_full_pre_metric_stdev = evaluate_psnr_stat(dataset=valid_
                              dataset_label='FULL', model_label='Pre-Trained', sam
```

```
%%time
# CPU - 10 FULL images
# Pre-Trained on FULL -> PSNR = 29.848770(+-4.914822)
# CPU times: user 7min 53s, sys: 11.5 s, total: 8min 4s
# Wall time: 4min 16s

psnr_full_pre_mean, psnr_full_pre_metric_stdev = evaluate_psnr_stat(dataset=valid_
                              dataset_label='FULL', model_label='Pre-Trained', sam
```

```
Pre-Trained on FULL -> PSNR = 29.848770(+-4.914822)
CPU times: user 7min 53s, sys: 11.5 s, total: 8min 4s
Wall time: 4min 16s
```

Cropped (96x96) RANDOM segments of images - valid_ds_transformed

```
%%time
# GPU - 1000 segments - NOT RUN YET! - NO available GPUs :)

psnr_cropped_pre_mean, psnr_cropped_pre_metric_stdev = evaluate_psnr_stat(dataset=
                              dataset_label='CROPPED', model_label='Pre-Trained',
```

```
%%time
# CPU - 1000 segments
# Pre-Trained on CROPPED -> PSNR = 32.418598(+-8.176866)
# CPU times: user 11min 14s, sys: 5min 23s, total: 16min 37s
# Wall time: 13min 18s

psnr_cropped_pre_mean, psnr_cropped_pre_metric_stdev = evaluate_psnr_stat(dataset=
                              dataset_label='CROPPED', model_label='Pre-Trained',
```

```
Pre-Trained on CROPPED -> PSNR = 32.418598(+-8.176866)
CPU times: user 11min 14s, sys: 5min 23s, total: 16min 37s
Wall time: 13min 18s
```

SSIM

FULL

```
%%time
# GPU - 10 FULL images - NOT RUN YET! - NO available GPUs :)# GPU - 10 FULL images

ssim_full_pre_mean, ssim_full_pre_metric_stdev = evaluate_ssim_stat(dataset=valid_
                              dataset_label='FULL', model_label='Pre-Trained', sam
```

```
%%time
# CPU - 10 FULL images
# Pre-Trained on FULL -> SSIM = 0.877528(+-0.056733)
# CPU times: user 15min 20s, sys: 12.2 s, total: 15min 32s
# Wall time: 8min 5s

ssim_full_pre_mean, ssim_full_pre_metric_stdev = evaluate_ssim_stat(dataset=valid_
                              dataset_label='FULL', model_label='Pre-Trained', sam
```

```
    Pre-Trained on FULL -> SSIM = 0.877528(+-0.056733)
    CPU times: user 15min 20s, sys: 12.2 s, total: 15min 32s
    Wall time: 8min 5s
```

Cropped

```
%%time
# GPU - 100 segments - NOT RUN YET! - NO available GPUs :)

ssim_cropped_pre_mean, ssim_cropped_pre_metric_stdev = evaluate_ssim_stat(dataset=
                              dataset_label='CROPPED', model_label='Pre-Trained',
```

```
%%time
# CPU - 100 segments
# Pre-Trained on CROPPED -> SSIM = 0.840822(+-0.140440)
# CPU times: user 1min 9s, sys: 32.2 s, total: 1min 42s
# Wall time: 1min 23s

ssim_cropped_pre_mean, ssim_cropped_pre_metric_stdev = evaluate_ssim_stat(dataset=
                              dataset_label='CROPPED', model_label='Pre-Trained',
```

```
    Pre-Trained on CROPPED -> SSIM = 0.840822(+-0.140440)
    CPU times: user 1min 9s, sys: 32.2 s, total: 1min 42s
    Wall time: 1min 23s
```

MS-SIM

FULL

```
%%time
# GPU Tesla K80 - 10 FULL images
# Pre-Trained on FULL -> MS-SSIM = 0.958319(+-0.023828)
# CPU times: user 7.42 s, sys: 5.77 s, total: 13.2 s
```

```
# Wall time: 13.2 s

ssim_ms_full_pre_mean, ssim_ms_full_pre_metric_stdev = evaluate_ssim_multiscale_st
                                dataset_label='FULL', model_label='Pre-Trained', sam
```

```
%%time
# CPU - 10 FULL images
# Pre-Trained on FULL -> MS-SSIM = 0.954331(+-0.034239)
# CPU times: user 10min 18s, sys: 15.4 s, total: 10min 34s
# Wall time: 5min 34s

ssim_ms_full_pre_mean, ssim_ms_full_pre_metric_stdev = evaluate_ssim_multiscale_st
                                dataset_label='FULL', model_label='Pre-Trained', sam
```

```
    Pre-Trained on FULL -> MS-SSIM = 0.969010(+-0.015526)
    CPU times: user 12min 19s, sys: 11.9 s, total: 12min 31s
    Wall time: 6min 36s
```

▼ Cropped

```
%%time
# GPU Tesla K80 - 100 segments

#Pre-Trained on CROPPED -> MS-SSIM = 0.965447(+-0.006500)
#CPU times: user 46 s, sys: 31.8 s, total: 1min 17s
#Wall time: 1min 5s

evaluate_ssim_multiscale_stat(dataset=valid_ds_transformed, model=pre_trainer.mode
                                dataset_label='CROPPED', model_label='Pre-Trained',
```

```
    Pre-Trained on CROPPED -> MS-SSIM = 0.965447(+-0.006500)
    CPU times: user 46 s, sys: 31.8 s, total: 1min 17s
    Wall time: 1min 5s
```

```
%%time
# CPU - 100 segments
# Pre-Trained on CROPPED -> MS-SSIM = 0.957531(+-0.038893)
# CPU times: user 1min 8s, sys: 32 s, total: 1min 40s
# Wall time: 1min 21s

ssim_ms_cropped_pre_mean, ssim_ms_cropped_pre_metric_stdev = evaluate_ssim_multisc
                                dataset_label='CROPPED', model_label='Pre-Trained',
```

```
    Pre-Trained on CROPPED -> MS-SSIM = 0.957531(+-0.038893)
    CPU times: user 1min 8s, sys: 32 s, total: 1min 40s
    Wall time: 1min 21s
```

▼ Perceptual (and other) losses

▼ Cropped

(only possible due to GAN workflow on cropped images 96x96)

```
%%time
# generator = pre_trainer.model

# GPU Tesla K80 - 1000 segments

# GPU Tesla T4 - 1000 segments
# Content Loss = 0.087240 (+-0.025456)
# Generator Loss = 0.013794 (+-0.019842)
# Discriminant Loss = 5.630428 (+-0.461577)
# Perceptual Loss = 0.087254 (+-0.025457)
# CPU times: user 1min 49s, sys: 35.7 s, total: 2min 25s
# Wall time: 1min 55s

# Perceptual Loss -> Evaluate model on full validation set
con_loss_pre_mean, con_loss_pre_stdev, gen_loss_pre_mean, gen_loss_pre_stdev, disc
print(f'Content Loss = {con_loss_pre_mean:3f} (+-{con_loss_pre_stdev:3f})')
print(f'Generator Loss = {gen_loss_pre_mean:3f} (+-{gen_loss_pre_stdev:3f})')
print(f'Discriminant Loss = {disc_loss_pre_mean:3f} (+-{disc_loss_pre_stdev:3f})')
print(f'Perceptual Loss = {perc_loss_pre_mean:3f} (+-{perc_loss_pre_stdev:3f})')
```

```
Content Loss = 0.087240 (+-0.025456)
Generator Loss = 0.013794 (+-0.019842)
Discriminant Loss = 5.630428 (+-0.461577)
Perceptual Loss = 0.087254 (+-0.025457)
CPU times: user 1min 49s, sys: 35.7 s, total: 2min 25s
Wall time: 1min 55s
```

Fine Tuned Version

Metrics -> Run

PSNR

FULL-size images - valid_ds

```
%%time
# GPU - 100 FULL images - NOT RUN YET! - NO available GPUs :)

psnr_full_ft_gan_mean, psnr_full_ft_gan_metric_stdev = evaluate_psnr_stat(dataset=
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
%%time
# CPU - 10 FULL images
# Fine Tuned on FULL -> PSNR = 27.847717(+-4.068777)
```

```
# CPU times: user 7min 53s, sys: 13.9 s, total: 8min 7s
# Wall time: 4min 18s

psnr_full_ft_gan_mean, psnr_full_ft_gan_metric_stdev = evaluate_psnr_stat(dataset=
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
Fine Tuned on FULL -> PSNR = 27.847717(+-4.068777)
CPU times: user 7min 53s, sys: 13.9 s, total: 8min 7s
Wall time: 4min 18s
```

Cropped (96x96) RANDOM segments of images - valid_ds_transformed

```
%%time
# GPU - 1000 segments - NOT RUN YET! - NO available GPUs :)

psnr_cropped_ft_gan_mean, psnr_cropped_ft_gan_metric_stdev = evaluate_psnr_stat(da
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
%%time
# CPU - 1000 segments
# Fine Tuned on FULL -> PSNR = 27.765120(+-5.932281)
# CPU times: user 13min 5s, sys: 20min 42s, total: 33min 47s
# Wall time: 29min 17s

psnr_cropped_ft_gan_mean, psnr_cropped_ft_gan_metric_stdev = evaluate_psnr_stat(da
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
Fine Tuned on FULL -> PSNR = 27.765120(+-5.932281)
CPU times: user 13min 5s, sys: 20min 42s, total: 33min 47s
Wall time: 29min 17s
```

SSIM

FULL

```
%%time
# GPU - 10 FULL images - NOT RUN YET! - NO available GPUs :)

ssim_full_ft_gan_mean, ssim_full_ft_gan_metric_stdev = evaluate_ssim_stat(dataset=
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
%%time
# CPU - 10 FULL images
# Fine Tuned on FULL -> SSIM = 0.763317(+-0.062627)
# CPU times: user 15min 17s, sys: 18.6 s, total: 15min 36s
# Wall time: 8min 12s

ssim_full_ft_gan_mean, ssim_full_ft_gan_metric_stdev = evaluate_ssim_stat(dataset=
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
Pre-Trained on FULL -> SSIM = 0.763317(+-0.062627)
CPU times: user 15min 17s, sys: 18.6 s, total: 15min 36s
Wall time: 8min 12s
```

Cropped

```
%%time
# GPU - 100 segments - NOT RUN YET! - NO available GPUs :)

ssim_cropped_ft_gan_mean, ssim_cropped_ft_gan_metric_stdev = evaluate_ssim_stat(da
                                dataset_label='CROPPED', model_label='Fine Tuned', s
```

```
%%time
# CPU - 100 segments
# Fine Tuned on CROPPED -> SSIM = 0.733918(+-0.179958)
# CPU times: user 1min 23s, sys: 2min 11s, total: 3min 35s
# Wall time: 3min 7s

ssim_cropped_ft_gan_mean, ssim_cropped_ft_gan_metric_stdev = evaluate_ssim_stat(da
                                dataset_label='CROPPED', model_label='Fine Tuned', s
```

```
Fine Tuned on CROPPED -> SSIM = 0.733918(+-0.179958)
CPU times: user 1min 23s, sys: 2min 11s, total: 3min 35s
Wall time: 3min 7s
```

MS-SIM

FULL

```
%%time
# GPU - 10 images - NOT RUN YET! - NO available GPUs :)

ssim_ms_full_ft_gan_mean, ssim_ms_full_ft_gan_metric_stdev = evaluate_ssim_multisc
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
%%time
# CPU - 10 FULL images
# Fine Tuned on FULL -> MS-SSIM = 0.939168(+-0.012113)
# CPU times: user 11min 39s, sys: 11.9 s, total: 11min 51s
# Wall time: 6min 15s

ssim_ms_full_ft_gan_mean, ssim_ms_full_ft_gan_metric_stdev = evaluate_ssim_multisc
                                dataset_label='FULL', model_label='Fine Tuned', samp
```

```
Fine Tuned on FULL -> MS-SSIM = 0.939168(+-0.012113)
CPU times: user 11min 39s, sys: 11.9 s, total: 11min 51s
Wall time: 6min 15s
```

Cropped

```
%%time
# GPU Tesla K80 - 100 segments  - NOT RUN YET! - NO available GPUs :)

ssim_ms_cropped_ft_gan_mean, ssim_ms_cropped_ft_gan_metric_stdev = evaluate_ssim_m
                              dataset_label='CROPPED', model_label='Fine Tuned', s
```

```
%%time
# CPU - 100 segments
# Fine Tuned on CROPPED -> MS-SSIM = 0.932917(+-0.059809)
# CPU times: user 1min 23s, sys: 50.1 s, total: 2min 13s
# Wall time: 1min 48s

ssim_ms_cropped_ft_gan_mean, ssim_ms_cropped_ft_gan_metric_stdev = evaluate_ssim_m
                              dataset_label='CROPPED', model_label='Fine Tuned', s
```

```
Fine Tuned on CROPPED -> MS-SSIM = 0.932917(+-0.059809)
CPU times: user 1min 23s, sys: 50.1 s, total: 2min 13s
Wall time: 1min 48s
```

Perceptual (and other) losses

Cropped

(only possible due to GAN workflow on cropped images 96x96)

```
%%time
# generator = pre_trainer.model

# GPU Tesla K80 - 1000 segments

# GPU Tesla T4 - 1000 segments
# Content Loss = 0.066158 (+-0.020306)
# Generator Loss = 0.257390 (+-0.453378)
# Discriminant Loss = 4.137450 (+-0.392165)
# Perceptual Loss = 0.066416 (+-0.020319)
# CPU times: user 1min 49s, sys: 35.7 s, total: 2min 25s
# Wall time: 1min 55s

# Perceptual Loss -> Evaluate model on full validation set
con_loss_ft_gan_mean, con_loss_ft_gan_stdev, gen_loss_ft_gan_mean, gen_loss_ft_gan
print(f'Content Loss = {con_loss_ft_gan_mean:3f} (+-{con_loss_ft_gan_stdev:3f})')
print(f'Generator Loss = {gen_loss_ft_gan_mean:3f} (+-{gen_loss_ft_gan_stdev:3f})'
print(f'Discriminant Loss = {disc_loss_ft_gan_mean:3f} (+-{disc_loss_ft_gan_stdev:
print(f'Perceptual Loss = {perc_loss_ft_gan_mean:3f} (+-{perc_loss_ft_gan_stdev:3f
```

```
Content Loss = 0.066158 (+-0.020306)
Generator Loss = 0.257390 (+-0.453378)
Discriminant Loss = 4.137450 (+-0.392165)
```

```
Perceptual Loss = 0.066416 (+-0.020319)
CPU times: user 1min 49s, sys: 35.7 s, total: 2min 25s
Wall time: 1min 55s
```

Comparative Analysis

```
ix = pd.MultiIndex.from_arrays(
    [
     ["full", "full", "cropped", "cropped"],
     ["pre", "ft", "pre", "ft"],
     ],
     names=["dataset", "model"],
     )
```

```
ix
```

```
MultiIndex([(   'full', 'pre'),
            (   'full',  'ft'),
            ('cropped', 'pre'),
            ('cropped',  'ft')],
           names=['dataset', 'model'])
```

PSNR

```
means = pd.DataFrame(
    {
        "PSNR": [psnr_full_pre_mean, psnr_full_ft_gan_mean, psnr_cropped_pre_mean,
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
     },
     index=ix,
     )
```

```
means
```

		PSNR
dataset	**model**	
full	**pre**	29.848770
	ft	27.847717
cropped	**pre**	32.418598
	ft	27.765120

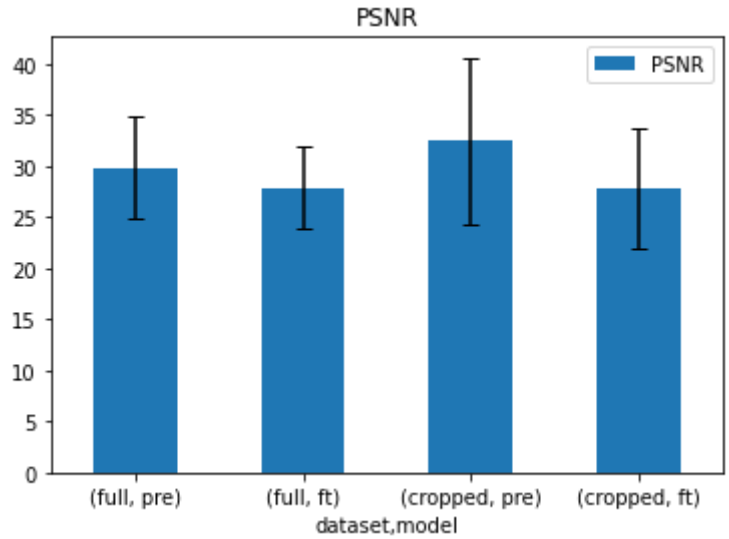
```
errors = pd.DataFrame(
    {
        "PSNR": [psnr_full_pre_metric_stdev, psnr_full_ft_gan_metric_stdev, psnr_c
```

```
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
    },
    index=ix,
    )
```

```
errors
```

		PSNR
dataset	**model**	
full	**pre**	4.914822
	ft	4.068777
cropped	**pre**	8.176866
	ft	5.932281

```
# Plot
fig, ax = plt.subplots()

plt.title('PSNR')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



SSIM

```
means = pd.DataFrame(
    {
        "SSIM": [ssim_full_pre_mean, ssim_full_ft_gan_mean, ssim_cropped_pre_mean,
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
    },
    index=ix,
    )
```

```

```

```
means
```

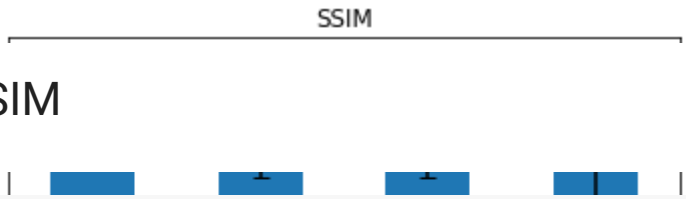
		SSIM
dataset	**model**	
full	**pre**	0.877528
	ft	0.763317
cropped	**pre**	0.840822
	ft	0.733918

```
errors = pd.DataFrame(
    {
        "SSIM": [ssim_full_pre_metric_stdev, ssim_full_ft_gan_metric_stdev, ssim_c
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
     },
     index=ix,
     )
```

```
errors
```

		SSIM
dataset	**model**	
full	**pre**	0.056733
	ft	0.062627
cropped	**pre**	0.140440
	ft	0.179958

```
# Plot
fig, ax = plt.subplots()

plt.title('SSIM')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



MS-SSIM


```
means = pd.DataFrame(
    {
        "MS-SSIM": [ssim_ms_full_pre_mean, ssim_ms_full_ft_gan_mean, ssim_ms_cropp
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
     },
     index=ix,
     )
```

```
means
```

		MS-SSIM
dataset	**model**	
full	**pre**	0.969010
	ft	0.939168
cropped	**pre**	0.957531
	ft	0.932917

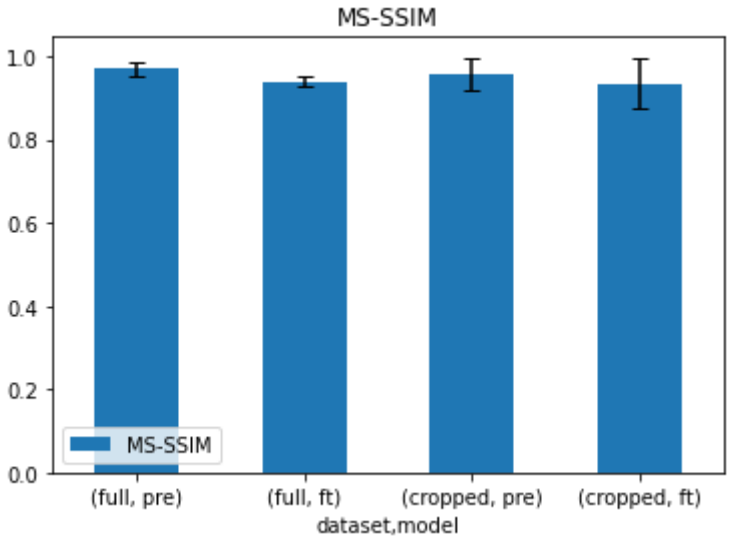
```
errors = pd.DataFrame(
    {
        "MS-SSIM": [ssim_ms_full_pre_metric_stdev, ssim_ms_full_ft_gan_metric_stde
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
     },
     index=ix,
     )
```

```
errors
```

		MS-SSIM
dataset	**model**	
full	**pre**	0.015526
	ft	0.012113
cropped	**pre**	0.038893
	ft	0.059809

```
# Plot
fig, ax = plt.subplots()
```

```
plt.title('MS-SSIM')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



GAN-related metrics

```
ix_new = pd.MultiIndex.from_arrays(
    [
     ["cropped", "cropped"],
     ["pre", "ft"],
     ],
     names=["dataset", "model"],
     )
```

```
ix_new
```

```
    MultiIndex([('cropped', 'pre'),
                ('cropped',  'ft')],
               names=['dataset', 'model'])
```

Content Loss

```
means = pd.DataFrame(
    {
        "Content Loss": [con_loss_pre_mean, con_loss_ft_gan_mean],
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
     },
     index=ix_new,
     )
```

```
means
```

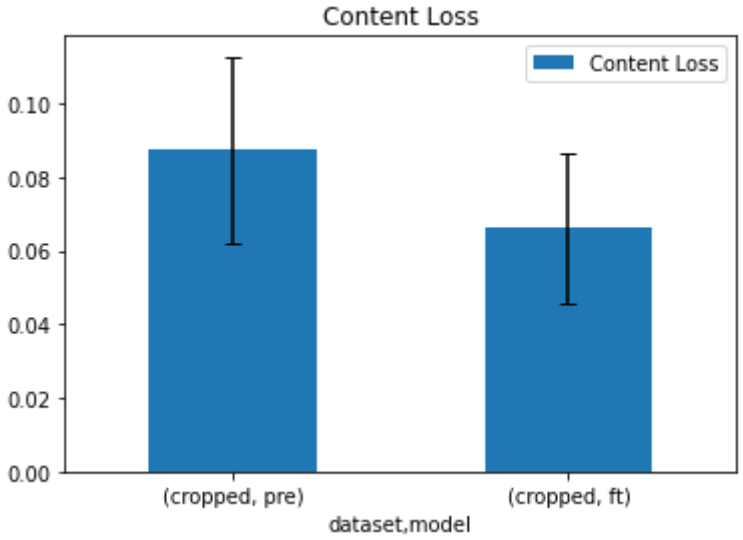
		Content Loss
dataset	model	

```
errors = pd.DataFrame(
    {
        "Content Loss": [con_loss_pre_stdev, con_loss_ft_gan_stdev],
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
     },
     index=ix_new,
     )
```

```
errors
```

		Content Loss
dataset	model	
cropped	pre	0.025456
	ft	0.020306

```
# Plot
fig, ax = plt.subplots()

plt.title('Content Loss')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



Generator Loss

```
means = pd.DataFrame(
    {
        "Generator Loss": [gen_loss_pre_mean, gen_loss_ft_gan_mean],
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
     },
```

```
    index=ix_new,
    )
```

```
means
```

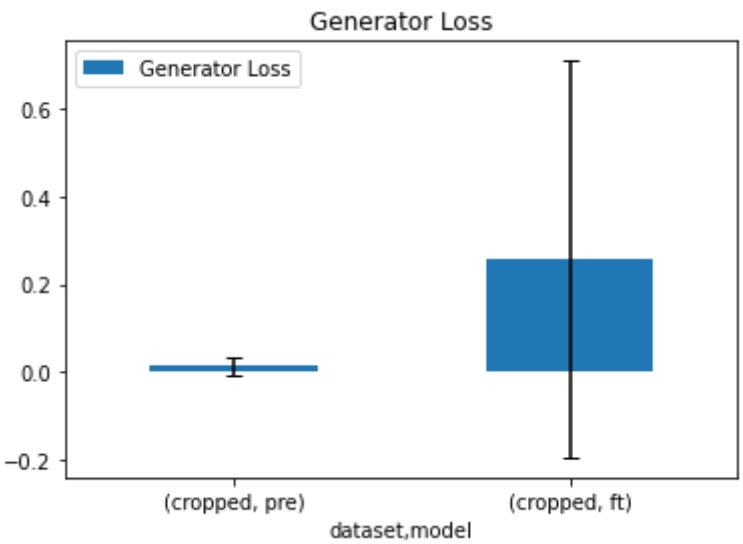
		Generator Loss
dataset	**model**	
cropped	**pre**	0.013794
	ft	0.257390

```
errors = pd.DataFrame(
    {
        "Generator Loss": [gen_loss_pre_stdev, gen_loss_ft_gan_stdev],
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
     },
     index=ix_new,
     )
```

```
errors
```

		Generator Loss
dataset	**model**	
cropped	**pre**	0.019842
	ft	0.453378

```
# Plot
fig, ax = plt.subplots()

plt.title('Generator Loss')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



Discriminator Loss

```
means = pd.DataFrame(
    {
        "Discriminator Loss": [disc_loss_pre_mean, disc_loss_ft_gan_mean],
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
     },
     index=ix_new,
     )
```

```
means
```

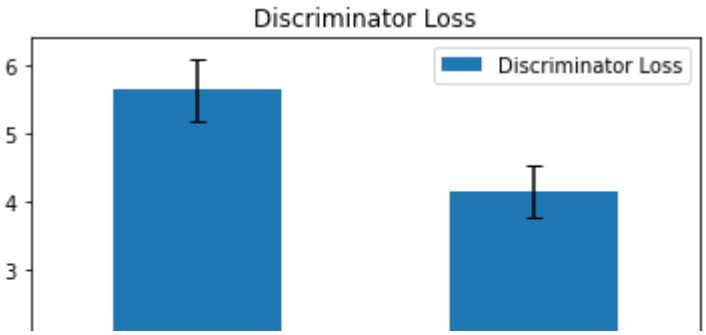
		Discriminator Loss
dataset	**model**	
cropped	**pre**	5.630428
	ft	4.137450

```
errors = pd.DataFrame(
    {
        "Discriminator Loss": [disc_loss_pre_stdev, disc_loss_ft_gan_stdev],
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
     },
     index=ix_new,
     )
```

```
errors
```

		Discriminator Loss
dataset	**model**	
cropped	**pre**	0.461577
	ft	0.392165

```
# Plot
fig, ax = plt.subplots()

plt.title('Discriminator Loss')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



Perceptual Loss

```
means = pd.DataFrame(
    {
        "Perceptual Loss": [perc_loss_pre_mean, perc_loss_ft_gan_mean],
        #"data2": [6, 5, 7, 5, 4, 5, 6, 5],
     },
     index=ix_new,
     )
```

```
means
```

		Perceptual Loss
dataset	**model**	
cropped	**pre**	0.087254
	ft	0.066416

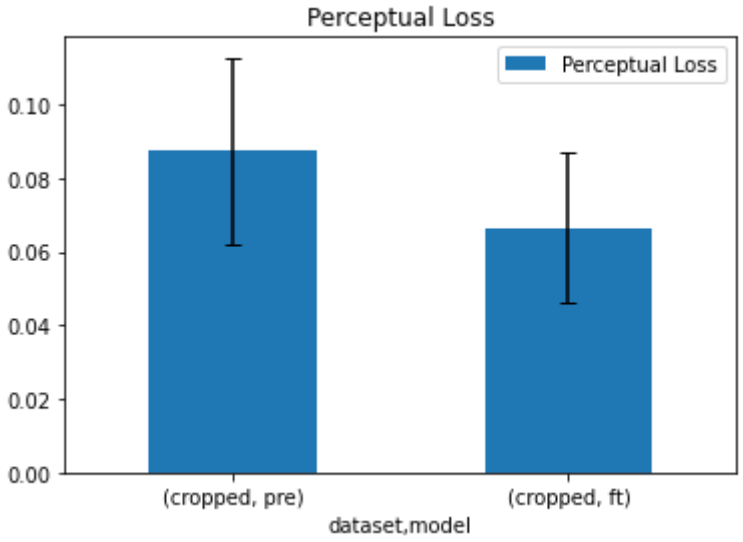
```
errors = pd.DataFrame(
    {
        "Perceptual Loss": [perc_loss_pre_stdev, perc_loss_ft_gan_stdev],
        #"data2": [0.6, 0.5, 0.7, 0.5, 0.4, 0.5, 0.6, 0.5],
     },
     index=ix_new,
     )
```

```
errors
```

		Perceptual Loss
dataset	**model**	
cropped	**pre**	0.025457
	ft	0.020319

```
# Plot
fig, ax = plt.subplots()
```

```
plt.title('Perceptual Loss')
means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```

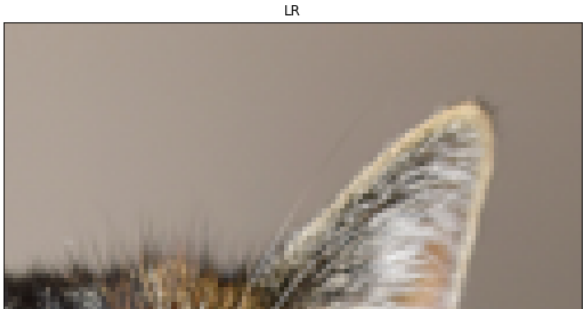


Visual Test

```
import os
import matplotlib.pyplot as plt

from model import resolve_single
from utils import load_image

%matplotlib inline

def resolve_and_plot(model_pre_trained, model_fine_tuned, lr_image_path):
    lr = load_image(lr_image_path)

    sr_pt = resolve_single(model_pre_trained, lr)
    sr_ft = resolve_single(model_fine_tuned, lr)

    plt.figure(figsize=(20, 20))

    model_name = model_pre_trained.name.upper()
    images = [lr, sr_pt, sr_ft]
    titles = ['LR', f'SR ({model_name}, pixel loss)', f'SR ({model_name}, perceptu
    positions = [1, 3, 4]

    for i, (image, title, position) in enumerate(zip(images, titles, positions)):
        plt.subplot(2, 2, position)
        plt.imshow(image)
        plt.title(title)
        plt.xticks([])
        plt.yticks([])

weights_dir = 'weights/article'
```

```
%%time

resolve_and_plot(pre_trainer.model, ft_gan_trainer.model, 'demo/0869x4-crop.png')
```

```
CPU times: user 8.45 s, sys: 137 ms, total: 8.58 s
Wall time: 4.71 s
```



Resume

The results of comparative analysis demonstrate the absence of ANY statistically reliable proofs (**by classic metrics like PSNR, SSIM, MS-SSIM**) about positive influence of "fine tuning by GAN" on image super resolution ... despite the vidsual better quality.

BUT ... :)

- the "new metrics" (**like perceptual, discriminator, content loss**) allow to conlcude that some trends of improvement can be oserved by decrease of their mean values,
- these "decrease" trends are in the limits of the standard deviation (**except for discriminator loss**) and cannot be statistically reliable proofs of GAN-related improvements.

Conclusions:

- at the moment **the discriminator loss only** (except for subjective visual tests) can be used as a statistically reliable proof for GAN-related improvements on the basis of VGG-like CNNs,
- in the future we need to use other "new metrics" (**like perceptual, discriminator, content loss**) to compare the quality of the new GAN-approaches on the basis of other CNN families (instead of VGG) like DenseNet, EfficientNet and others.

Appendices

Appendix 1.Model Arhitectures

Pre-trainer

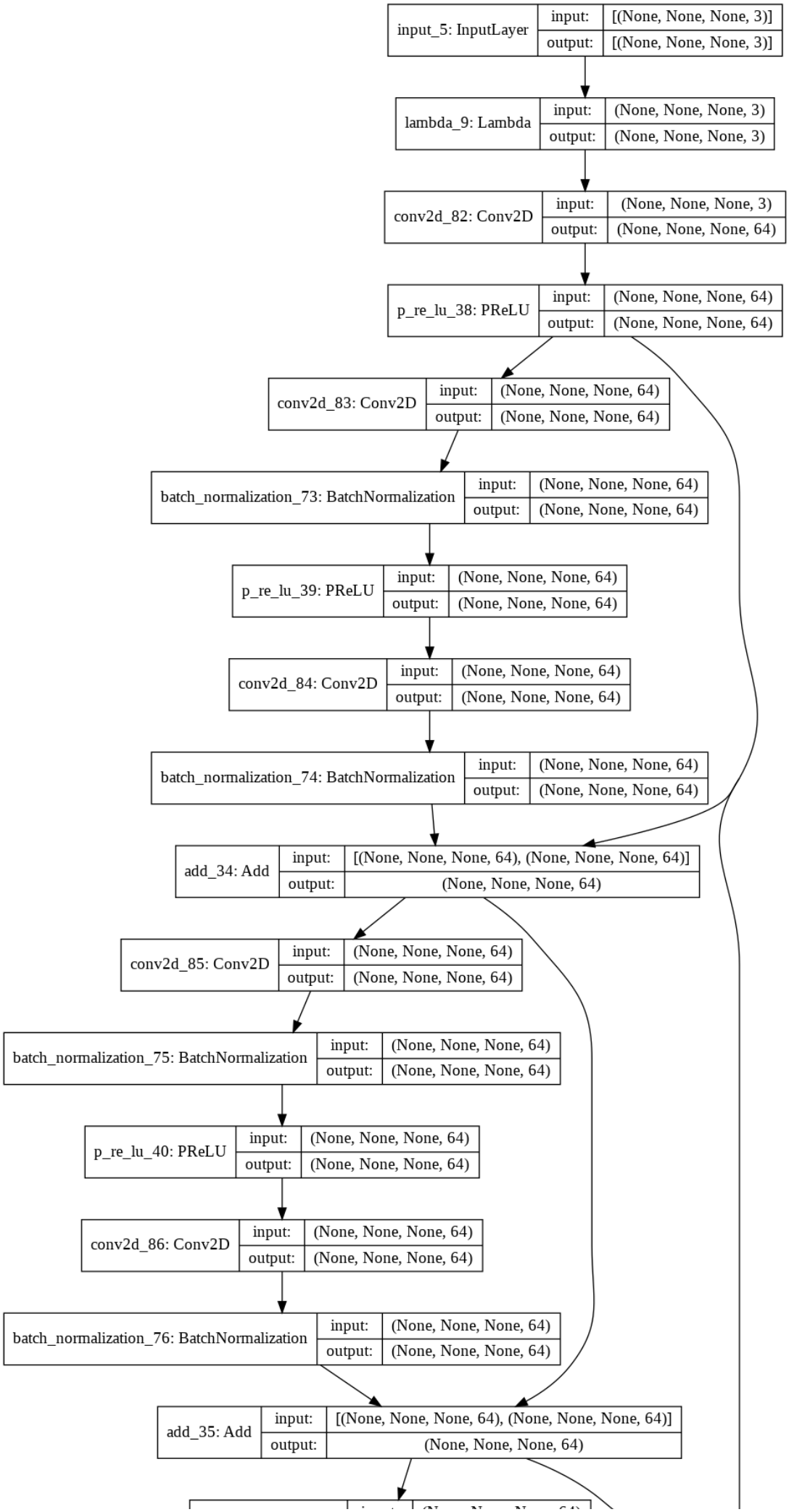
Model Summary

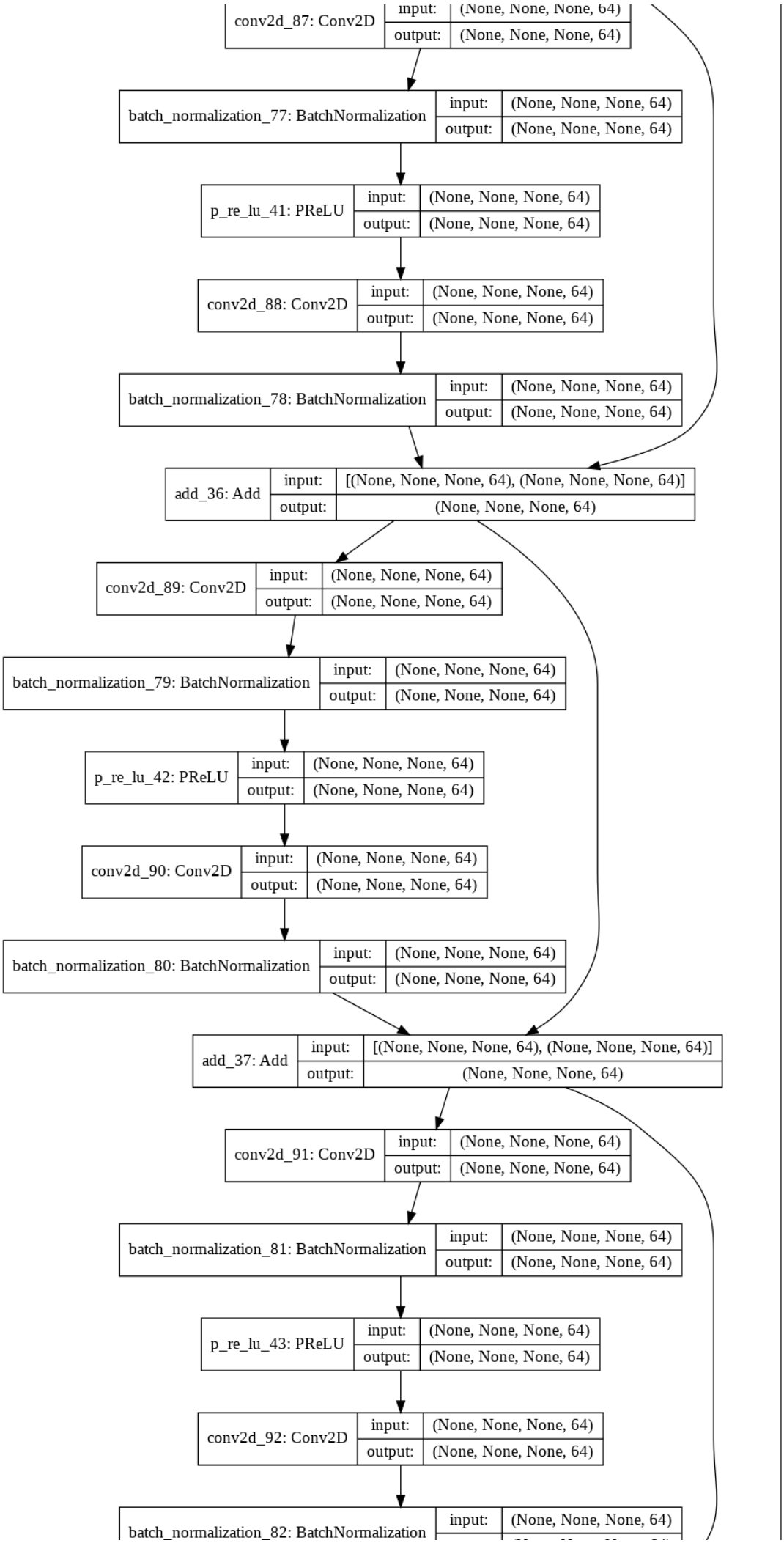
```
pre_trainer.model.summary()
```

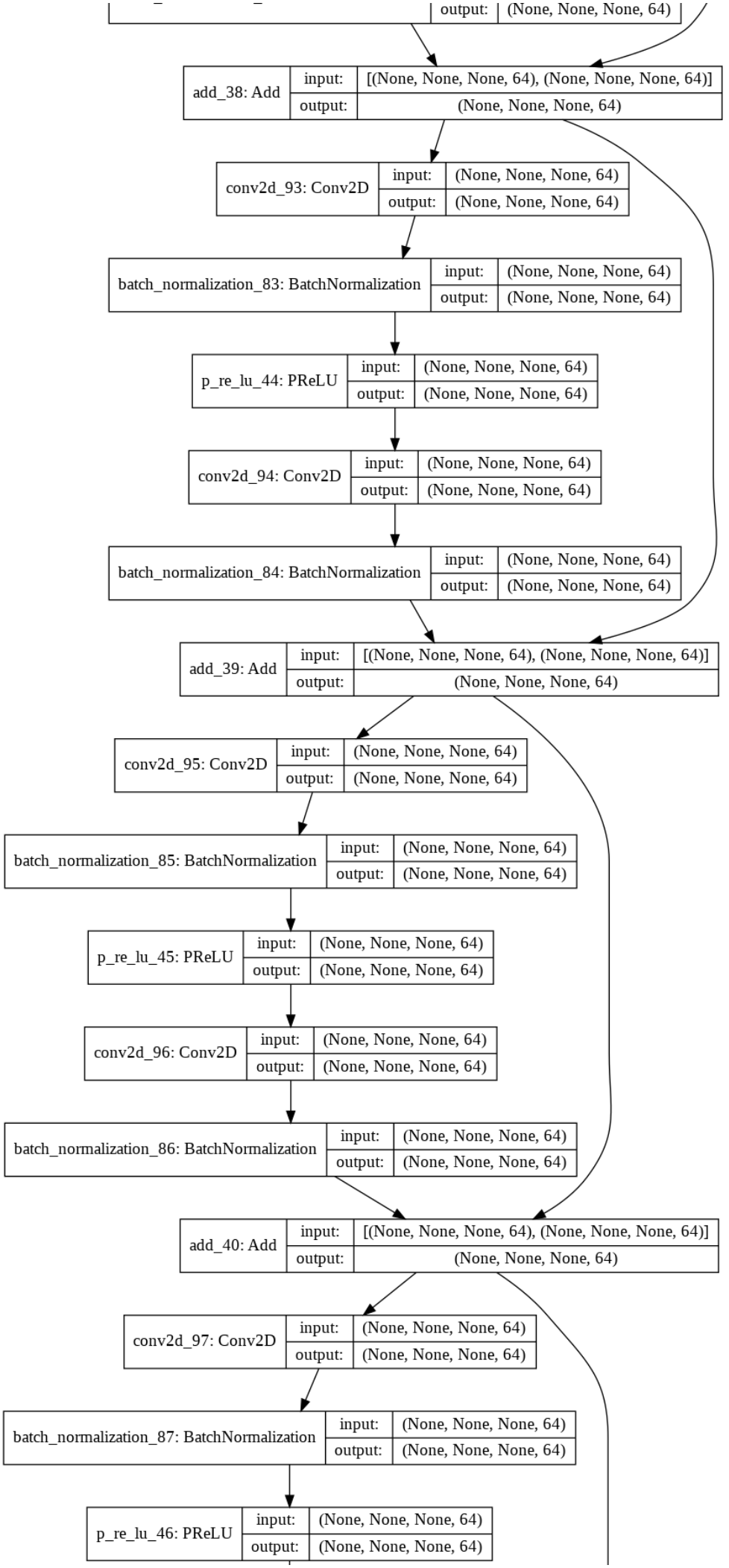
```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected
=================================================================
input_5 (InputLayer)         [(None, None, None,  0
_________________________________________________________________
lambda_9 (Lambda)            (None, None, None, 3 0           input_5[0]
_________________________________________________________________
conv2d_82 (Conv2D)           (None, None, None, 6 15616       lambda_9[0
_________________________________________________________________
p_re_lu_38 (PReLU)           (None, None, None, 6 64          conv2d_82[
_________________________________________________________________
conv2d_83 (Conv2D)           (None, None, None, 6 36928       p_re_lu_38
_________________________________________________________________
batch_normalization_73 (BatchNo (None, None, None, 6 256      conv2d_83[
_________________________________________________________________
p_re_lu_39 (PReLU)           (None, None, None, 6 64          batch_norm
_________________________________________________________________
conv2d_84 (Conv2D)           (None, None, None, 6 36928       p_re_lu_39
_________________________________________________________________
batch_normalization_74 (BatchNo (None, None, None, 6 256      conv2d_84[
_________________________________________________________________
add_34 (Add)                 (None, None, None, 6 0           p_re_lu_38
                                                              batch_norm
_________________________________________________________________
conv2d_85 (Conv2D)           (None, None, None, 6 36928       add_34[0][
_________________________________________________________________
batch_normalization_75 (BatchNo (None, None, None, 6 256      conv2d_85[
_________________________________________________________________
p_re_lu_40 (PReLU)           (None, None, None, 6 64          batch_norm
_________________________________________________________________
conv2d_86 (Conv2D)           (None, None, None, 6 36928       p_re_lu_40
_________________________________________________________________
batch_normalization_76 (BatchNo (None, None, None, 6 256      conv2d_86[
_________________________________________________________________
add_35 (Add)                 (None, None, None, 6 0           add_34[0][
                                                              batch_norm
_________________________________________________________________
conv2d_87 (Conv2D)           (None, None, None, 6 36928       add_35[0][
_________________________________________________________________
batch_normalization_77 (BatchNo (None, None, None, 6 256      conv2d_87[
_________________________________________________________________
p_re_lu_41 (PReLU)           (None, None, None, 6 64          batch_norm
_________________________________________________________________
conv2d_88 (Conv2D)           (None, None, None, 6 36928       p_re_lu_41
_________________________________________________________________
batch_normalization_78 (BatchNo (None, None, None, 6 256      conv2d_88[
_________________________________________________________________
add_36 (Add)                 (None, None, None, 6 0           add_35[0][
                                                              batch_norm
_________________________________________________________________
conv2d_89 (Conv2D)           (None, None, None, 6 36928       add_36[0][
_________________________________________________________________
batch_normalization_79 (BatchNo (None, None, None, 6 256      conv2d_89[
_________________________________________________________________
p_re_lu_42 (PReLU)           (None, None, None, 6 64          batch_norm
_________________________________________________________________
conv2d_90 (Conv2D)           (None, None, None, 6 36928       p_re_lu_42
```

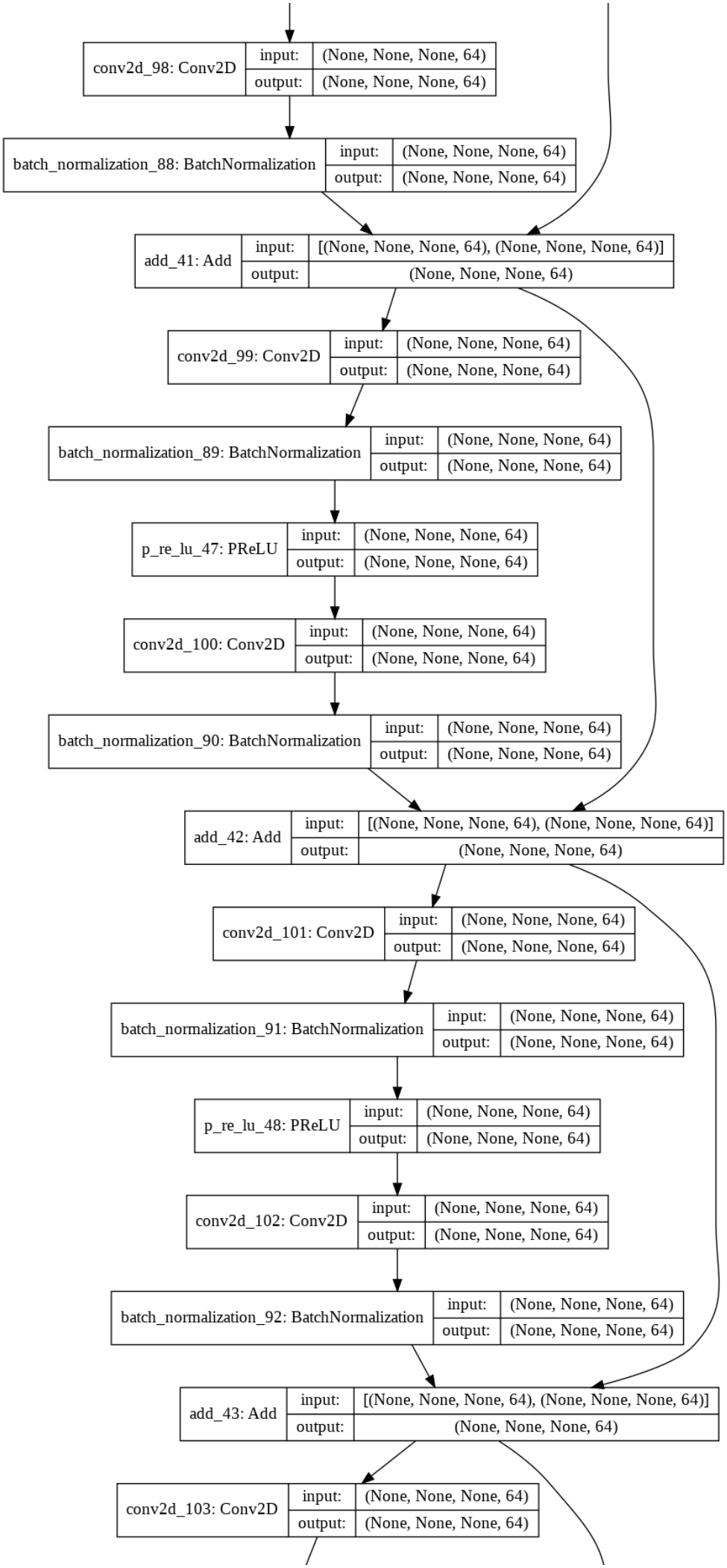
Model Architecture

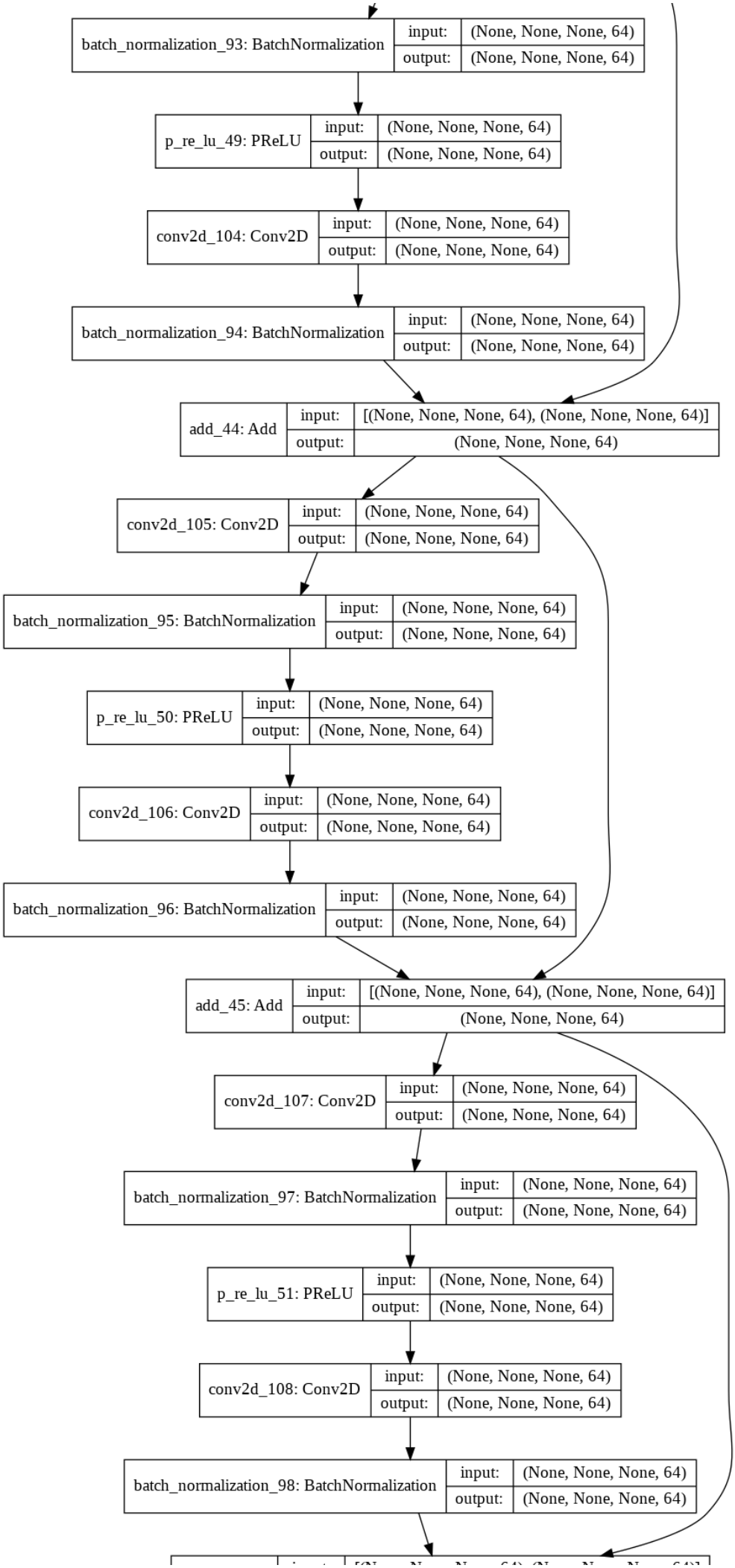
```
tf.keras.utils.plot_model(pre_trainer.model, "pre_trainer_model.png", show_shapes=
```

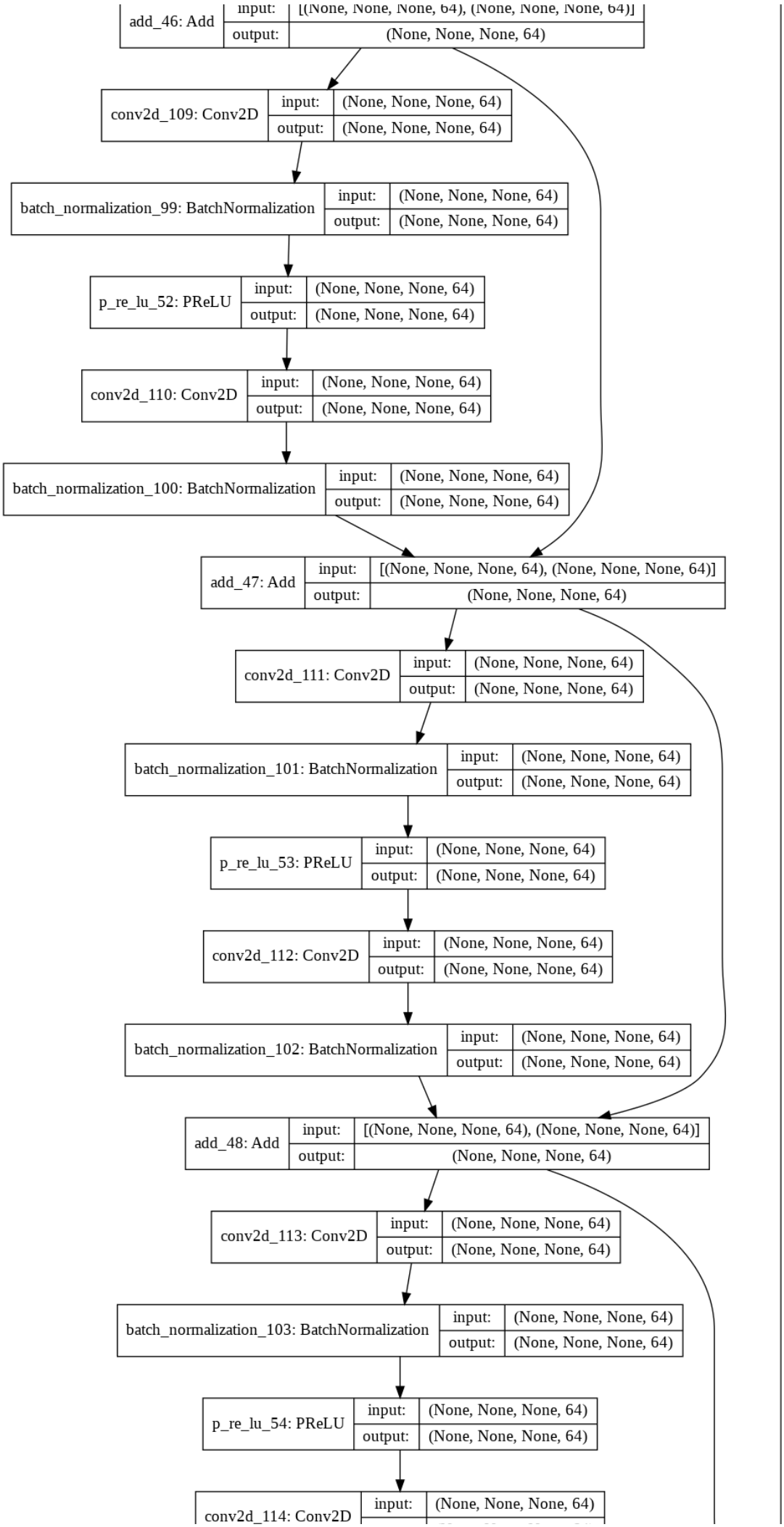

input_5: InputLayer
input: [(None, None, None, 3)]
output: [(None, None, None, 3)]
lambda_9: Lambda
input: (None, None, None, 3)
output: (None, None, None, 3)
conv2d_82: Conv2D
input: (None, None, None, 3)
output: (None, None, None, 64)
p_re_lu_38: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_83: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_73: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_39: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_84: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_74: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_34: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_85: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_75: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_40: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_86: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_76: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_35: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)


conv2d_87: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_77: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_41: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_88: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_78: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_36: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_89: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_79: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_42: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_90: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_80: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_37: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_91: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_81: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_43: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_92: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_82: BatchNormalization
input: (None, None, None, 64)


output: (None, None, None, 64)
add_38: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_93: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_83: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_44: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_94: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_84: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_39: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_95: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_85: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_45: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_96: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_86: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_40: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_97: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_87: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_46: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)


conv2d_98: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_88: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_41: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_99: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_89: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_47: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_100: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_90: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_42: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_101: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_91: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_48: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_102: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_92: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_43: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_103: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)


batch_normalization_93: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_49: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_104: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_94: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_44: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_105: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_95: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_50: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_106: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_96: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_45: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_107: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_97: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_51: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_108: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_98: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)


add_46: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_109: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_99: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_52: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_110: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_100: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_47: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_111: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_101: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_53: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_112: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_102: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
add_48: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
conv2d_113: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 64)
batch_normalization_103: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
p_re_lu_54: PReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv2d_114: Conv2D
input: (None, None, None, 64)

Нейронні мережі

Лекція_13

********** **********

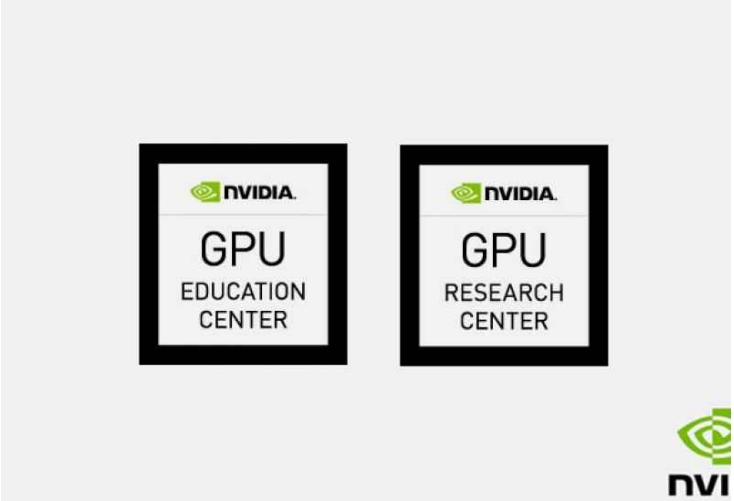
Слайди лекцій+ інтерактивні ноутбуки Jupyter для Google Colaboratory CPU/GPU/TPU cloud: https://cloud.comsys.kpi.ua/s/SMkBSsxRTazoTD6

********** **********

Лекція 13 - Рекурентні нейронні мережі - Вступ

Курс містить матеріали, запропоновані NVIDIA Deep Learning Institute (DLI) в рамках спільних:

Дослідницький центр NVIDIA
і
Освітній центр NVIDIA.



https://kpi.ua/nvidia-info

Інтерактивні демонстрації

ДЕМО 1
Рекурентні нейронні мережі - Вступ
https://drive.google.com/file/d/17FhwPfJGn-nT7QBDgBTyss7BwUtaAvH7/view?usp=sharing

Lecture 13 - Recurrent Neural Networks

(C) partially based on the works by [d2l Open Source book] authors, [TF Datasets] contributors, Ved, Karpathny, Sarang, Schmidhuber, Bengio, ...

Beside convolutional neural networks, the other very important and canonical task is **recurrent neural networks (RNN)**.

This DEMO is an introduction to RNN implemented by Tensorflow 2.0.

This DEMO will introduce techniques used for RNN with introduction to:

- sequence models,
- text preprocessing tasks,
- language models,
- recurrent neural networks,
- modern RNNs.

Part 0.Preparatory Actions

```
! nvidia-smi
```

```
Wed Apr 28 19:32:29 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.19.01    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   52C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
! pip install d2l
```

```
Collecting d2l
  Downloading https://files.pythonhosted.org/packages/69/62/a53c44bf0d076df
     |████████████████████████████████| 81kB 5.8MB/s
```

```
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.7/
Requirement already satisfied: notebook in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: tornado>=4.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: traitlets>=4.1.0 in /usr/local/lib/python3.7
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.7/d
Requirement already satisfied: ipython>=4.0.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.7/dis
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.0 in /usr/local/l
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: nbformat in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/
Requirement already satisfied: jupyterlab-widgets>=1.0.0; python_version >=
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/loc
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/py
Requirement already satisfied: ptyprocess; os_name != "nt" in /usr/local/li
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dis
```

Part 1.Basic Notions

So far we encountered two types of data: tabular data and image data. For the latter we designed specialized layers to take advantage of the regularity in them. In other words, if we were to permute the pixels in an image, it would be much more difficult to reason about its content of something that would look much like the background of a test pattern in the times of analog TV.

Most importantly, so far we tacitly assumed that our data are all drawn from some distribution, and all the examples are independently and identically distributed (i.i.d.). Unfortunately, this is not true for most data. For instance, the words in this paragraph are written in sequence, and it would be quite difficult to decipher its meaning if they were permuted randomly. Likewise, image frames in a video, the audio signal in a conversation, and the browsing behavior on a website, all follow sequential order. It is thus reasonable to assume that specialized models for such data will do better at describing them.

Another issue arises from the fact that we might not only receive a sequence as an input but rather might be expected to continue the sequence. For instance, the task could be to continue the series 2,4,6,8,10,…

This is quite common in time series analysis, to predict the stock market, the fever curve of a patient, or the acceleration needed for a race car. Again we want to have models that can handle such data.

In short, while CNNs can efficiently process spatial information, recurrent neural networks (RNNs) are designed to better handle sequential information. RNNs introduce state variables to store past information, together with the current inputs, to determine the current outputs.

Many of the examples for using recurrent networks are based on text data. Hence, we will emphasize language models in this chapter. After a more formal review of sequence data we introduce practical techniques for preprocessing text data. Next, we discuss basic concepts of a language model and use this discussion as the inspiration for the design of RNNs. In the end, we describe the gradient calculation method for RNNs to explore problems that may be encountered when training such networks.

8.1.Sequence Models

NOTE: here the section numbering follows the numbering in d2l Open Source book.

Imagine that you are watching movies on Netflix. As a good Netflix user, you decide to rate each of the movies religiously. After all, a good movie is a good movie, and you want to watch more of them, right? As it turns out, things are not quite so simple. People's opinions on movies can change quite significantly over time. In fact, psychologists even have names for some of the effects:

- **anchoring** - is based on someone else's opinion. For instance, after the Oscar awards, ratings for the corresponding movie go up, even though it is still the same movie. This effect persists for a few months until the award is forgotten. It has been shown that the effect lifts rating by over half a point`.

Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. Proceedings of the tenth ACM international conference on web search and data mining (pp. 495–503).

- **hedonic adaptation** - humans quickly adapt to accept an improved or a worsened situation as the new normal. For instance, after watching many good movies, the expectations that the next movie is equally good or better are high. Hence, even an average movie might be considered as bad after many great ones are watched.
- **seasonality** - very few viewers like to watch a Santa Claus movie in August.
- **misbehavior** - in some cases, movies become unpopular due to the misbehaviors of directors or actors in the production.
- **paradox** - some movies become cult movies, because they were almost comically bad. *Plan 9 from Outer Space* and *Troll 2* achieved a high degree of notoriety for this reason.

In short, movie ratings are **NOT stationary**. Thus, using temporal dynamics led to more accurate movie recommendations.

Koren, Y. (2009). Collaborative filtering with temporal dynamics. Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 447–456).

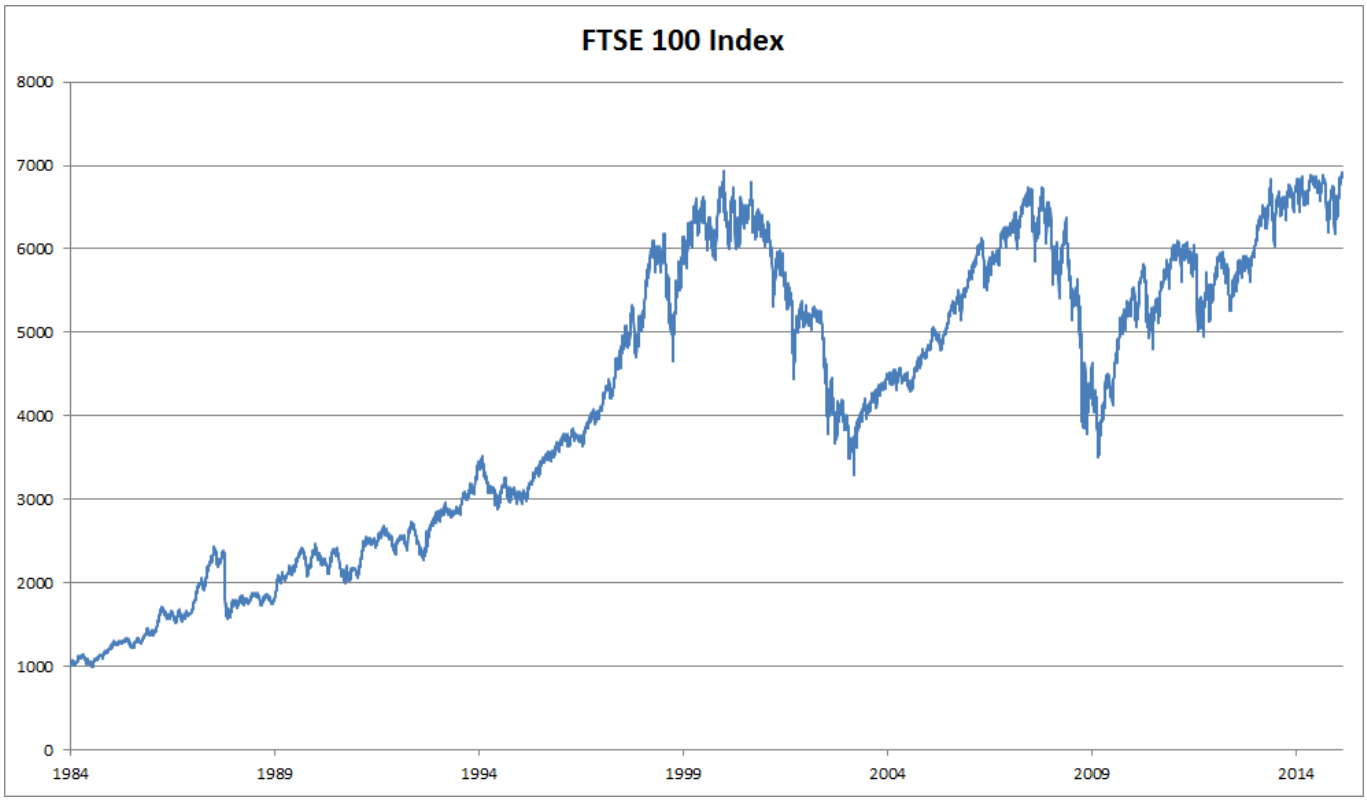
Other examples of sequence data:

- **Social activity** - Many users have highly particular behavior when it comes to the time when they open apps. For instance, social media apps are much more popular after school with students.
- **Stock market** - trading apps are more commonly used when the markets are open. **Foresight** is so much harder than hindsight. In statistics, the former (predicting beyond the known observations) is called **extrapolation** whereas the latter (estimating between the existing observations) is called **interpolation**.
- **Multimedia** - music, speech, text, and videos are all sequential in nature. If we were to permute them they would make little sense. The headline *dog bites man* is much less surprising than *man bites dog*, even though the words are identical.
- **Natural ... physical, chemical, biological, historical, social (see below), ... events** - earthquakes are strongly correlated, i.e., after a massive earthquake there are very likely several smaller aftershocks, much more so than without the strong quake. In fact, earthquakes are **spatiotemporally correlated**, i.e., the **aftershocks** typically occur within a short time span and in close proximity.
- **Social interactions** - humans interact with each other in a sequential nature, as can be seen in Twitter fights, dance patterns, and debates.

Statistical Tools

We need statistical tools and new deep neural network architectures to deal with sequence data.

As an example, let's consider the stock price (FTSE 100 index) illustrated in Figure below.



FTSE 100 index over about 30 years.

Let us denote the prices by x_t, i.e., at **time step** $t \in \mathbb{Z}^+$ we observe price x_t.

NOTE: For **sequences** in this text, t will typically be discrete and vary over integers or its subset.

Suppose that a trader who wants to do well in the stock market on day t predicts x_t via

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1).$$

Autoregressive Models

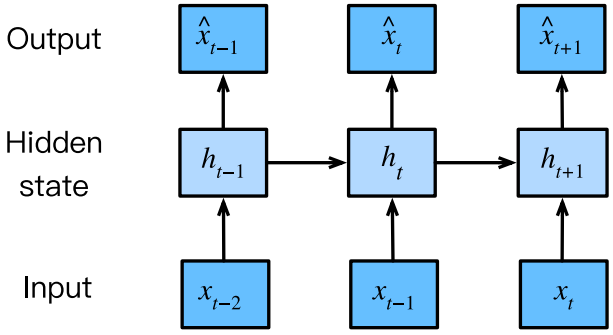
In order to achieve this, our trader could use a regression model such as the one that we used in the previous lectures.

BUT ... **problem** is: the number of inputs, $x_{t-1}, \ldots, x_1$ varies, depending on t.

That is to say, the number increases with the amount of data that we encounter, and we will need an approximation to make this computationally tractable. Much of what follows in this chapter will revolve around how to estimate $P(x_t \mid x_{t-1}, \ldots, x_1)$ efficiently.

We can use two strategies:

- **autoregressive model** - assume that the potentially rather long sequence $x_{t-1}, \ldots, x_1$ is not really necessary. In this case we might content ourselves with some timespan of length τ and only use $x_{t-1}, \ldots, x_{t-\tau}$ observations. The immediate benefit is that now **the number of arguments is always the same**, at least for $t > \tau$. This allows us to train a deep network as indicated above. Such models are called **autoregressive models**, because they perform regression on themselves.
- **latent autoregressive models** - to keep some summary h_t of the past observations, and at the same time update h_t in addition to the prediction $\hat{x}_t$ (see Figure below). This leads to models that estimate x_t with $\hat{x}_t = P(x_t \mid h_t)$ and moreover updates of the form $h_t = g(h_{t-1}, x_{t-1})$. Since h_t is never observed, these models are also called **latent autoregressive models**.



Dynamics

Stationary Dynamics

Both cases raise the obvious question of how to generate training data. One typically uses historical observations to predict the next observation given the ones up to right now. Obviously we do not expect time to stand still. However, a common assumption is that while the specific values of x_t might change, at least the dynamics of the sequence itself will not. This is reasonable, since novel dynamics are just that, novel and thus not predictable using data that we have so far. Statisticians call dynamics that do not change **stationary**. Regardless of what we do, we will thus get an estimate of the entire sequence via

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1).$$

Note that the above considerations still hold if we deal with discrete objects, such as words, rather than continuous numbers. The only difference is that in such a situation we need to use a classifier rather than a regression model to estimate $P(x_t \mid x_{t-1}, \ldots, x_1)$.

Markov Models

Recall the approximation that in an autoregressive model we use only $x_{t-1}, \ldots, x_{t-\tau}$ instead of $x_{t-1}, \ldots, x_1$ to estimate x_t. Whenever this approximation is accurate we say that the

sequence satisfies a **Markov condition**. In particular, if $\tau = 1$, we have a **first-order Markov model** and $P(x)$ is given by

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1).$$

Such models are particularly nice whenever x_t assumes only a discrete value, since in this case dynamic programming can be used to compute values along the chain exactly. For instance, we can compute $P(x_{t+1} \mid x_{t-1})$ efficiently:

$$\begin{aligned} P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\ &= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\ &= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1}) \end{aligned}$$

by using the fact that we only need to take into account a very short history of past observations: $P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$. Going into details of dynamic programming is beyond the scope of this section. Control and reinforcement learning algorithms use such tools extensively.

Causality

In principle, there is nothing wrong with unfolding $P(x_1, \ldots, x_T)$ in reverse order. After all, by conditioning we can always write it via

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T).$$

In fact, if we have a Markov model, we can obtain a reverse conditional probability distribution, too.

In many cases, however, there exists a **natural direction for the data**, namely going forward in time. It is clear that future events cannot influence the past. Hence, if we change x_t, we may be able to influence what happens for x_{t+1} going forward but not the converse. That is, if we change x_t, the distribution over past events will not change. Consequently, it ought to be easier to explain $P(x_{t+1} \mid x_t)$ rather than $P(x_t \mid x_{t+1})$.

Example: it has been shown that in some cases we can find $x_{t+1} = f(x_t) + \epsilon$ for some additive noise ϵ, whereas the converse is not true.

See details in:

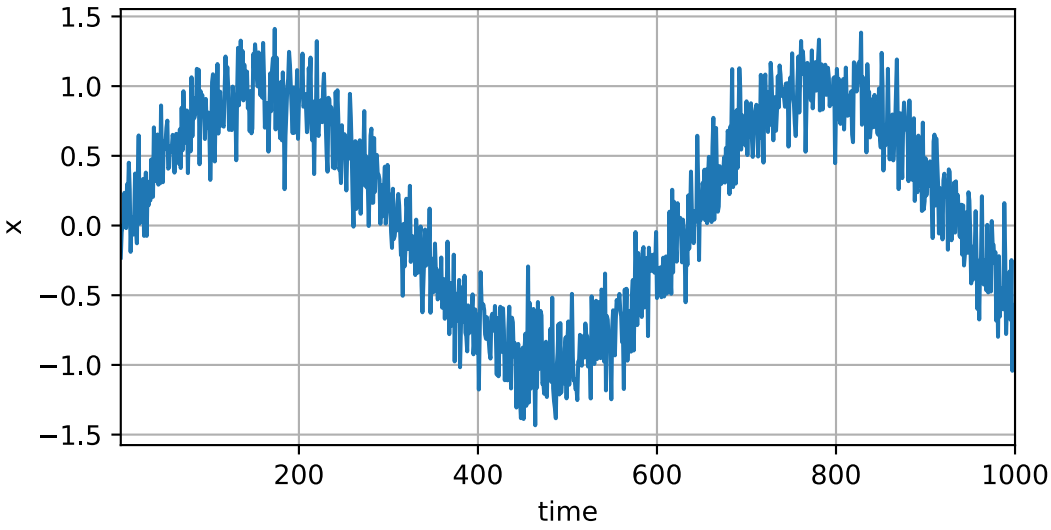
Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. Advances in neural information processing systems (pp. 689–696).

Training

After reviewing so many statistical tools, let us try this out in practice. We begin by generating some data. To keep things simple we generate our sequence data by using a sine function with some additive noise for time steps $1, 2, \ldots, 1000$.

```
%matplotlib inline
import tensorflow as tf
from d2l import tensorflow as d2l
import matplotlib.pyplot as plt
from tqdm import tqdm
```

```
T = 1000  # Generate a total of 1000 points
time = tf.range(1, T + 1, dtype=tf.float32)
x = tf.sin(0.01 * time) + tf.random.normal([T], 0, 0.2)
d2l.plot(time, [x], 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



Next, we need to turn such a sequence into features and labels that our model can train on. Based on the embedding dimension τ we map the data into pairs $y_t = x_t$ and $\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$. The astute reader might have noticed that this gives us τ fewer data examples, since we do not have sufficient history for the first τ of them. A simple fix, in particular if the sequence is long, is to discard those few terms. Alternatively we could pad the sequence with zeros. Here we only use the first 600 feature-label pairs for training.

```
tau = 4
features = tf.Variable(tf.zeros((T - tau, tau)))
for i in range(tau):
    features[:, i].assign(x[i:T - tau + i])
labels = tf.reshape(x[tau:], (-1, 1))
```

```
batch_size, n_train = 16, 600
# Only the first `n_train` examples are used for training
```

```
# Only the first `n_train` examples are used for training
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

Here we keep the architecture fairly simple: just an MLP with two fully-connected layers, ReLU activation, and squared loss.

```
# Vanilla MLP architecture
def get_net():
    net = tf.keras.Sequential([
        tf.keras.layers.Dense(10, activation='relu'),
        tf.keras.layers.Dense(1)])
    return net

# Least mean squares loss
# Note: L2 Loss = 1/2 * MSE Loss. TensorFlow has MSE Loss that is slightly
# different from MXNet's L2Loss by a factor of 2. Hence we halve the loss
# value to get L2Loss in TF
loss = tf.keras.losses.MeanSquaredError()
```

Now we are ready to train the model. The code below is essentially identical to the training loop in previous sections (and our firts lectures). Thus, we will not delve into much detail.

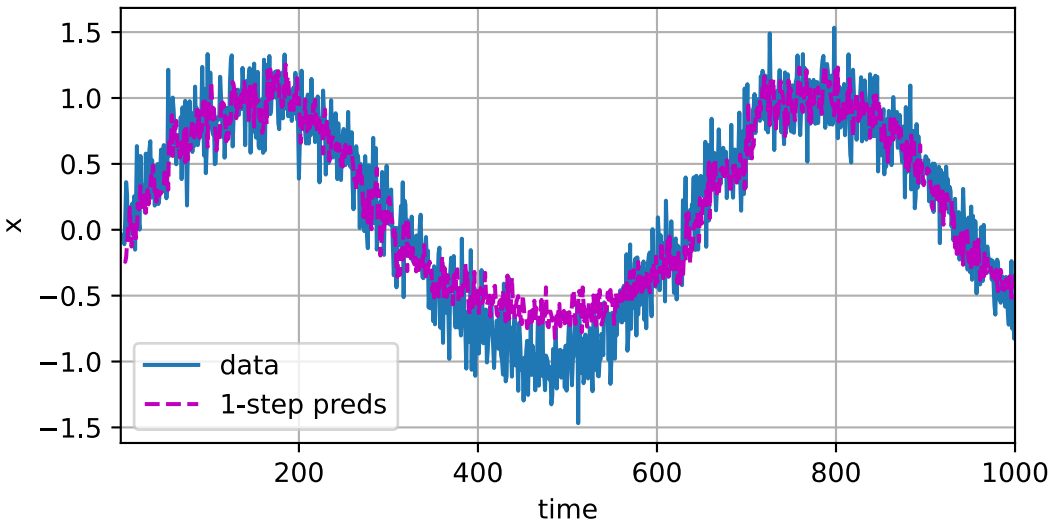
```
%%time

def train(net, train_iter, loss, epochs, lr):
    trainer = tf.keras.optimizers.Adam()
    for epoch in tqdm(range(epochs)):
        for X, y in train_iter:
            with tf.GradientTape() as g:
                out = net(X)
                l = loss(y, out) / 2
                params = net.trainable_variables
                grads = g.gradient(l, params)
            trainer.apply_gradients(zip(grads, params))
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')

net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
 20%|██          | 1/5 [00:00<00:01,  2.99it/s]epoch 1, loss: 0.373879
 40%|████        | 2/5 [00:00<00:00,  3.01it/s]epoch 2, loss: 0.258709
 60%|██████      | 3/5 [00:00<00:00,  3.04it/s]epoch 3, loss: 0.171063
 80%|████████    | 4/5 [00:01<00:00,  3.11it/s]epoch 4, loss: 0.111197
100%|██████████| 5/5 [00:01<00:00,  3.10it/s]epoch 5, loss: 0.078036
CPU times: user 1.61 s, sys: 73.1 ms, total: 1.69 s
Wall time: 1.63 s
```

Prediction

Since the training loss is small, we would expect our model to work well. Let us see what this means in practice. The first thing to check is how well the model is able to predict what happens just in the next time step, namely the *one-step-ahead prediction*.

```
onestep_preds = net(features)
d2l.plot([time, time[tau:]], [x.numpy(), onestep_preds.numpy()], 'time', 'x',
         legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



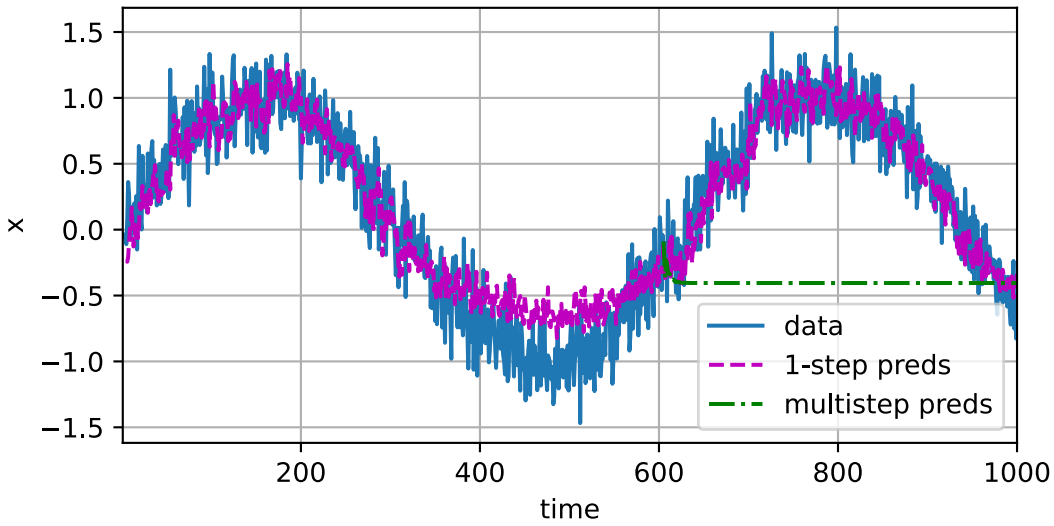
The one-step-ahead predictions look nice, just as we expected. Even beyond 604 (`n_train + tau`) observations the predictions still look trustworthy. However, there is just one little problem to this: if we observe sequence data only until time step 604, we cannot hope to receive the inputs for all the future one-step-ahead predictions. Instead, we need to work our way forward one step at a time:

$$\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}),\\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}),\\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}),\\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}),\\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}),\\
&\ldots
\end{aligned}$$

Generally, for an observed sequence up to x_t, its predicted output $\hat{x}_{t+k}$ at time step $t+k$ is called the k*-step-ahead prediction*. Since we have observed up to x_{604}, its k-step-ahead prediction is $\hat{x}_{604+k}$. In other words, we will have to use our own predictions to make multistep-ahead predictions. Let us see how well this goes.

```
multistep_preds = tf.Variable(tf.zeros(T))
multistep_preds[:n_train + tau].assign(x[:n_train + tau])
for i in range(n_train + tau, T):
    multistep_preds[i].assign(
        tf.reshape(net(tf.reshape(multistep_preds[i - tau:i], (1, -1))), ()))
```

```
d2l.plot([time, time[tau:], time[n_train + tau:]], [
    x.numpy(),
    onestep_preds.numpy(), multistep_preds[n_train + tau:].numpy()], 'time',
         'x', legend=['data', '1-step preds',
                      'multistep preds'], xlim=[1, 1000], figsize=(6, 3))
```



As the above example shows, this is a spectacular failure. The predictions decay to a constant pretty quickly after a few prediction steps.

Question: Why did the algorithm work so poorly?

Answer: This is ultimately due to the fact that the **errors build up**.

Let us say that after step 1 we have some error $\epsilon_1 = \bar{\epsilon}$. Now the *input* for step 2 is perturbed by ϵ_1, hence we suffer some error in the order of $\epsilon_2 = \bar{\epsilon} + c\epsilon_1$ for some constant c, and so on. The error can diverge rather rapidly from the true observations. This is a common phenomenon.

Example: Weather forecasts for the next 24 hours tend to be pretty accurate but beyond that the accuracy declines rapidly.

We will discuss methods for improving this throughout this Lecture and beyond.

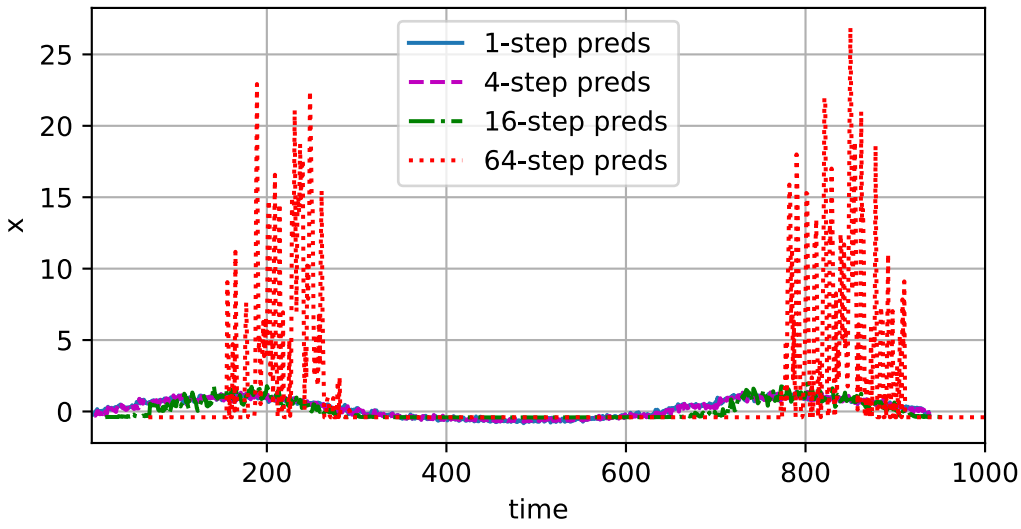
Let us take a closer look at the difficulties in k-step-ahead predictions by computing predictions on the entire sequence for $k = 1, 4, 16, 64$.

```
max_steps = 64
```

```
features = tf.Variable(tf.zeros((T - tau - max_steps + 1, tau + max_steps)))
# Column `i` (`i` < `tau`) are observations from `x` for time steps from
# `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau):
    features[:, i].assign(x[i:i + T - tau - max_steps + 1].numpy())

# Column `i` (`i` >= `tau`) are the (`i - tau + 1`)-step-ahead predictions for
# time steps from `i + 1` to `i + T - tau - max_steps + 1`
```

```
for i in range(tau, tau + max_steps):
    features[:, i].assign(tf.reshape(net((features[:, i - tau:i])), -1))
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1:T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].numpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds'
                 for i in steps], xlim=[5, 1000], figsize=(6, 3))
```



This clearly illustrates how the quality of the prediction changes as we try to predict further into the future. While the 4-step-ahead predictions still look good, anything beyond that is almost useless.

Example: the new model (net_50steps) after more (50) training steps

Naïve assumption: Let's try to improve the model by the longer training! :)

Note: about **naïve**.

Origin: mid 17th century: from French *naïve*, feminine of *naïf*, from Latin *nativus* (native, natural).

Тобто давайте зробимо **naïve** або **nativus**, тобто природне, припущення про покращення моделі додатковим тренуванням. :)

1-step-ahead prediction

```
tau = 4
features = tf.Variable(tf.zeros((T - tau, tau)))
for i in range(tau):
    features[:, i].assign(x[i:T - tau + i])
labels = tf.reshape(x[tau:], (-1, 1))
```

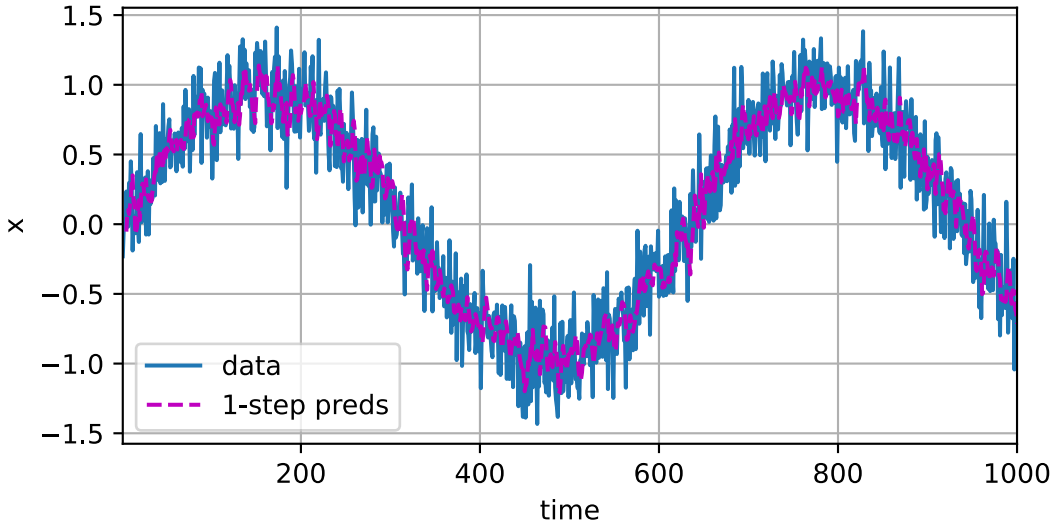
```
batch_size, n_train = 16, 600
# Only the first `n_train` examples are used for training
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

```
%%time

net_50steps = get_net()
train(net_50steps, train_iter, loss, 50, 0.01)
```

```
  2%|▏          | 1/50 [00:00<00:16,  3.00it/s]epoch 1, loss: 0.077044
  4%|▍          | 2/50 [00:00<00:15,  3.08it/s]epoch 2, loss: 0.068694
  6%|▌          | 3/50 [00:00<00:15,  3.09it/s]epoch 3, loss: 0.067377
  8%|▊          | 4/50 [00:01<00:14,  3.13it/s]epoch 4, loss: 0.064462
 10%|▉          | 5/50 [00:01<00:14,  3.12it/s]epoch 5, loss: 0.062622
 12%|█▏         | 6/50 [00:01<00:13,  3.15it/s]epoch 6, loss: 0.061527
 14%|█▍         | 7/50 [00:02<00:13,  3.19it/s]epoch 7, loss: 0.060215
 16%|█▌         | 8/50 [00:02<00:13,  3.20it/s]epoch 8, loss: 0.059803
 18%|█▊         | 9/50 [00:02<00:12,  3.19it/s]epoch 9, loss: 0.058771
 20%|██         | 10/50 [00:03<00:12,  3.22it/s]epoch 10, loss: 0.057696
 22%|██▏        | 11/50 [00:03<00:12,  3.21it/s]epoch 11, loss: 0.057085
 24%|██▍        | 12/50 [00:03<00:11,  3.20it/s]epoch 12, loss: 0.056940
 26%|██▌        | 13/50 [00:04<00:11,  3.21it/s]epoch 13, loss: 0.055517
 28%|██▊        | 14/50 [00:04<00:11,  3.20it/s]epoch 14, loss: 0.054887
 30%|███        | 15/50 [00:04<00:10,  3.20it/s]epoch 15, loss: 0.054876
 32%|███▏       | 16/50 [00:05<00:10,  3.16it/s]epoch 16, loss: 0.054587
 34%|███▍       | 17/50 [00:05<00:10,  3.14it/s]epoch 17, loss: 0.053849
 36%|███▌       | 18/50 [00:05<00:10,  3.17it/s]epoch 18, loss: 0.054080
 38%|███▊       | 19/50 [00:05<00:09,  3.20it/s]epoch 19, loss: 0.053352
 40%|████       | 20/50 [00:06<00:09,  3.19it/s]epoch 20, loss: 0.053163
 42%|████▏      | 21/50 [00:06<00:09,  3.17it/s]epoch 21, loss: 0.053486
 44%|████▍      | 22/50 [00:06<00:08,  3.21it/s]epoch 22, loss: 0.052596
 46%|████▌      | 23/50 [00:07<00:08,  3.21it/s]epoch 23, loss: 0.052923
 48%|████▊      | 24/50 [00:07<00:08,  3.21it/s]epoch 24, loss: 0.052483
 50%|█████      | 25/50 [00:07<00:07,  3.15it/s]epoch 25, loss: 0.052205
 52%|█████▏     | 26/50 [00:08<00:07,  3.15it/s]epoch 26, loss: 0.052222
 54%|█████▍     | 27/50 [00:08<00:07,  3.16it/s]epoch 27, loss: 0.052538
 56%|█████▌     | 28/50 [00:08<00:06,  3.20it/s]epoch 28, loss: 0.052424
 58%|█████▊     | 29/50 [00:09<00:06,  3.22it/s]epoch 29, loss: 0.052097
 60%|██████     | 30/50 [00:09<00:06,  3.24it/s]epoch 30, loss: 0.051564
 62%|██████▏    | 31/50 [00:09<00:05,  3.29it/s]epoch 31, loss: 0.051871
 64%|██████▍    | 32/50 [00:10<00:05,  3.28it/s]epoch 32, loss: 0.052094
 66%|██████▌    | 33/50 [00:10<00:05,  3.21it/s]epoch 33, loss: 0.051440
 68%|██████▊    | 34/50 [00:10<00:04,  3.24it/s]epoch 34, loss: 0.051946
 70%|███████    | 35/50 [00:10<00:04,  3.21it/s]epoch 35, loss: 0.051635
 72%|███████▏   | 36/50 [00:11<00:04,  3.19it/s]epoch 36, loss: 0.052237
 74%|███████▍   | 37/50 [00:11<00:04,  3.19it/s]epoch 37, loss: 0.051363
 76%|███████▌   | 38/50 [00:11<00:03,  3.22it/s]epoch 38, loss: 0.051129
 78%|███████▊   | 39/50 [00:12<00:03,  3.21it/s]epoch 39, loss: 0.051821
 80%|████████   | 40/50 [00:12<00:03,  3.18it/s]epoch 40, loss: 0.051612
 82%|████████▏  | 41/50 [00:12<00:02,  3.21it/s]epoch 41, loss: 0.051429
 84%|████████▍  | 42/50 [00:13<00:02,  3.21it/s]epoch 42, loss: 0.051209
 86%|████████▌  | 43/50 [00:13<00:02,  3.19it/s]epoch 43, loss: 0.050920
 88%|████████▊  | 44/50 [00:13<00:01,  3.19it/s]epoch 44, loss: 0.051267
 90%|█████████  | 45/50 [00:14<00:01,  3.20it/s]epoch 45, loss: 0.051019
 92%|█████████▏ | 46/50 [00:14<00:01,  3.20it/s]epoch 46, loss: 0.051708
 94%|█████████▍ | 47/50 [00:14<00:00,  3.21it/s]epoch 47, loss: 0.051127
 96%|█████████▌ | 48/50 [00:15<00:00,  3.18it/s]epoch 48, loss: 0.051774
```

```
 98%|█████████▊| 49/50 [00:15<00:00,  3.14it/s]epoch 49, loss: 0.051307
100%|██████████| 50/50 [00:15<00:00,  3.19it/s]epoch 50, loss: 0.051147
CPU times: user 15.8 s, sys: 697 ms, total: 16.5 s
Wall time: 15.7 s
```

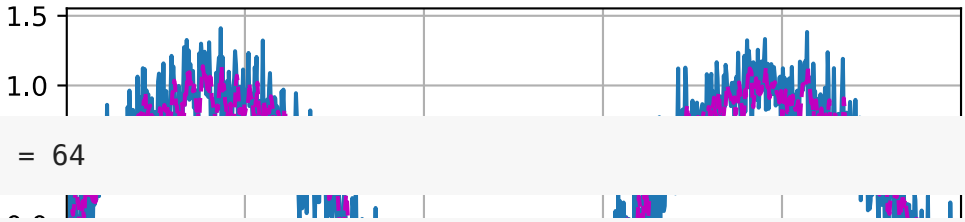
```
onestep_preds = net_50steps(features)
d2l.plot([time, time[tau:]], [x.numpy(), onestep_preds.numpy()], 'time', 'x',
         legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



k-step-ahead prediction

```
multistep_preds = tf.Variable(tf.zeros(T))
multistep_preds[:n_train + tau].assign(x[:n_train + tau])
for i in range(n_train + tau, T):
    multistep_preds[i].assign(
        tf.reshape(net_50steps(tf.reshape(multistep_preds[i - tau:i], (1, -1))), (
```

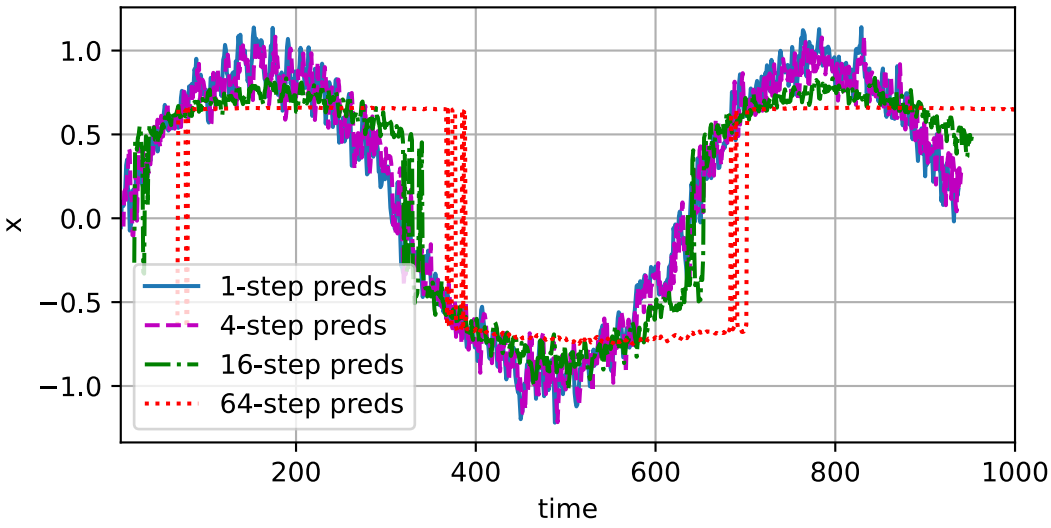
```
d2l.plot([time, time[tau:], time[n_train + tau:]], [
    x.numpy(),
    onestep_preds.numpy(), multistep_preds[n_train + tau:].numpy()], 'time',
         'x', legend=['data', '1-step preds',
                      'multistep preds'], xlim=[1, 1000], figsize=(6, 3))
```




```
max_steps = 64
```

```
features = tf.Variable(tf.zeros((T - tau - max_steps + 1, tau + max_steps)))
# Column `i` (`i` < `tau`) are observations from `x` for time steps from
# `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau):
    features[:, i].assign(x[i:i + T - tau - max_steps + 1].numpy())

# Column `i` (`i` >= `tau`) are the (`i - tau + 1`)-step-ahead predictions for
# time steps from `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau, tau + max_steps):
    features[:, i].assign(tf.reshape(net_50steps((features[:, i - tau:i])), -1))
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1:T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].numpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds'
                 for i in steps], xlim=[5, 1000], figsize=(6, 3))
```



Again, this clearly illustrates how the quality of the prediction changes as we try to predict further into the future. While the 4-step-ahead predictions still look good, anything beyond that is almost useless ... even for the better **(aftre the longer training)** model.

Summary

- **Prediction on sequences** are used for:
    - interpolation,
    - extrapolation,

- **Causality** - if you have a sequence, always respect the **temporal order** of the data when training, i.e., **never train on future data**,
- Sequence models require **specialized statistical tools** for estimation:
    - **autoregressive** models,
    - **latent-variable autoregressive** models,
- **k-step-ahead prediction** - it is the predicted output at time step $t + k$ for an observed sequence up to time step t, BUT ... **if we predict further** in time by increasing k, the **errors will accumulate** and the quality of the prediction **will degrade**, often dramatically.

8.2.Text Preprocessing

We have reviewed and evaluated statistical tools and prediction challenges for sequence data.

Sequence data can take many forms.

Text is one of the most popular examples of sequence data.

For example, an article can be simply viewed as a sequence of words, or even a sequence of characters.

The common preprocessing steps for text data are:

- **Load** text as **strings** into memory.
- **Split** strings into **tokens** (e.g., words and characters).
- Build a table of **vocabulary** -> to **map** the split **tokens** to numerical **indices**.
- **Convert text** into **sequences** of numerical **indices** so they can be manipulated by models easily.

```
import collections
import re
from d2l import tensorflow as d2l
```

Reading the Dataset

To get started we load text from H. G. Wells' *The Time Machine*.

The Time Machine

H. G. Wells

The Time Machine (at Gutenberg Project)

This is a fairly small corpus of just over 30000 words, but for the purpose of what we want to illustrate this is just fine. More realistic document collections contain many billions of words. The following function reads the dataset into a list of text lines, where each line is a string. For simplicity, here we ignore punctuation and capitalization.

```
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.amazor
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

If you want you can check the full text:

https://www.gutenberg.org/files/35/35-h/35-h.htm

Tokenization

The following `tokenize` function takes a list (`lines`) as the input, where each list is a text sequence (e.g., a text line). Each text sequence is split into a list of tokens. A *token* is the basic unit in text. In the end, a list of token lists are returned, where each token is a string.

```
def tokenize(lines, token='word'):
    """Split text lines into word or character tokens."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)
```

```
# Spaces
tokens = tokenize(lines, 'space')
for i in range(11):
    print(tokens[i])
```

```
ERROR: unknown token type: space
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-106-82d4ffebc76b> in <module>()
      2 tokens = tokenize(lines, 'space')
      3 for i in range(11):
----> 4     print(tokens[i])

TypeError: 'NoneType' object is not subscriptable
```

SEARCH STACK OVERFLOW

```
# Chars
tokens = tokenize(lines, 'char')
for i in range(11):
    print(tokens[i])
```

```
['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm', 'a', 'c', 'h', 'i', 'n', '
[]
[]
[]
[]
['i']
[]
[]
['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 't', 'r', 'a', 'v', 'e', 'l', '
['w', 'a', 's', ' ', 'e', 'x', 'p', 'o', 'u', 'n', 'd', 'i', 'n', 'g', ' ', '
['t', 'w', 'i', 'n', 'k', 'l', 'e', 'd', ' ', 'a', 'n', 'd', ' ', 'h', 'i', '
```

```
# Words
tokens = tokenize(lines, 'word')
for i in range(11):
    print(tokens[i])
```

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', '
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey',
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and'
```

Vocabulary

The string type of the token is inconvenient to be used by models, which take numerical inputs.

Now let us build a **dictionary**, often called **vocabulary** as well, to map string tokens into numerical indices starting from 0.

- count the **unique tokens** in all the documents from the training set, namely a **corpus**,
- assign a numerical index to each unique token according to its **frequency**.

NOTEs:

- rarely appeared tokens are often removed to reduce the complexity.
- any token that does not exist in the corpus or has been removed is mapped into a special unknown token “<unk>”.
- optionally add a list of reserved tokens:
    - “<pad>” for padding,
    - “<bos>” to present the beginning for a sequence, and
    - “<eos>” for the end of a sequence.

```
class Vocab:
    """Vocabulary for text."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # The index for the unknown token is 0
        self.unk, uniq_tokens = 0, ['<unk>'] + reserved_tokens
        uniq_tokens += [
            token for token, freq in self.token_freqs
            if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

def count_corpus(tokens):
    """Count token frequencies."""
```

```
    # Here `tokens` is a 1D list or 2D list
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Flatten a list of token lists into a list of tokens
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

We construct a vocabulary using the time machine dataset as the corpus. Then we print the first few frequent tokens with their indices.

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to',
```

Now we can convert each text line into a list of numerical indices.

```
len(tokens)
```

```
3221
```

```
len(vocab)
```

```
4580
```

```
#for i in [0, 10]:
for i in range(10):
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 2183, 2184, 400]
words: []
indices: []
words: []
indices: []
words: []
indices: []
words: []
indices: []
words: ['i']
indices: [2]
words: []
indices: []
words: []
indices: []
words: ['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'conveni
indices: [1, 19, 71, 16, 37, 11, 115, 42, 680, 6, 586, 4, 108]
words: ['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', '
indices: [7, 1420, 5, 2185, 587, 6, 126, 25, 330, 127, 439, 3]
```

Putting All Things Together

Using the above functions, we package everything into:

- `load_corpus_time_machine` function, which returns **corpus**, a list of token indices, and
- `vocab`, the **vocabulary** of the time machine corpus.

The modifications we did here are:

- **tokenize** text **into characters**, not words, to simplify the training in later sections;
- **corpus** is a single **list**, and **NOT** a **list of token lists**, since each text line in the `Time Machine` dataset is not necessarily a sentence or a paragraph.

```
def load_corpus_time_machine(max_tokens=-1):
    """Return token indices and the vocabulary of the time machine dataset."""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Since each text line in the time machine dataset is not necessarily a
    # sentence or a paragraph, flatten all the text lines into a single list
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(170580, 28)
```

Summary

- Text is an important form of sequence data.
- To preprocess text, we usually split text into tokens, build a vocabulary to map token strings into numerical indices, and convert text data into token indices for models to manipulate.

NOTEs:

- English words are often separated from each other by whitespace, but whitespace is not always sufficient.
    - “New York” and “rock ’n’ roll” are sometimes treated as large words despite the fact that they contain spaces.
    - And revrese, sometimes we’ll need to separate “I’m” into the two words I and am.
- Some other languages, like Chinese, don’t have spaces between words, so word tokenization becomes more difficult.

8.3.Language Models and the Dataset

Language Model -> Aim

Above we saw how to map text data into tokens, where these tokens can be viewed as a sequence of discrete observations, such as words or characters.

Assume that the tokens in a text sequence of length T are in turn $x_1, x_2, \ldots, x_T$. Then, in the text sequence, $x_t (1 \leq t \leq T)$ can be considered as the observation or label at time step t.

Given such a text sequence, the **aim** of a **language model** is to estimate the joint probability of the sequence

$$P(x_1, x_2, \ldots, x_T).$$

Language models are incredibly useful.

Example: an ideal language model would be able to generate natural text just on its own, simply by drawing one token at a time:

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1).$$

Quite unlike the monkey using a typewriter, all text emerging from such a model would pass as natural language, e.g., English text.

Furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments.

Clearly we are still very far from designing such a system, since it would need to *understand* the text rather than just generate grammatically sensible content.

Nonetheless, language models are of great service even in their limited form.

Example: the phrases "to recognize speech" and "to wreck a nice beach" sound very similar.

This can cause ambiguity in speech recognition, which is easily resolved through a language model that rejects the second translation as outlandish.

Likewise, in a document summarization algorithm it is worthwhile knowing that "dog bites man" is much more frequent than "man bites dog", or that "I want to eat grandma" is a rather disturbing statement, whereas "I want to eat, grandma" is much more benign.

Learning Language Model

Question: how to model a document, or even a sequence of tokens?

Suppose that we tokenize text data at the word level.

Let us start by applying basic probability rules from sections above:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}).$$

Example: the probability of a text sequence containing four words would be given as:

$$P(\text{deep}, \text{learning}, \text{is}, \text{fun}) = P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid$$

To **compute the language model**, we need to calculate:

- the probability of words,
- the conditional probability of a word given the previous few words.

These probabilities are **language model parameters**.

The **probability** of words can be calculated from the **relative word frequency** of a given word in the training dataset.

Examples:

- the estimate $\hat{P}(\text{deep})$ can be calculated as the probability of any **sentence starting** with the word "deep".
- a slightly less accurate approach would be to count **all occurrences** of the word "deep" and divide it by the total number of words in the corpus.

This works fairly well, particularly for frequent words. Moving on, we could attempt to estimate

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})},$$

where

- $n(x)$ is the number of occurrences of **singletons**,
- $n(x, x')$ is the number of occurrences of **consecutive word pairs**.

BUT ... estimating the probability of a word pair is somewhat more difficult, since the occurrences of "deep learning" are a lot less frequent.

In particular, for some unusual word combinations it may be tricky to find enough occurrences to get accurate estimates.

It becomes **worse for three-word combinations** and beyond. There will be many plausible three-word combinations that we likely will not see in our dataset.

Problem:

- Unless we provide some solution to assign such word combinations nonzero count, we will not be able to use them in a language model.
- If the dataset is small or if the words are very rare, we might not find even a single one of them.

Solution: A common strategy is to perform some form of **Laplace smoothing**: add a small constant to all counts.

Let's denote by n the total number of words in the training set and m the number of unique words.

This solution helps with singletons, e.g., via

$$\hat{P}(x) = \frac{n(x) + \epsilon_1/m}{n + \epsilon_1},$$
$$\hat{P}(x' \mid x) = \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2},$$
$$\hat{P}(x'' \mid x, x') = \frac{n(x, x', x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.$$

where ϵ_1, ϵ_2, and ϵ_3 are hyperparameters.

Example:

- when $\epsilon_1 = 0$, no smoothing is applied to x;
- when ϵ_1 approaches positive infinity, $\hat{P}(x)$ approaches the uniform probability $1/m$.

The above is a rather primitive variant of what other techniques can accomplish. See details in:

Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. Communications of the ACM, 54(2), 91–98.

BUT ... models like this become cumbersome very quickly for the following reasons:

- we need to **store all** counts,
- this approach entirely **ignores the meaning** of the words, for instance, "cat" and "feline" should occur in related contexts,
- it is quite **difficult to adjust** such models **to additional contexts**, whereas, **BUT** ... **DL-based** language models **are well suited** to take this into account,
- **long word sequences** are almost certain to be **novel**, hence a model that simply counts the frequency of previously seen word sequences is bound to **perform poorly** there.

Dataset

Let's assume that the **training dataset** is a large text corpus, such as all Wikipedia entries, Project Gutenberg, and all text posted on the Web.

Markov Models and n-grams

Before we discuss solutions involving DL, we need some more terminology and concepts.

Let's apply Markov Models to language modeling:

- a distribution over sequences satisfies the Markov property of first order if $P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$,
- the higher orders correspond to longer dependencies.

This leads to a number of approximations that we could apply to model a sequence:

$$
\begin{aligned}
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4),\\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3),\\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3).
\end{aligned}
$$

The probability formulae that involve one, two, and three variables are typically referred to as **unigram**, **bigram**, and **trigram** models, respectively.

In the following, we will learn how to design better models.

Natural Language Statistics

Let's construct a vocabulary based on the `Time Machine` dataset and print the top 10 most frequent words.

```
import random
import tensorflow as tf
from d2l import tensorflow as d2l
```

Unigram Frequency

```
tokens = d2l.tokenize(d2l.read_time_machine())
# Since each text line is not necessarily a sentence or a paragraph, we
# concatenate all text lines
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
vocab.token_freqs[:10]
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.amazor
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

As we can see, **the most popular words** are actually quite **boring** to look at.

They are often referred to as **stop words** and thus filtered out.

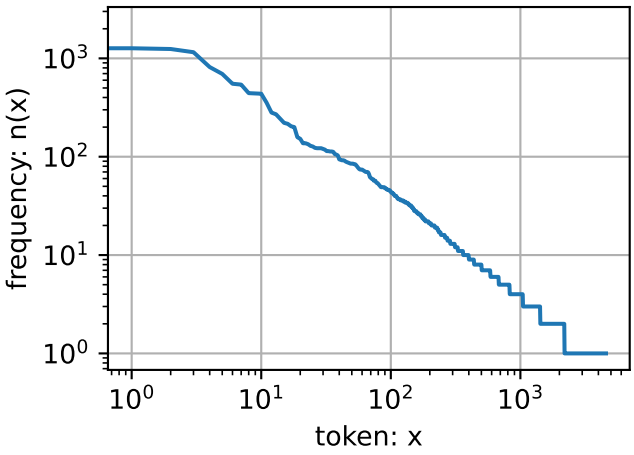
BUT ... they still carry meaning and we will still use them!

NOTE: the word frequency decays rather rapidly!

The 10^{th} most frequent word is less than $1/5$ as common as the most popular one.

To get a better idea, we plot the figure of the word frequency.

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)', xscale='log',
         yscale='log')
```



We are on to something quite fundamental here: the word frequency decays rapidly in a well-defined way.

After dealing with the first few words as exceptions, all the remaining **word frequencies** roughly follow a straight **line** on a **log-log** plot.

This means that words satisfy **Zipf's law**, which states that the frequency n_i of the i^{th} most frequent word is:

$$n_i \propto \frac{1}{i^\alpha},$$

which is equivalent to

$$\log n_i = -\alpha \log i + c,$$

where α is the exponent that characterizes the distribution and c is a constant.

This should already give us pause if we want to model words by count statistics and smoothing. After all, we will significantly overestimate the frequency of the tail, also known as the infrequent words.

But what about the other word combinations, such as bigrams, trigrams, and beyond?

Bigram Frequency

Let us see whether the **bigram frequency** behaves in the same manner as the unigram frequency.

```
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[(('of', 'the'), 309),
 (('in', 'the'), 169),
 (('i', 'had'), 130),
 (('i', 'was'), 112),
 (('and', 'the'), 109),
 (('the', 'time'), 102),
 (('it', 'was'), 99),
 (('to', 'the'), 85),
 (('as', 'i'), 78),
 (('of', 'a'), 73)]
```

NOTE: out of the 10 most frequent word pairs, 9 are composed of **both stop words** and only 1 is relevant to the actual book — "the time".

Trigram Frequency

Furthermore, let us see whether the **trigram frequency** behaves in the same manner.

```
trigram_tokens = [
    triple for triple in zip(corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

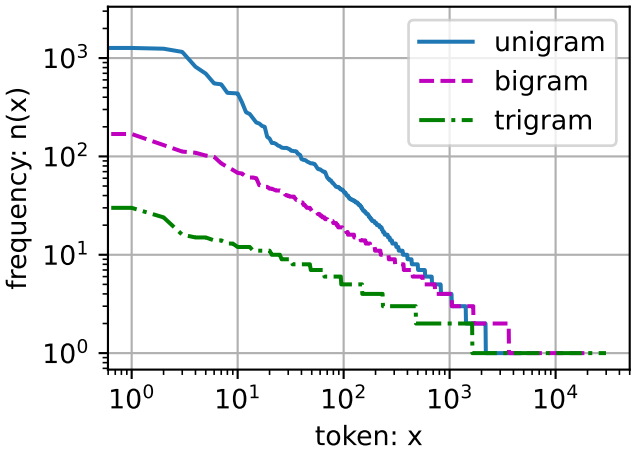
```
[(('the', 'time', 'traveller'), 59),
 (('the', 'time', 'machine'), 30),
 (('the', 'medical', 'man'), 24),
 (('it', 'seemed', 'to'), 16),
 (('it', 'was', 'a'), 15),
 (('here', 'and', 'there'), 15),
 (('seemed', 'to', 'me'), 14),
 (('i', 'did', 'not'), 14),
 (('i', 'saw', 'the'), 13),
 (('i', 'began', 'to'), 13)]
```

Last, let us visualize the token frequency among these three models:

- unigrams,
- bigrams,
- trigrams.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
```

```
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



This figure is quite exciting for a number of reasons:

- the lengthy sequences of words **also** appear **to be following Zipf's law**, albeit with a smaller exponent α, depending on the sequence length,
- the number of distinct n-grams is not that large and it gives us hope and hint(!) that there is quite a lot of **structure in language**,
- many n-grams occur very rarely, which makes **Laplace smoothing rather unsuitable** for language modeling and that is why ... **instead**, we will use DL-based models.

Reading Long Sequence Data

Since sequence data are by their very nature sequential, we need to address the issue of processing it.

Above we did so in a rather ad-hoc manner.

When sequences get too long to be processed by models all at once, we may wish to split such sequences for reading.

Now let us describe general strategies.

Before introducing the model, let us assume that we will use a neural network to train a language model, where the network processes a minibatch of sequences with predefined length, say n time steps, at a time.

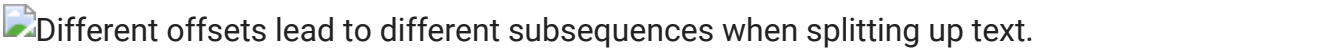
Now the question is how to read minibatches of features and labels at random.

To begin with, since a text sequence can be arbitrarily long, such as the entire `Time Machine` book, we can partition such a long sequence into subsequences with the same number of time steps.

When training our neural network, a minibatch of such subsequences will be fed into the model.

Suppose that the network processes a subsequence of n time steps at a time.

All the different ways to obtain subsequences from an original text sequence, where $n = 5$ and a token at each time step corresponds to a character, are shown in Figure below.



Different offsets lead to different subsequences when splitting up text.

Note that we have quite some freedom since we could pick an arbitrary offset that indicates the initial position. Hence, which one should we pick from this Figure?

In fact, all of them are equally good.

However, if we pick just one offset, there is limited coverage of all the possible subsequences for training our network.

Therefore, we can start with a random offset to partition a sequence to get both **coverage** and **randomness**.

In the following, we describe how to accomplish this for two strategies:

- **random sampling**,
- **sequential partitioning**.

Random Sampling

In random sampling, each example is a subsequence arbitrarily captured on the original long sequence.

The subsequences from two adjacent random minibatches during iteration are not necessarily adjacent on the original sequence.

For language modeling, the target is to predict the next token based on what tokens we have seen so far, hence the labels are the original sequence, shifted by one token.

The following code randomly generates a minibatch from the data each time.

Here, the argument `batch_size` specifies the number of subsequence examples in each minibatch and `num_steps` is the predefined number of time steps in each subsequence.

```
def seq_data_iter_random(corpus, batch_size, num_steps):
    """Generate a minibatch of subsequences using random sampling."""
    # Start with a random offset (inclusive of `num_steps - 1`) to partition a
    # sequence
    corpus = corpus[random.randint(0, num_steps - 1):]
    # Subtract 1 since we need to account for labels
    num_subseqs = (len(corpus) - 1) // num_steps
    # The starting indices for subsequences of length `num_steps`
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))
    # In random sampling, the subsequences from two adjacent random
    # minibatches during iteration are not necessarily adjacent on the
```

```
    # original sequence
    random.shuffle(initial_indices)

    def data(pos):
        # Return a sequence of length `num_steps` starting from `pos`
        return corpus[pos:pos + num_steps]

    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Here, `initial_indices` contains randomized starting indices for
        # subsequences
        initial_indices_per_batch = initial_indices[i:i + batch_size]
        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield tf.constant(X), tf.constant(Y)
```

Let's manually generate a sequence from 0 to 34.

Let's assume that the batch size and numbers of time steps are 2 and 5, respectively.

This means that we can generate $\lfloor (35 - 1)/5 \rfloor = 6$ feature-label subsequence pairs.

With a minibatch size of 2, we only get 3 minibatches.

```
my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tf.Tensor(
[[20 21 22 23 24]
 [ 5  6  7  8  9]], shape=(2, 5), dtype=int32)
Y: tf.Tensor(
[[21 22 23 24 25]
 [ 6  7  8  9 10]], shape=(2, 5), dtype=int32)
X:  tf.Tensor(
[[ 0  1  2  3  4]
 [25 26 27 28 29]], shape=(2, 5), dtype=int32)
Y: tf.Tensor(
[[ 1  2  3  4  5]
 [26 27 28 29 30]], shape=(2, 5), dtype=int32)
X:  tf.Tensor(
[[15 16 17 18 19]
 [10 11 12 13 14]], shape=(2, 5), dtype=int32)
Y: tf.Tensor(
[[16 17 18 19 20]
 [11 12 13 14 15]], shape=(2, 5), dtype=int32)
```

Sequential Partitioning

In this strategy the subsequences from two adjacent minibatches during iteration are **adjacent** on the original sequence.

This strategy preserves the order of split subsequences when iterating over minibatches, hence

```
def seq_data_iter_sequential(corpus, batch_size, num_steps):
    """Generate a minibatch of subsequences using sequential partitioning."""
    # Start with a random offset to partition a sequence
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = tf.constant(corpus[offset:offset + num_tokens])
    Ys = tf.constant(corpus[offset + 1:offset + 1 + num_tokens])
    Xs = tf.reshape(Xs, (batch_size, -1))
    Ys = tf.reshape(Ys, (batch_size, -1))
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_batches * num_steps, num_steps):
        X = Xs[:, i:i + num_steps]
        Y = Ys[:, i:i + num_steps]
        yield X, Y
```

Using the same settings, let's print features `X` and labels `Y` for each minibatch of subsequences read by sequential partitioning.

Note that the subsequences from two adjacent minibatches during iteration are indeed adjacent on the original sequence.

```
for X, Y in seq_data_iter_sequential(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tf.Tensor(
[[ 1  2  3  4  5]
 [17 18 19 20 21]], shape=(2, 5), dtype=int32)
Y: tf.Tensor(
[[ 2  3  4  5  6]
 [18 19 20 21 22]], shape=(2, 5), dtype=int32)
X:  tf.Tensor(
[[ 6  7  8  9 10]
 [22 23 24 25 26]], shape=(2, 5), dtype=int32)
Y: tf.Tensor(
[[ 7  8  9 10 11]
 [23 24 25 26 27]], shape=(2, 5), dtype=int32)
X:  tf.Tensor(
[[11 12 13 14 15]
 [27 28 29 30 31]], shape=(2, 5), dtype=int32)
Y: tf.Tensor(
[[12 13 14 15 16]
 [28 29 30 31 32]], shape=(2, 5), dtype=int32)
```

Now we wrap the above two sampling functions to a class so that we can use it as a data iterator later.

```
class SeqDataLoader:
    """An iterator to load sequence data."""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
```

```
            self.data_iter_fn = d2l.seq_data_iter_random
        else:
            self.data_iter_fn = d2l.seq_data_iter_sequential
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

Last, we define a function `load_data_time_machine` that returns both the data iterator and the vocabulary, so we can use it similarly as other other functions with the `load_data` prefix.

```
def load_data_time_machine(batch_size, num_steps,
                           use_random_iter=False, max_tokens=10000):
    """Return the iterator and the vocabulary of the time machine dataset."""
    data_iter = SeqDataLoader(batch_size, num_steps, use_random_iter,
                              max_tokens)
    return data_iter, data_iter.vocab
```

Summary

- Language models are key to **natural language processing (NLP)**.
- n-grams provide a convenient model for dealing with long sequences by truncating the dependence.
- Long sequences suffer from the problem that they occur very rarely or never.
- Zipf's law governs the word distribution for not only unigrams but also the other n-grams.
- There is a lot of structure but not enough frequency to deal with infrequent word combinations efficiently via Laplace smoothing.
- The main choices for reading long sequences are:
    - random sampling,
    - sequential partitioning.
- The sequential partitioning can ensure that the subsequences from two adjacent minibatches during iteration are adjacent on the original sequence.

8.4.Recurrent Neural Networks

Before we introduced n-gram models, where the conditional probability of word x_t at time step t only depends on the $n-1$ previous words.

If we want to incorporate the possible effect of words earlier than time step $t-(n-1)$ on x_t, we need to increase n.

However, the number of model parameters would also increase exponentially with it, as we need to store $|\mathcal{V}|^n$ numbers for a vocabulary set $\mathcal{V}$.

Hence, rather than modeling $P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$ it is preferable to use a latent variable model:

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}),$$

where h_{t-1} is a **hidden state** (also known as a hidden variable) that stores the sequence information up to time step $t-1$.

In general, the hidden state at any time step t could be computed based on both the current input x_t and the previous hidden state h_{t-1}:

$$h_t = f(x_t, h_{t-1}).$$

For a sufficiently powerful function f in this equation, the latent variable model is not an approximation. After all, h_t may simply store all the data it has observed so far.

BUT ... it could potentially make both computation and storage expensive.

Recall that we have discussed hidden layers with hidden units above.

NOTE: Let's consider two very different concepts:

- **hidden layers** are hidden from view on the path from input to output in DNN,
- **hidden states** are technically speaking **inputs** to whatever we do at a given step, and they can **only be computed** by **looking** at data **at previous time steps**.

Recurrent neural networks (RNNs) are neural networks with hidden states.

Neural Networks without Hidden States

Let's take a look at an MLP with a single hidden layer.

Let the hidden layer's activation function be ϕ.

Given a **minibatch of examples** $\mathbf{X} \in \mathbb{R}^{n \times d}$ with **batch size** n and d **inputs**, the hidden layer's **output** $\mathbf{H} \in \mathbb{R}^{n \times h}$ is calculated as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h).$$

In this equation, we have

- the **weight** parameter $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$,
- the **bias** parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, the number of **hidden** units h, for the hidden layer.

Thus, broadcasting (see :numref: `subsec_broadcasting`) is applied during the summation.

Next, the **hidden** variable $\mathbf{H}$ is used as the input of the output layer.

The **output layer** is given by

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q,$$

where

$\mathbf{O} \in \mathbb{R}^{n \times q}$ is the **output** variable,

$\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ is the **weight** parameter,

$\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ is the **bias** parameter of the output layer.

If it is a **classification problem**, we can use $\mathrm{softmax}(\mathbf{O})$ to compute the **probability distribution** of the output **categories**.

This is entirely analogous to the regression problem we solved previously.

Suffice it to say that we can:

- **pick feature-label pairs at random**,
- **learn the parameters** of our network via automatic differentiation and stochastic gradient descent.

Recurrent Neural Networks with Hidden States

BUT ... it is entirely different when we have hidden states.

Let's look at the structure in some more detail.

Assume that we have a minibatch of inputs $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ at time step t.

In other words, for a minibatch of n sequence examples, each row of $\mathbf{X}_t$ corresponds to one example at time step t from the sequence.

Next, denote by $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ the hidden variable of time step t.

Unlike the MLP, here we save the hidden variable $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ to describe how to use the hidden variable of the previous time step in the current time step.

Namely, the calculation of the hidden variable of the current time step is determined by:

- the **input** of the **current time step**,
- together with the **hidden variable** of the **previous time step**:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

Compared with the previous equation, this equation has one more term $\mathbf{H}_{t-1} \mathbf{W}_{hh}$.

From the relationship between hidden variables $\mathbf{H}_t$ and $\mathbf{H}_{t-1}$ of adjacent time steps, we know that these variables captured and retained the sequence's historical information up to their current time step, just like the state or memory of the neural network's current time step.

Therefore, such a hidden variable is called a **hidden state**.

Since the hidden state uses the same definition of the previous time step in the current time step, then:

- *computation* is **recurrent** one,
- *neural networks* with hidden states based on recurrent computation are named **recurrent neural networks**,
- *layers* that perform the computation in RNNs are called **recurrent layers**.

There are many different ways for constructing RNNs.

RNNs with a hidden state defined by the last equation are very common.

For time step t, the output of the output layer is similar to the computation in the MLP:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

Parameters of the RNN include:

- the **weights** $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ and the **bias** $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ of the **hidden layer**,
- together with the **weights** $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ and the **bias** $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ of the **output layer**.

NOTE: Even at different time steps, RNNs always use these model parameters. Therefore, the **parameterization cost** of an RNN does **NOT** grow as the number of **time steps increases**.

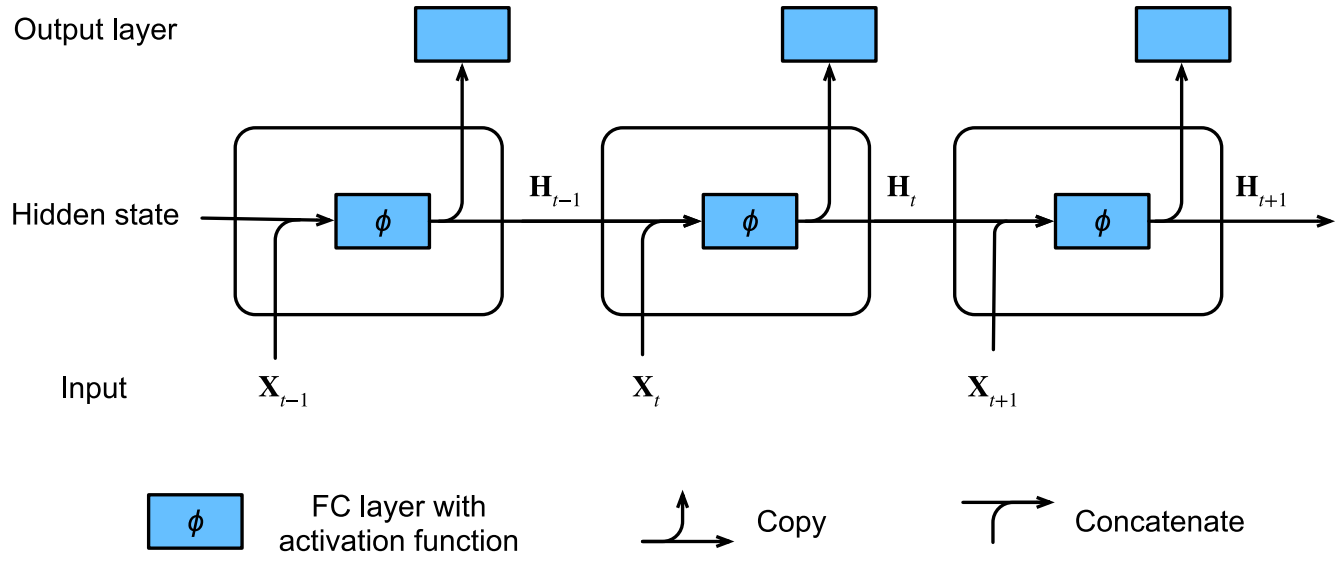
Figure (below) illustrates the computational logic of an RNN at three adjacent time steps.

At any time step t, the computation of the hidden state can be treated as:

- **concatenating** the **input** $\mathbf{X}_t$ at the **current** time step t and the **hidden state** $\mathbf{H}_{t-1}$ at the **previous** time step $t-1$;
- **feeding** the concatenation **result** into a fully-connected **layer** with the activation function ϕ .

NOTEs:

- The **output** of such a fully-connected layer is the **hidden state** $\mathbf{H}_t$ of the current time step t.
- In this case, the **model parameters** are the **concatenation** of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, and a **bias** of $\mathbf{b}_h$.
- The **hidden state** of the **current** time step t, $\mathbf{H}_t$, will **participate in computing** the hidden state $\mathbf{H}_{t+1}$ of the **next** time step $t+1$.
- Moreover, $\mathbf{H}_t$ will **also be fed** into the fully-connected **output layer** to compute the output $\mathbf{O}_t$ of the **current** time step t.



An RNN with a hidden state.

Mini Example

We just mentioned that the calculation of $\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$ for the hidden state is equivalent to matrix multiplication of concatenation of $\mathbf{X}_t$ and $\mathbf{H}_{t-1}$ and concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$.

Though this can be proven in mathematics, in the following we just use a simple code snippet to show this.

To begin with, we define matrices `X`, `W_xh`, `H`, and `W_hh`, whose shapes are (3, 1), (1, 4), (3, 4), and (4, 4), respectively.

Multiplying `X` by `W_xh`, and `H` by `W_hh`, respectively, and then adding these two multiplications, we obtain a matrix of shape (3, 4).

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

```
X, W_xh = tf.random.normal((3, 1), 0, 1), tf.random.normal((1, 4), 0, 1)
H, W_hh = tf.random.normal((3, 4), 0, 1), tf.random.normal((4, 4), 0, 1)
tf.matmul(X, W_xh) + tf.matmul(H, W_hh)
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[-1.3689078 , -1.321936  , -0.07917786,  2.2443914 ],
       [-2.6267166 , -1.1234753 , -1.2144722 ,  1.7868385 ],
       [-3.1305854 , -2.7384236 ,  1.1009977 ,  4.4005747 ]],
      dtype=float32)>
```

Now we concatenate the matrices `X` and `H` along columns (axis 1), and the matrices `W_xh` and `W_hh` along rows (axis 0). These two concatenations result in matrices of shape (3, 5) and of

shape (5, 4), respectively. Multiplying these two concatenated matrices, we obtain the same

```
tf.matmul(tf.concat((X, H), 1), tf.concat((W_xh, W_hh), 0))
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[-1.3689078 , -1.321936  , -0.07917783,  2.2443912 ],
       [-2.6267164 , -1.1234753 , -1.214472  ,  1.7868384 ],
       [-3.1305852 , -2.7384238 ,  1.1009978 ,  4.400575  ]],
      dtype=float32)>
```

RNN-based Character-Level Language Models

Recall that for language modeling, our aim is to predict the next token based on the current and past tokens, thus we shift the original sequence by one token as the labels.

Bengio et al. first proposed to use a neural network for language modeling.

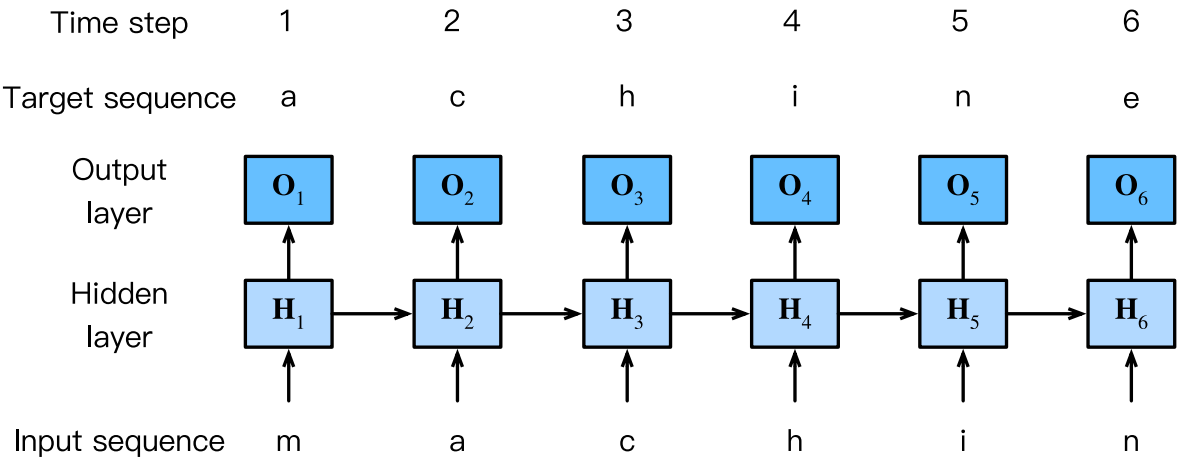
Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. Journal of machine learning research, 3(Feb), 1137–1155.

In the following Figure we illustrate how RNNs can be used to build a language model.

Let the minibatch size be one, and the sequence of the text be "machine".

To simplify training in subsequent sections, we tokenize text into characters rather than words and consider a **character-level language model**.

Figure below demonstrates how to predict the next character based on the current and previous characters via an RNN for character-level language modeling.



A character-level language model based on the RNN. The input and label sequences are "machin" and "achine", respectively.

During the training process, we run a softmax operation on the output from the output layer for each time step, and then use the cross-entropy loss to compute the error between the model output and the label.

Due to the recurrent computation of the hidden state in the hidden layer, the output of time step 3 in Figure, $\mathbf{O}_3$, is determined by the text sequence "m", "a", and "c".

Since the next character of the sequence in the training data is "h", the loss of time step 3 will depend on the probability distribution of the next character generated based on the feature sequence "m", "a", "c" and the label "h" of this time step.

In practice, each token is represented by a d-dimensional vector, and we use a batch size $n > 1$.

Therefore, the input $\mathbf{X}_t$ at time step t will be a $n \times d$ matrix.

Perplexity

Let's discuss how to measure the language model quality, which will be used to evaluate our RNN-based models in the subsequent sections.

One way is to check how surprising the text is.

A good language model is able to predict with high-accuracy tokens that what we will see next.

Consider the following continuations of the phrase "It is raining", as proposed by different language models:

1. "It is raining outside"
2. "It is raining banana tree"
3. "It is raining piouw;kcj pwepoiut"

- Example 1 is clearly the best in terms of quality. The words are sensible and logically coherent. While it might not quite accurately reflect which word follows semantically ("in San Francisco" and "in winter" would have been perfectly reasonable extensions), the model is able to capture which kind of word follows.
- Example 2 is considerably worse by producing a nonsensical extension. Nonetheless, at least the model has learned how to spell words and some degree of correlation between words.
- Example 3 indicates a poorly trained model that does not fit data properly.

We might measure the quality of the model by computing the likelihood of the sequence.

Unfortunately this is a number that is hard to understand and difficult to compare.

After all, shorter sequences are much more likely to occur than the longer ones, hence evaluating the model on Tolstoy's magnum opus *War and Peace* will inevitably produce a much smaller likelihood than, say, on Saint-Exupery's novella *The Little Prince*.

What is missing ... is the equivalent of an **average**.

BUT ... **information theory** can help here with some notions like:

- entropy,
- surprisal,

- cross-entropy.

More on information theory is discussed in the online appendix on information theory.

If we want to compress text, we can ask about predicting the next token given the current set of tokens.

A better language model should allow us to predict the next token more accurately.

Thus, it should allow us to spend fewer bits in compressing the sequence.

So we can measure it by the **cross-entropy loss** averaged over all the n tokens of a sequence:

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t \mid x_{t-1}, \ldots, x_1),$$

where

- P is given by a language model,
- x_t is the actual token observed at time step t from the sequence.

This makes the performance on documents of different lengths comparable.

For historical reasons, scientists in natural language processing (NLP) prefer to use a quantity called **perplexity**, which is the exponential of the **cross-entropy loss**:

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{t-1}, \ldots, x_1)\right).$$

Perplexity can be best understood as the **harmonic mean** of the **number of real choices** that we have when deciding which token to pick next.

Let us look at a number of cases:

- In the **best** case scenario, the model always perfectly estimates the **probability** of the label token as **1**, and in this case the **perplexity of the model is 1**.
- In the **worst** case scenario, the model always predicts the **probability** of the label token as **0**, and in this situation, the **perplexity is positive infinity**.
- At the baseline, the model **predicts a uniform distribution** over all the available tokens of the vocabulary, and in this case, the **perplexity equals the number of unique tokens** of the vocabulary. In fact, if we were to store the sequence without any compression, this would be the best we could do to encode it. Hence, this provides a nontrivial upper bound that any useful model must beat.

In the following sections, we will implement RNNs for character-level language models and use perplexity to evaluate such models.

Summary

- A neural network that uses recurrent computation for hidden states is called a **recurrent neural network (RNN)**.
- The **hidden state** of an RNN can capture historical information of the sequence up to the current time step.
- The **number of RNN model parameters** does **not grow** as the number of time steps increases.
- Later we try to create **character-level language** models using an RNN.
- We can use **perplexity to evaluate** the quality of language models.

8.5.Recurrent Neural Networks -> from Scratch

Let's implement an RNN **from scratch** for a **character-level language model**, according to our descriptions above.

The model will be trained on H. G. Wells' *The Time Machine* "dataset".

As before, let's start by reading the dataset first.

```
%matplotlib inline
import math
import tensorflow as tf
from d2l import tensorflow as d2l
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.amazor
```

```
train_random_iter, vocab_random_iter = d2l.load_data_time_machine(
    batch_size, num_steps, use_random_iter=True)
```

One-Hot Encoding

Recall that each token is represented as a numerical index in `train_iter`.

Feeding these indices directly to a neural network might make it hard to learn.

We often represent each token as a more expressive feature vector.

The easiest representation is called **one-hot encoding**:

- map each index to a different unit vector: assume that the number of different tokens in the vocabulary is N (`len(vocab)`), and the token indices range from 0 to $N-1$,

- if the index of a token is the integer i, then we create a vector of all 0s with a length of N and set the element at position i to 1.

This vector is the one-hot vector of the original token.

The one-hot vectors with indices 0 and 2 are shown below.

```
tf.one_hot(tf.constant([0, 2]), len(vocab))
```

```
<tf.Tensor: shape=(2, 28), dtype=float32, numpy=
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)>
```

The shape of the minibatch that we sample each time is (batch size, number of time steps).

The `one_hot` function transforms such a minibatch into a three-dimensional tensor with the last dimension equals to the vocabulary size (`len(vocab)`).

We often transpose the input so that we will obtain an output of shape (number of time steps, batch size, vocabulary size).

This will allow us to more conveniently loop through the outermost dimension for updating hidden states of a minibatch, time step by time step.

```
X = tf.reshape(tf.range(10), (2, 5))
tf.one_hot(tf.transpose(X), 28).shape
```

```
TensorShape([5, 2, 28])
```

Initializing the Model Parameters

Next, we initialize the model parameters for the RNN model.

The number of hidden units `num_hiddens` is a tunable hyperparameter.

When training language models, the inputs and outputs are from the same vocabulary.

Hence, they have the same dimension, which is equal to the vocabulary size.

```
def get_params(vocab_size, num_hiddens):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return tf.random.normal(shape=shape, stddev=0.01, mean=0,
                                dtype=tf.float32)

    # Hidden layer parameters
    W_xh = tf.Variable(normal((num_inputs, num_hiddens)), dtype=tf.float32)
```

```
    W_hh = tf.Variable(normal((num_hiddens, num_hiddens)), dtype=tf.float32)
    b_h = tf.Variable(tf.zeros(num_hiddens), dtype=tf.float32)
    # Output layer parameters
    W_hq = tf.Variable(normal((num_hiddens, num_outputs)), dtype=tf.float32)
    b_q = tf.Variable(tf.zeros(num_outputs), dtype=tf.float32)
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    return params
```

RNN Model

To define an RNN model, we first need an `init_rnn_state` function to return the hidden state at initialization.

It returns a tensor filled with 0 and with a shape of (batch size, number of hidden units).

Using tuples makes it easier to handle situations where the hidden state contains multiple variables, which we will encounter in later sections.

```
def init_rnn_state(batch_size, num_hiddens):
    return (tf.zeros((batch_size, num_hiddens)),)
```

The following `rnn` function defines how to compute the hidden state and output at a time step.

NOTE: RNN model loops through the outermost dimension of `inputs` so that it updates hidden states `H` of a minibatch, time step by time step.

Besides, the activation function here uses the $\tanh$ function. The mean value of the $\tanh$ function is 0, when the elements are uniformly distributed over the real numbers.

```
def rnn(inputs, state, params):
    # Here `inputs` shape: (`num_steps`, `batch_size`, `vocab_size`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # Shape of `X`: (`batch_size`, `vocab_size`)
    for X in inputs:
        X = tf.reshape(X, [-1, W_xh.shape[0]])
        H = tf.tanh(tf.matmul(X, W_xh) + tf.matmul(H, W_hh) + b_h)
        Y = tf.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return tf.concat(outputs, axis=0), (H,)
```

With all the needed functions being defined, next we will:

- create a class to wrap these functions, and
- store parameters for an RNN model implemented from scratch.

```
class RNNModelScratch:
    """A RNN Model implemented from scratch."""
```

```
    def __init__(self, vocab_size, num_hiddens, init_state, forward_fn,
                 get_params):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.init_state, self.forward_fn = init_state, forward_fn
        self.trainable_variables = get_params(vocab_size, num_hiddens)

    def __call__(self, X, state):
        X = tf.one_hot(tf.transpose(X), self.vocab_size)
        X = tf.cast(X, tf.float32)
        return self.forward_fn(X, state, self.trainable_variables)

    def begin_state(self, batch_size, *args, **kwargs):
        return self.init_state(batch_size, self.num_hiddens)
```

Let us check whether the outputs have the correct shapes, e.g., to ensure that the dimensionality of the hidden state remains unchanged.

```
# defining tensorflow training strategy
device_name = d2l.try_gpu()._device_name
strategy = tf.distribute.OneDeviceStrategy(device_name)

num_hiddens = 512
with strategy.scope():
    net = RNNModelScratch(len(vocab), num_hiddens, init_rnn_state, rnn,
                          get_params)
state = net.begin_state(X.shape[0])
Y, new_state = net(X, state)
Y.shape, len(new_state), new_state[0].shape
```

```
(TensorShape([10, 28]), 1, TensorShape([2, 512]))
```

We can see that the output shape is

$$\text{(number of time steps} \times \text{batch size, vocabulary size)},$$

while the hidden state shape remains the same, i.e.,

$$\text{(batch size, number of hidden units)}.$$

Prediction

Let's first define the **prediction** function **to generate new characters** following the user-provided `prefix`, which is a string containing several characters.

When looping through these beginning characters in `prefix`, we keep passing the hidden state to the next time step without generating any output.

This is called the **warm-up** period, during which the model updates itself (e.g., update the hidden state) but does not make predictions.

After the warm-up period, the hidden state is generally better than its initialized value at the beginning.

So we generate the predicted characters and emit them.

```
def predict_ch8(prefix, num_preds, net, vocab):
    """Generate new characters following the `prefix`."""
    state = net.begin_state(batch_size=1, dtype=tf.float32)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: tf.reshape(tf.constant([outputs[-1]]), (1, 1)).numpy()
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.numpy().argmax(axis=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

Now we can test the `predict_ch8` function.

We specify the prefix as `time traveller` and have it generate 10 additional characters.

Given that we have not trained the network, it will generate nonsensical predictions.

```
predict_ch8('time traveller ', 10, net, vocab)
```

```
'time traveller zigbnumjem'
```

Gradient Clipping

For a sequence of length T, we compute the gradients over these T time steps in an iteration, which results in a chain of matrix-products with length $\mathcal{O}(T)$ during backpropagation.

It might result in numerical instability, e.g., the gradients may either explode or vanish, when T is large. Therefore, RNN models often need extra help to stabilize the training.

Generally speaking, when solving an optimization problem, we take update steps for the model parameter, say in the vector form $\mathbf{x}$, in the direction of the negative gradient $\mathbf{g}$ on a minibatch.

Example: with $\eta > 0$ as the learning rate, in one iteration we update $\mathbf{x}$ as $\mathbf{x} - \eta\mathbf{g}$.

Let us further assume that the objective function f is well behaved, say, **Lipschitz continuous** with constant L. That is to say, for any $\mathbf{x}$ and $\mathbf{y}$ we have

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|.$$

In this case we can safely assume that if we update the parameter vector by $\eta\mathbf{g}$, then

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|,$$

which means that we will not observe a change by more than $L\eta\|\mathbf{g}\|$.

This is both bad and good side:

- **bad** side -> it **limits the speed** of making progress;
- **good** side -> it **limits the extent** to which things can **go wrong** if we move in the wrong direction.

Sometimes ... and very often:

- the **gradients** can be quite **large**, and
- the **optimization algorithm** may **fail to converge**.

We could address this by reducing the learning rate η.

But what if we only *rarely* get large gradients?

In this case such an approach may appear entirely unwarranted. One popular alternative is to clip the gradient $\mathbf{g}$ by projecting them back to a ball of a given radius, say θ via

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right) \mathbf{g}.$$

By doing so we know that:

- the **gradient norm never exceeds** θ,
- the updated **gradient** is entirely **aligned** with the original **direction of** $\mathbf{g}$.

It also gives:

- the desirable side-effect of **limiting the influence any given minibatch** (and within it any given sample) can exert on the parameter vector,
- a certain degree of **robustness** to the model,
- a quick **fix** to the gradient exploding.

While it **does not entirely solve the problem**, it is one of the many techniques to alleviate it.

Below we define a function to clip the gradients of a model that is implemented from scratch or a model constructed by the high-level APIs.

Also note that we compute the gradient norm over all the model parameters.

```
def grad_clipping(grads, theta):
    """Clip the gradient."""
    theta = tf.constant(theta, dtype=tf.float32)
    norm = tf.math.sqrt(
        sum((tf.reduce_sum(grad**2)).numpy() for grad in grads))
    norm = tf.cast(norm, tf.float32)
    new_grad = []
    if tf.greater(norm, theta):
        for grad in grads:
            new_grad.append(grad * theta / norm)
    else:
        for grad in grads:
            new_grad.append(grad)
    return new_grad
```

Training

Before training the model, let us define a function to train the model in one epoch.

It differs from how we train the model above by three aspects:

1. **Different sampling** methods for sequential data (random sampling and sequential partitioning) will result in **differences in the initialization of hidden states**.
2. **Clipping the gradients** before updating the model parameters ensures that the **model does not diverge** even when gradients **blow up** at some point during the training process.
3. Using **perplexity** to evaluate the model, because as discussed above this **ensures that sequences of different length are comparable**.

Sampling Specifics

Sequential Partitioning

When **sequential partitioning** is used, we initialize the hidden state only at the beginning of each epoch.

Since the i^{th} subsequence example in the next minibatch is adjacent to the current i^{th} subsequence example, the hidden state at the end of the current minibatch will be used to initialize the hidden state at the beginning of the next minibatch. In this way, **historical information** of the sequence stored in the hidden state might **flow over adjacent subsequences** within an epoch.

However, the **computation** of the hidden state at any point **depends on all the previous minibatches** in the same epoch, which **complicates the gradient computation**.

To **reduce computational cost**, we **detach the gradient** before processing any minibatch so that the **gradient computation** of the hidden state is always **limited** to the **time steps in one minibatch**.

Random Sampling

When using the **random sampling**, we need to re-initialize the hidden state for each iteration since each example is sampled with a random position.

The `updater` is a general function to update the model parameters which can be:

- `d2l.sgd` function implemented from scratch,

or

- the built-in optimization function in a DL framework.

```
def train_epoch_ch8(net, train_iter, loss, updater, use_random_iter):
    """Train a model within one epoch (defined in Chapter 8)."""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # Sum of training loss, no. of tokens
```

```
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # Initialize `state` when either it is the first iteration or
            # using random sampling
            state = net.begin_state(batch_size=X.shape[0], dtype=tf.float32)
        with tf.GradientTape(persistent=True) as g:
            y_hat, state = net(X, state)
            y = tf.reshape(tf.transpose(Y), (-1))
            l = loss(y, y_hat)
        params = net.trainable_variables
        grads = g.gradient(l, params)
        grads = grad_clipping(grads, 1)
        updater.apply_gradients(zip(grads, params))

        # Keras loss by default returns the average loss in a batch
        # l_sum = l * float(tf.size(y).numpy()) if isinstance(
        #     loss, tf.keras.losses.Loss) else tf.reduce_sum(l)
        metric.add(l * tf.size(y).numpy(), tf.size(y).numpy())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

The training function supports an RNN model implemented either from scratch or using high-level APIs.

```
from tqdm import tqdm

def train_ch8(net, train_iter, vocab, num_hiddens, lr, num_epochs, strategy,
              use_random_iter=False):
    """Train a model (defined in Chapter 8)."""
    with strategy.scope():
        loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        updater = tf.keras.optimizers.SGD(lr)
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab)
    # Train and predict
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(net, train_iter, loss, updater,
                                     use_random_iter)
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch + 1, [ppl])
    device = d2l.try_gpu()._device_name
    print(f'perplexity {ppl:.1f}, {speed:.1f} tokens/sec on {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

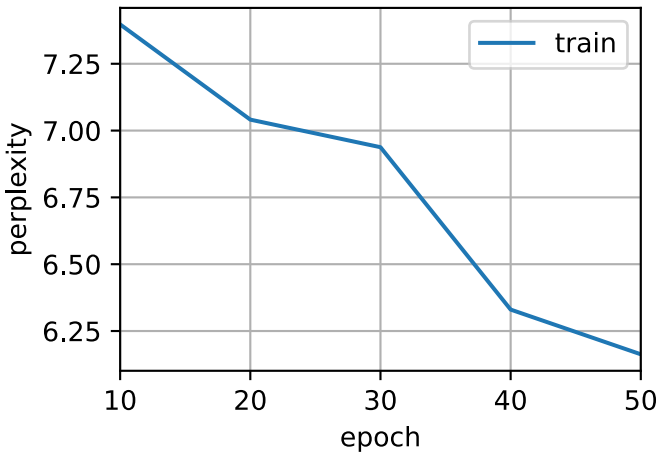
Sequential Partitioning

Now we can train the RNN model.

Since we only use 10000 tokens in the dataset, the model needs more epochs to converge better.

```
%%time
num_epochs, lr = 50, 1  # 500, 1
train_ch8(net, train_iter, vocab, num_hiddens, lr, num_epochs, strategy)
```

```
perplexity 6.2, 5771.6 tokens/sec on /CPU:0
time traveller alle proul that in were the that in that in tha f
traveller all thise and the inding sion and and the time tr
CPU times: user 1min 41s, sys: 4 s, total: 1min 45s
Wall time: 1min 19s
```

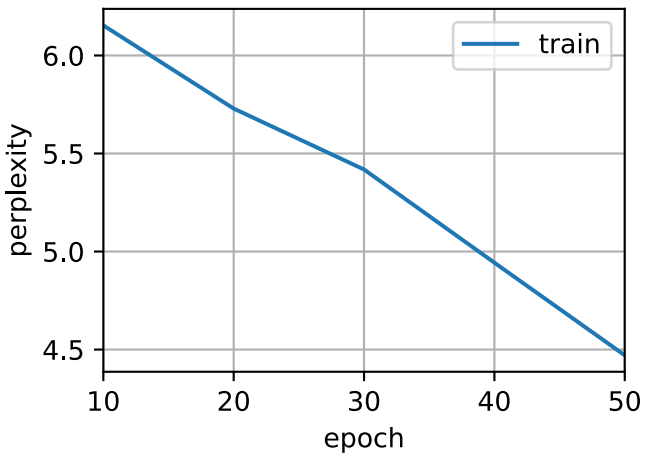


Random Sampling

Finally, let us check the results of using the random sampling method.

```
%%time

train_ch8(net, train_iter, vocab_random_iter, num_hiddens, lr, num_epochs,
          strategy, use_random_iter=True)
```

```
perplexity 4.5, 5668.4 tokens/sec on /CPU:0
time traveller acting and the erdit is the athise muthes bact of
traveller about the and ho wish ard in the medican mare ard
CPU times: user 1min 41s, sys: 4.29 s, total: 1min 46s
Wall time: 1min 20s
```



While implementing the above RNN model from scratch is instructive, it is not convenient.

In the next section we will see how to improve the RNN model, such as how to make it easier to implement and make it run faster.

Summary

- We can train an RNN-based **character-level language** model to generate text following the user-provided text prefix.
- A simple RNN language model consists of:
  - input encoding,
  - RNN modeling,
  - output generation.
- RNN models need state initialization for training, though random sampling and sequential partitioning use different ways.
- When using sequential partitioning, we need to detach the gradient to reduce computational cost.
- A warm-up period allows a model to update itself (e.g., obtain a better hidden state than its initialized value) before making any prediction.
- Gradient clipping prevents gradient explosion, but it cannot fix vanishing gradients.

8.6.Recurrent Neural Networks -> by TensorFlow

While implementation from scratch is useful from pedagogical point of view to see how RNNs are implemented, this is not convenient or fast.

This section will show how to implement the same language model more efficiently using functions provided by high-level APIs of a DL framework like TensorFlow.

We begin as before by reading the time machine dataset.

```
import tensorflow as tf
from d2l import tensorflow as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Defining the Model

High-level APIs provide implementations of recurrent neural networks.

We construct the recurrent neural network layer `rnn_layer` with a single hidden layer and 256 hidden units.

In fact, we have not even discussed yet what it means to have multiple layers---this will happen in the next section.

For now, suffice it to say that multiple layers simply amount to the output of one layer of RNN being used as the input for the next layer of RNN.

```
num_hiddens = 256
rnn_cell = tf.keras.layers.SimpleRNNCell(num_hiddens,
                                         kernel_initializer='glorot_uniform')
rnn_layer = tf.keras.layers.RNN(rnn_cell, time_major=True,
                                return_sequences=True, return_state=True)
```

```
state = rnn_cell.get_initial_state(batch_size=batch_size, dtype=tf.float32)
state.shape
```

```
TensorShape([32, 256])
```

With a hidden state and an input, we can compute the output with the updated hidden state.

It should be emphasized that the "output" (`Y`) of `rnn_layer` does **not** involve computation of output layers:

- it refers to the hidden state at **each** time step,
- they can be used as the input to the subsequent output layer.

```
X = tf.random.uniform((num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, len(state_new), state_new[0].shape
```

```
(TensorShape([35, 32, 256]), 32, TensorShape([256]))
```

Similar to :numref:`sec_rnn_scratch`, we define an `RNNModel` class for a complete RNN model.

Note: `rnn_layer` only contains the hidden recurrent layers, we need to create a separate output layer.

```
class RNNModel(tf.keras.layers.Layer):
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.dense = tf.keras.layers.Dense(vocab_size)

    def call(self, inputs, state):
        X = tf.one_hot(tf.transpose(inputs), self.vocab_size)
        # Later RNN like `tf.keras.layers.LSTMCell` return more than two values
        Y, *state = self.rnn(X, state)
```

```
        output = self.dense(tf.reshape(Y, (-1, Y.shape[-1])))
        return output, state

    def begin_state(self, *args, **kwargs):
        return self.rnn.cell.get_initial_state(*args, **kwargs)
```

Training and Predicting

Before training the model, let us make a prediction with the a model that has random weights.

```
%%time

device_name = d2l.try_gpu()._device_name
strategy = tf.distribute.OneDeviceStrategy(device_name)
with strategy.scope():
    model = RNNModel(rnn_layer, vocab_size=len(vocab))

d2l.predict_ch8('time traveller', 10, model, vocab)
```

```
CPU times: user 102 ms, sys: 2.96 ms, total: 105 ms
Wall time: 106 ms
```

As is quite obvious, this model does not work at all.

Next, we call `train_ch8` with the same hyperparameters defined above (when we create the RMM from scratch) and train our model with high-level APIs.

```
%%time

num_epochs, lr = 50, 1
d2l.train_ch8(model, train_iter, vocab, num_hiddens, lr, num_epochs, strategy)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

Compared with the last section, this model achieves comparable perplexity, albeit within a shorter period of time, due to the code being more optimized by high-level APIs of the deep learning framework.

Summary

- High-level APIs of the deep learning framework provides an implementation of the RNN layer.
- The RNN layer of high-level APIs returns an output and an updated hidden state, where the output does not involve output layer computation.
- Using high-level APIs leads to faster RNN training than using its implementation from scratch.

Part 2.Modern RNN - GRU, LSTM, ...

Above we have introduced:

- the basics of RNNs, which can better handle sequence data,
- RNN-based language models were implemented on text data for demonstration.

BUT ... such techniques **may not be sufficient** for practitioners when they face a wide range of sequence learning problems nowadays.

Potential Problems:

- the numerical instability of RNNs,
- and despite some counter-measures (like gradient clipping) the **more sophisticated designs** of sequence models are possible.

Now the **gated RNNs** are much more common in practice, namely:

- **gated recurrent units (GRUs)**,
- **long short-term memory (LSTM)**.

Below we will consider some expansions that are frequently adopted in modern RNNs:

- **expand** the RNN architecture **with a single unidirectional hidden layer** that has been discussed so far,
- deep architectures with **multiple hidden layers**,
- **bidirectional designs** with both forward and backward recurrent computations.

When explaining these RNN variants, we **continue to consider the same language modeling problem** introduced above.

NOTE: Language modeling reveals only a small fraction of what sequence learning is capable of. In a variety of sequence learning problems, such as automatic speech recognition, text to speech, and machine translation, both inputs and outputs are sequences of arbitrary length. To explain how to fit this type of data, we will take machine translation as an example, and introduce the **encoder-decoder** architecture based on RNNs and **beam search** for sequence generation.

9.1.Gated Recurrent Units (GRU)

Limitations of Previous Approaches

In previous part we discussed:

- how gradients are calculated in RNNs,
- and found that long products of matrices can lead to vanishing or exploding gradients.

Let us briefly think about what such gradient anomalies mean in practice, when we might encounter a situation where:

- An **early** observation is **highly significant** for **predicting all future observations**.

Example: Let's assume the first observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence. In this case, the **influence of the first token is vital**. We would like to have some mechanisms for storing vital early information in a **memory cell**. Without such a mechanism, we will have to assign a very large gradient to this observation, since it affects all the subsequent observations.

- **Some tokens** carry **no pertinent** observation.

Example: Let's assume when parsing a web page there might be auxiliary HTML code that is irrelevant for the purpose of assessing the sentiment conveyed on the page. We would like to have some mechanism for **skipping** such tokens in the latent state representation.

- a **logical break** between parts of a sequence.

Example: Let's assume a transition between chapters in a book, or a transition between a bear and a bull market for securities. In this case it would be nice to have a means of **resetting** our internal state representation.

New Alternatives

Several methods have been proposed to address this:

- **long short-term memory (LSTM)** is one of the earliest methods -> see the details in Schmidhuber(!) et al works below ...

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735–1780. **NOTE: 47266 citations! :) The most influential IT-paper in XX century!**

- **gated recurrent unit (GRU)** is a slightly more streamlined variant that often offers comparable performance and is significantly faster to compute -> see the details in Bengio(!) et al works below ...

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. arXiv preprint arXiv:1409.1259. **NOTE: 3656 citations.**

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555. **NOTE: 6599 citations.**

Gated Hidden State

The key distinction between **vanilla RNNs** and **GRUs** is that the GRUs support **gating** of the hidden state.

This means that we have dedicated mechanisms for:

- when a hidden state should be **updated**,
- and also when it should be **reset**.

These mechanisms are learned and they address the concerns listed above!

Example:

- if the **first token is of great importance** we will **learn not to update the hidden state** after the first observation,
- and, likewise, we will **learn to skip irrelevant** temporary observations,
- and we will learn to **reset the latent state** whenever needed.

Reset Gate and Update Gate

The first thing we need to introduce are*

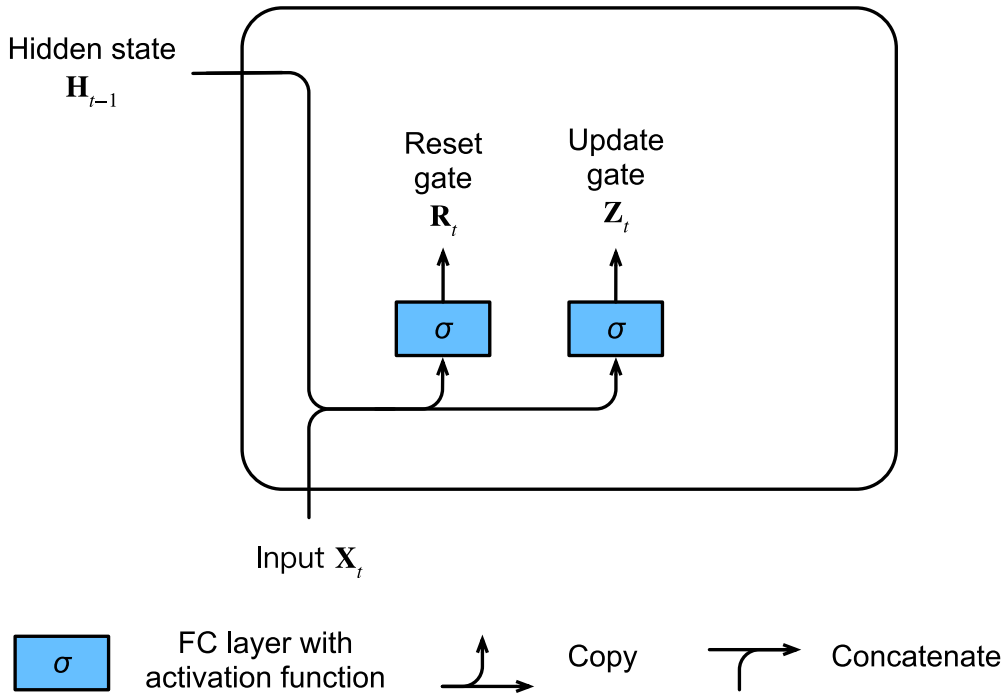
- **reset** gate - would allow us to control **how much of the previous** state we might still want **to remember**,
- **update** gate - would allow us to control **how much of the new** state is just a **copy** of the old state.

We engineer them to be vectors with entries in $(0, 1)$ such that we can perform convex combinations.

Figure below illustrates:

- the **inputs** for both the reset and update gates in a GRU, given:
    - the input of the **current time step**,
    - and the **hidden** state of the **previous time step**.

- the **outputs** of two gates are given by two fully-connected layers with a sigmoid activation function after:
    - the reset gate,
    - the update gate.



Computing the reset gate and the update gate in a GRU model.

Mathematically,for a given time step t, suppose that the input is a minibatch

$$\mathbf{X}_t \in \mathbb{R}^{n \times d}$$

where

n - number of examples, d - number of inputs,

and the hidden state of the previous time step is

$$\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$$

where h - number of hidden units.

Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),$$

where $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ are biases.

We use sigmoid functions to transform input values to the interval $(0, 1)$.

Candidate Hidden State

Next, let us integrate the reset gate $\mathbf{R}_t$ with the regular latent state updating mechanism.

It leads to the following **candidate hidden state** $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ at time step t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h),$$

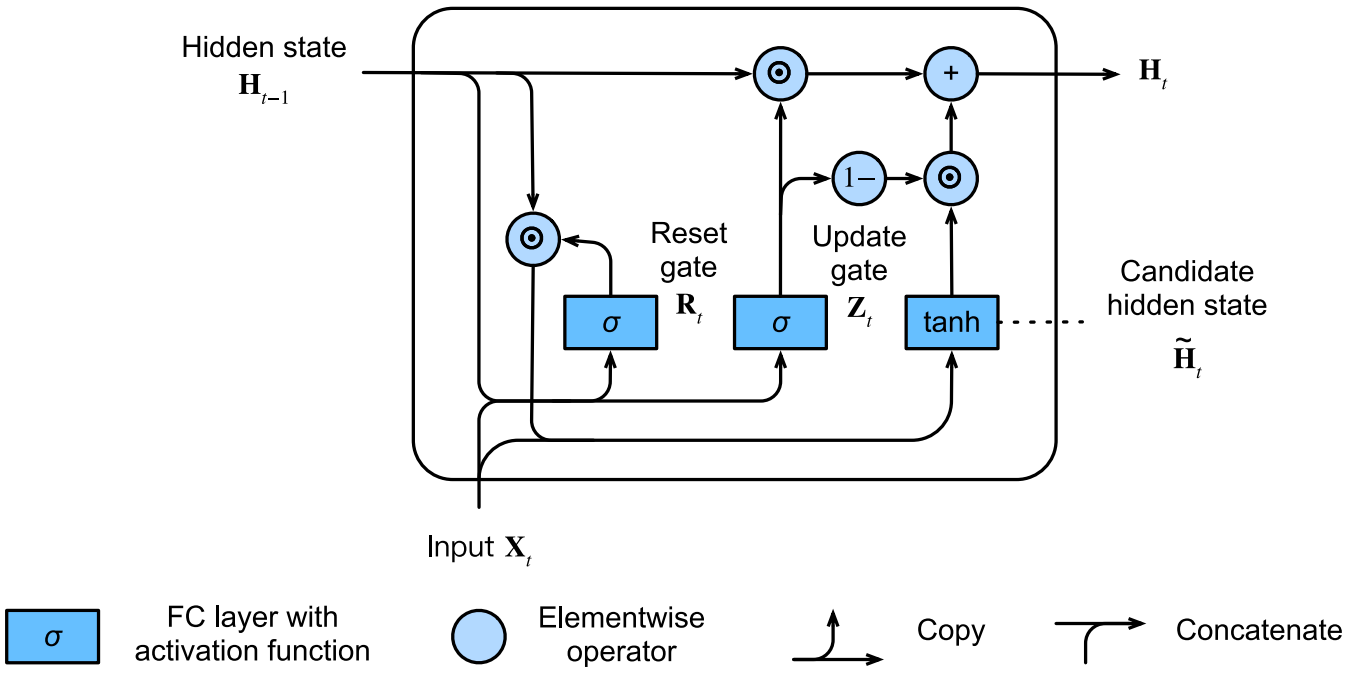
where $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ are weight parameters, $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ is the bias, and the symbol $\odot$ is the Hadamard (elementwise) product operator. Here we use a nonlinearity in the form of tanh to ensure that the values in the candidate hidden state remain in the interval $(-1, 1)$.

The result is a **candidate** since we still need to incorporate the action of the update gate.

Now the influence of the previous states can be reduced with the elementwise multiplication of $\mathbf{R}_t$ and $\mathbf{H}_{t-1}$.

- Whenever the entries in the reset gate $\mathbf{R}_t$ are close to 1, we recover a vanilla RNN.
- For all entries of the reset gate $\mathbf{R}_t$ that are close to 0, the candidate hidden state is the result of an MLP with $\mathbf{X}_t$ as the input. Any pre-existing hidden state is thus **reset** to defaults.

Figure below illustrates the computational flow after applying the **reset gate**.



Computing the candidate hidden state in a GRU model.

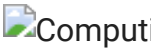
Hidden State

Finally, we need to incorporate the effect of the update gate $\mathbf{Z}_t$. This determines the extent to which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used. The update gate $\mathbf{Z}_t$ can be used for this purpose, simply by taking elementwise convex combinations between both $\mathbf{H}_{t-1}$ and $\tilde{\mathbf{H}}_t$. This leads to the final update equation for the GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

Whenever the update gate $\mathbf{Z}_t$ is close to 1, we simply retain the old state. In this case the information from $\mathbf{X}_t$ is essentially ignored, effectively skipping time step t in the dependency chain. In contrast, whenever $\mathbf{Z}_t$ is close to 0, the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$. These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances. For instance, if the update gate has been close to 1 for all the time steps of an entire subsequence, the old hidden state at the time step of its beginning will be easily retained and passed to its end, regardless of the length of the subsequence.

:numref: `fig_gru_3` illustrates the computational flow after the update gate is in action.

 Computing the hidden state in a GRU model. :label: `fig_gru_3`

In summary, GRUs have the following two distinguishing features:

- Reset gates help capture short-term dependencies in sequences.
- Update gates help capture long-term dependencies in sequences.

Implementation - from Scratch

To gain a better understanding of the GRU model, let us implement it from scratch. We begin by reading the time machine dataset that we used in :numref: `sec_rnn_scratch`. The code for reading the dataset is given below.

```
import tensorflow as tf
from d2l import tensorflow as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.amazor
```

Initializing Model Parameters

The next step is to initialize the model parameters.

We draw the weights from a Gaussian distribution with standard deviation to be 0.01 and set the bias to 0.

The hyperparameter `num_hiddens` defines the number of hidden units.

We instantiate all weights and biases relating to:

- the update gate,
- the reset gate,
- the candidate hidden state, and
- the output layer.

```
def get_params(vocab_size, num_hiddens):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return tf.random.normal(shape=shape, stddev=0.01, mean=0,
                                dtype=tf.float32)

    def three():
        return (tf.Variable(normal((num_inputs, num_hiddens)),
                            dtype=tf.float32),
                tf.Variable(normal((num_hiddens, num_hiddens)),
                            dtype=tf.float32),
                tf.Variable(tf.zeros(num_hiddens), dtype=tf.float32))

    W_xz, W_hz, b_z = three()  # Update gate parameters
    W_xr, W_hr, b_r = three()  # Reset gate parameters
    W_xh, W_hh, b_h = three()  # Candidate hidden state parameters
    # Output layer parameters
    W_hq = tf.Variable(normal((num_hiddens, num_outputs)), dtype=tf.float32)
    b_q = tf.Variable(tf.zeros(num_outputs), dtype=tf.float32)
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    return params
```

Defining the Model

Now we will define the hidden state initialization function `init_gru_state`.

Just like the `init_rnn_state` function defined above, this function returns a tensor with a shape (batch size, number of hidden units) whose values are all zeros.

```
def init_gru_state(batch_size, num_hiddens):
    return (tf.zeros((batch_size, num_hiddens)),)
```

Now we are ready to define the GRU model. Its structure is the same as that of the basic RNN cell, except that the update equations are more complex.

```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        X = tf.reshape(X, [-1, W_xh.shape[0]])
        Z = tf.sigmoid(tf.matmul(X, W_xz) + tf.matmul(H, W_hz) + b_z)
        R = tf.sigmoid(tf.matmul(X, W_xr) + tf.matmul(H, W_hr) + b_r)
        H_tilda = tf.tanh(tf.matmul(X, W_xh) + tf.matmul(R * H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = tf.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return tf.concat(outputs, axis=0), (H,)
```

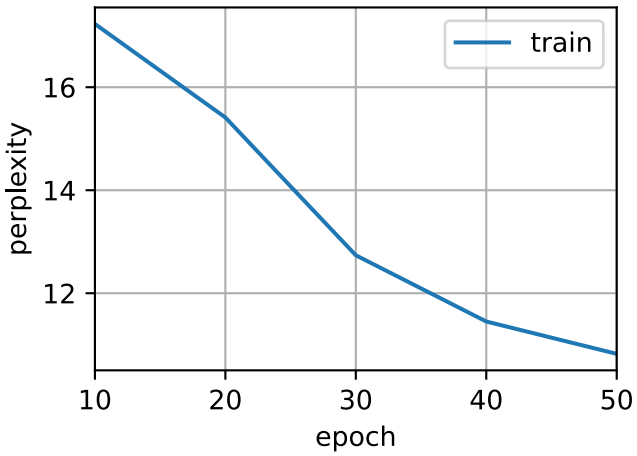
Training and Prediction

Training and prediction work in exactly the same manner as for RNN from scratch before.

After training, we print out the perplexity on the training set and the predicted sequence following the provided prefixes "time traveller" and "traveller", respectively.

```
%%time

vocab_size, num_hiddens, device_name = len(
    vocab), 256, d2l.try_gpu()._device_name
# defining tensorflow training strategy
strategy = tf.distribute.OneDeviceStrategy(device_name)
num_epochs, lr = 50, 1 # 500, 1
with strategy.scope():
    model = d2l.RNNModelScratch(len(vocab), num_hiddens, init_gru_state, gru,
                                get_params)

d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, strategy)
```

```
perplexity 10.8, 7671.1 tokens/sec on /GPU:0
time travellere the the the the the the the the the the the the
travellere the the the the the the the the the the the the
CPU times: user 1min 1s, sys: 677 ms, total: 1min 1s
Wall time: 1min 1s
```



Implementation - by TensorFlow

In high-level APIs, we can directly instantiate a GPU model.

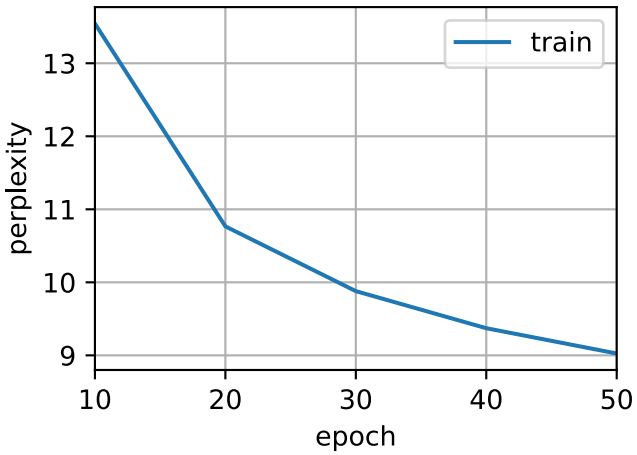
This encapsulates all the configuration detail that we made explicit above.

The code is significantly faster as it uses compiled operators rather than Python for many details that we spelled out before.

```
%%time

gru_cell = tf.keras.layers.GRUCell(num_hiddens,
```

```
                                    kernel_initializer='glorot_uniform')
gru_layer = tf.keras.layers.RNN(gru_cell, time_major=True,
                                return_sequences=True, return_state=True)

device_name = d2l.try_gpu()._device_name
strategy = tf.distribute.OneDeviceStrategy(device_name)
with strategy.scope():
    model = d2l.RNNModel(gru_layer, vocab_size=len(vocab))

d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, strategy)
```

```
perplexity 9.0, 10956.9 tokens/sec on /GPU:0
time traveller and and and and and and and and and and and and and a
traveller and and and and and and and and and and and and and a
CPU times: user 44.5 s, sys: 458 ms, total: 45 s
Wall time: 44.7 s
```



Summary

- Gated RNNs can **better** capture dependencies for sequences with **large time step distances**.
- **Reset** gates help **capture short-term dependencies** in sequences.
- **Update** gates help **capture long-term dependencies** in sequences.
- GRUs contain **basic RNNs** as their **extreme case** whenever the reset gate is switched on, and they can also **skip subsequences by turning on the update** gate.

9.2.Long Short-Term Memory (LSTM)

The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time.

One of the earliest approaches to address this was the long short-term memory (LSTM).

> Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735–1780.

> **NOTE (again): 47266 citations! :) The most influential IT-paper in XX century!**

It shares many of the properties of the GRU.

Interestingly, **LSTMs** have a slightly **more complex** design than GRUs **but predates GRUs by almost two decades!**

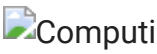
Double-click (or enter) to edit

Gated Memory Cell

Arguably LSTM's design is inspired by logic gates of a computer. LSTM introduces a *memory cell* (or *cell* for short) that has the same shape as the hidden state (some literatures consider the memory cell as a special type of the hidden state), engineered to record additional information. To control the memory cell we need a number of gates. One gate is needed to read out the entries from the cell. We will refer to this as the *output gate*. A second gate is needed to decide when to read data into the cell. We refer to this as the *input gate*. Last, we need a mechanism to reset the content of the cell, governed by a *forget gate*. The motivation for such a design is the same as that of GRUs, namely to be able to decide when to remember and when to ignore inputs in the hidden state via a dedicated mechanism. Let us see how this works in practice.

Input Gate, Forget Gate, and Output Gate

Just like in GRUs, the data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in :numref:`lstm_0`. They are processed by three fully-connected layers with a sigmoid activation function to compute the values of the input, forget. and output gates. As a result, values of the three gates are in the range of $(0, 1)$.


Computing the input gate, the forget gate, and the output gate in an LSTM model.
:label:`lstm_0`

Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$. Correspondingly, the gates at time step t are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n \times h}$, and the output gate is $\mathbf{O}_t \in \mathbb{R}^{n \times h}$. They are calculated as follows:

$$
\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned}
$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ are bias parameters.

Candidate Memory Cell

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the *candidate* memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$. Its computation is similar to that of the three gates described above, but using a $\tanh$ function with a value range for $(-1, 1)$ as the activation function. This leads to the following equation at time step t:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ is a bias parameter.

A quick illustration of the candidate memory cell is shown in :numref:`lstm_1`.


Computing the candidate memory cell in an LSTM model. :label:`lstm_1`

Memory Cell

In GRUs, we have a mechanism to govern input and forgetting (or skipping). Similarly, in LSTMs we have two dedicated gates for such purposes: the input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget gate $\mathbf{F}_t$ addresses how much of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells $\mathbf{C}_{t-1}$ will be saved over time and passed to the current time step. This design is introduced to alleviate the vanishing gradient problem and to better capture long range dependencies within sequences.

We thus arrive at the flow diagram in :numref:`lstm_2`.


Computing the memory cell in an LSTM model.

:label:`lstm_2`

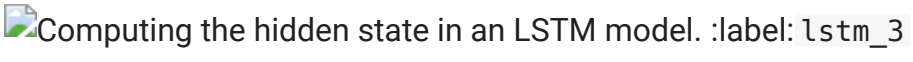
Hidden State

Last, we need to define how to compute the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$. This is where the output gate comes into play. In LSTM it is simply a gated version of the $\tanh$ of the memory cell. This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

Whenever the output gate approximates 1 we effectively pass all memory information through to the predictor, whereas for the output gate close to 0 we retain all the information only within the memory cell and perform no further processing.

:numref:`lstm_3` has a graphical illustration of the data flow.


Computing the hidden state in an LSTM model. :label:`lstm_3`

Implementation - from Scratch

Now let us implement an LSTM from scratch. As same as the experiments in :numref:`sec_rnn_scratch`, we first load the time machine dataset.

```
import tensorflow as tf
from d2l import tensorflow as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

Next we need to define and initialize the model parameters. As previously, the hyperparameter `num_hiddens` defines the number of hidden units. We initialize weights following a Gaussian distribution with 0.01 standard deviation, and we set the biases to 0.

```
def get_lstm_params(vocab_size, num_hiddens):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return tf.Variable(
            tf.random.normal(shape=shape, stddev=0.01, mean=0,
                             dtype=tf.float32))

    def three():
        return (normal(
            (num_inputs, num_hiddens)), normal((num_hiddens, num_hiddens)),
                tf.Variable(tf.zeros(num_hiddens), dtype=tf.float32))

    W_xi, W_hi, b_i = three()  # Input gate parameters
    W_xf, W_hf, b_f = three()  # Forget gate parameters
    W_xo, W_ho, b_o = three()  # Output gate parameters
    W_xc, W_hc, b_c = three()  # Candidate memory cell parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = tf.Variable(tf.zeros(num_outputs), dtype=tf.float32)
    # Attach gradients
    params = [
        W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
```

```
        W_hq, b_q]
    return params
```

Defining the Model

In the initialization function, the hidden state of the LSTM needs to return an *additional* memory cell with a value of 0 and a shape of (batch size, number of hidden units). Hence we get the following state initialization.

```
def init_lstm_state(batch_size, num_hiddens):
    return (tf.zeros(shape=(batch_size, num_hiddens)),
            tf.zeros(shape=(batch_size, num_hiddens)))
```

The actual model is defined just like what we discussed before: providing three gates and an auxiliary memory cell. Note that only the hidden state is passed to the output layer. The memory cell $\mathbf{C}_t$ does not directly participate in the output computation.

```
def lstm(inputs, state, params):
    W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c, W_hq, b_q
    (H, C) = state
    outputs = []
    for X in inputs:
        X = tf.reshape(X, [-1, W_xi.shape[0]])
        I = tf.sigmoid(tf.matmul(X, W_xi) + tf.matmul(H, W_hi) + b_i)
        F = tf.sigmoid(tf.matmul(X, W_xf) + tf.matmul(H, W_hf) + b_f)
        O = tf.sigmoid(tf.matmul(X, W_xo) + tf.matmul(H, W_ho) + b_o)
        C_tilda = tf.tanh(tf.matmul(X, W_xc) + tf.matmul(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * tf.tanh(C)
        Y = tf.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return tf.concat(outputs, axis=0), (H, C)
```

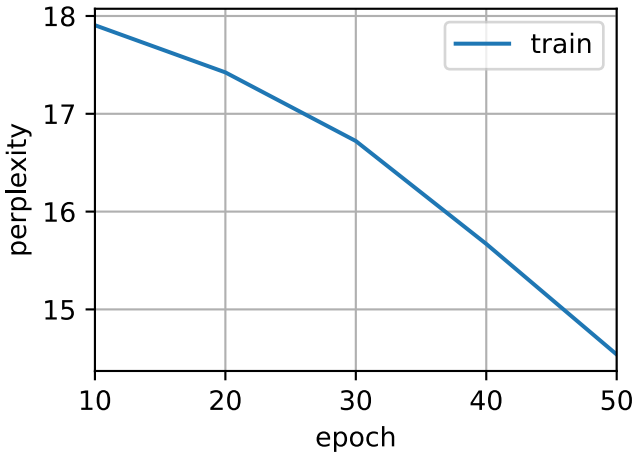
Training and Prediction

Let us train an LSTM as same as what we did in :numref: `sec_gru`, by instantiating the `RNNModelScratch` class as introduced in :numref: `sec_rnn_scratch`.

```
%%time

vocab_size, num_hiddens, device_name = len(
    vocab), 256, d2l.try_gpu()._device_name
num_epochs, lr = 50,1 # 500, 1
strategy = tf.distribute.OneDeviceStrategy(device_name)
with strategy.scope():
```

```
    model = d2l.RNNModelScratch(len(vocab), num_hiddens, init_lstm_state,
                                lstm, get_lstm_params)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, strategy)
```

```
perplexity 14.5, 6879.0 tokens/sec on /GPU:0
time traveller ate at ate at ate at ate at ate at ate at ate at
traveller ate at ate at ate at ate at ate at ate at ate at
CPU times: user 1min 8s, sys: 714 ms, total: 1min 9s
Wall time: 1min 8s
```

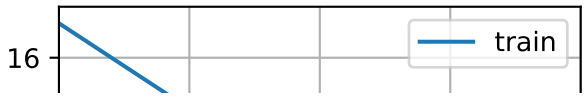


Implementation - by TensorFlow

Using high-level APIs, we can directly instantiate an `LSTM` model. This encapsulates all the configuration details that we made explicit above. The code is significantly faster as it uses compiled operators rather than Python for many details that we spelled out in detail before.

```
%%time

lstm_cell = tf.keras.layers.LSTMCell(num_hiddens,
                                     kernel_initializer='glorot_uniform')
lstm_layer = tf.keras.layers.RNN(lstm_cell, time_major=True,
                                 return_sequences=True, return_state=True)
device_name = d2l.try_gpu()._device_name
strategy = tf.distribute.OneDeviceStrategy(device_name)
with strategy.scope():
    model = d2l.RNNModel(lstm_layer, vocab_size=len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, strategy)
```

```
perplexity 10.3, 13837.1 tokens/sec on /GPU:0
time traveller the the the the the the the the the the the the t
traveller the the the the the the the the the the the the t
CPU times: user 34.3 s, sys: 412 ms, total: 34.7 s
Wall time: 34.4 s
```



LSTMs are the prototypical latent variable autoregressive model with nontrivial state control.

Many variants thereof have been proposed over the years, e.g., multiple layers, residual connections, different types of regularization.

However, training LSTMs and other sequence models (such as GRUs) are quite costly due to the long range dependency of the sequence.

Later we will encounter alternative models such as transformers that can be used in some cases.

Summary

- LSTMs have three types of gates: input gates, forget gates, and output gates that control the flow of information.
- The hidden layer output of LSTM includes the hidden state and the memory cell. Only the hidden state is passed into the output layer. The memory cell is entirely internal.
- LSTMs can alleviate vanishing and exploding gradients.

9.3.Deep Recurrent Neural Networks

Up to now, we only discussed RNNs with a single unidirectional hidden layer. In it the specific functional form of how latent variables and observations interact is rather arbitrary. This is not a big problem as long as we have enough flexibility to model different types of interactions. With a single layer, however, this can be quite challenging. In the case of the linear models, we fixed this problem by adding more layers. Within RNNs this is a bit trickier, since we first need to decide how and where to add extra nonlinearity.

In fact, we could stack multiple layers of RNNs on top of each other. This results in a flexible mechanism, due to the combination of several simple layers. In particular, data might be relevant at different levels of the stack. For instance, we might want to keep high-level data about financial market conditions (bear or bull market) available, whereas at a lower level we only record shorter-term temporal dynamics.

Beyond all the above abstract discussion it is probably easiest to understand the family of models we are interested in by reviewing :numref: `fig_deep_rnn`. It describes a deep RNN with

L hidden layers. Each hidden state is continuously passed to both the next time step of the current layer and the current time step of the next layer.

 Architecture of a deep RNN. :label: `fig_deep_rnn`

Functional Dependencies

We can formalize the functional dependencies within the deep architecture of L hidden layers depicted in :numref: `fig_deep_rnn`. Our following discussion focuses primarily on the vanilla RNN model, but it applies to other sequence models, too.

Suppose that we have a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs in each example: d) at time step t. At the same time step, let the hidden state of the l^{th} hidden layer ($l = 1, \ldots, L$) be $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (number of hidden units: h) and the output layer variable be $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q). Setting $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, the hidden state of the l^{th} hidden layer that uses the activation function ϕ_l is expressed as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

:eqlabel: `eq_deep_rnn_H`

where the weights $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$ and $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, together with the bias $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$, are the model parameters of the l^{th} hidden layer.

In the end, the calculation of the output layer is only based on the hidden state of the final L^{th} hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q,$$

where the weight $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ and the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer.

Just as with MLPs, the number of hidden layers L and the number of hidden units h are hyperparameters. In other words, they can be tuned or specified by us. In addition, we can easily get a deep gated RNN by replacing the hidden state computation in :eqref: `eq_deep_rnn_H` with that from a GRU or an LSTM.

Implementation - by TensorFlow

Fortunately many of the logistical details required to implement multiple layers of an RNN are readily available in high-level APIs. To keep things simple we only illustrate the implementation using such built-in functionalities. Let us take an LSTM model as an example. The code is very similar to the one we used previously in :numref: `sec_lstm`. In fact, the only difference is that we specify the number of layers explicitly rather than picking the default of a single layer. As usual, we begin by loading the dataset.

```
import tensorflow as tf
from d2l import tensorflow as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

The architectural decisions such as choosing hyperparameters are very similar to those of :numref:`sec_lstm`. We pick the same number of inputs and outputs as we have distinct tokens, i.e., `vocab_size`. The number of hidden units is still 256. The only difference is that we now select a nontrivial number of hidden layers by specifying the value of `num_layers`.

```
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
device_name = d2l.try_gpu()._device_name
strategy = tf.distribute.OneDeviceStrategy(device_name)
rnn_cells = [tf.keras.layers.LSTMCell(num_hiddens) for _ in range(num_layers)]
stacked_lstm = tf.keras.layers.StackedRNNCells(rnn_cells)
lstm_layer = tf.keras.layers.RNN(stacked_lstm, time_major=True,
                                 return_sequences=True, return_state=True)
with strategy.scope():
    model = d2l.RNNModel(lstm_layer, len(vocab))
```

Training and Prediction

Since now we instantiate two layers with the LSTM model, this rather more complex architecture slows down training considerably.

```
%%time

num_epochs, lr = 50, 2 #500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, strategy)
```

perplexity 11.5, 6568.9 tokens/sec on /GPU:0

Summary

- In deep RNNs, the hidden state information is passed to the next time step of the current layer and the current time step of the next layer.
- There exist many different flavors of deep RNNs, such as LSTMs, GRUs, or vanilla RNNs. Conveniently these models are all available as parts of the high-level APIs of deep learning frameworks.
- Initialization of models requires care. Overall, deep RNNs require considerable amount of work (such as learning rate and clipping) to ensure proper convergence.

9.4.Bidirectional Recurrent Neural Networks

In sequence learning, so far we assumed that our goal is to model the next output given what we have seen so far, e.g., in the context of a time series or in the context of a language model. While this is a typical scenario, it is not the only one we might encounter. To illustrate the issue, consider the following three tasks of filling in the blank in a text sequence:

- I am `___`.
- I am `___` hungry.
- I am `___` hungry, and I can eat half a pig.

Depending on the amount of information available, we might fill in the blanks with very different words such as "happy", "not", and "very". Clearly the end of the phrase (if available) conveys significant information about which word to pick. A sequence model that is incapable of taking advantage of this will perform poorly on related tasks. For instance, to do well in named entity recognition (e.g., to recognize whether "Green" refers to "Mr. Green" or to the color) longer-range context is equally vital. To get some inspiration for addressing the problem let us take a detour to probabilistic graphical models.

Dynamic Programming in Hidden Markov Models

This subsection serves to illustrate the dynamic programming problem. The specific technical details do not matter for understanding the deep learning models but they help in motivating why one might use deep learning and why one might pick specific architectures.

If we want to solve the problem using probabilistic graphical models we could for instance design a latent variable model as follows. At any time step t, we assume that there exists some latent variable h_t that governs our observed emission x_t via $P(x_t \mid h_t)$. Moreover, any transition $h_t \to h_{t+1}$ is given by some state transition probability $P(h_{t+1} \mid h_t)$. This probabilistic graphical model is then a *hidden Markov model* as in :numref:`fig_hmm`.


A hidden Markov model. :label: `fig_hmm`

Thus, for a sequence of T observations we have the following joint probability distribution over the observed and hidden states:

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t), \text{ where } P(h_1 \mid h_0) = P(h_1).$$

:eqlabel: `eq_hmm_jointP`

Now assume that we observe all x_i with the exception of some x_j and it is our goal to compute $P(x_j \mid x_{-j})$, where $x_{-j} = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_T)$. Since there is no latent variable in $P(x_j \mid x_{-j})$, we consider summing over all the possible combinations of choices for $h_1, \ldots, h_T$. In case any h_i can take on k distinct values (a finite number of states), this means that we need to sum over k^T terms---usually mission impossible! Fortunately there is an elegant solution for this: *dynamic programming*.

To see how it works, consider summing over latent variables $h_1, \ldots, h_T$ in turn. According to :eqref: `eq_hmm_jointP`, this yields:

$$\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_2, \ldots, h_T} \underbrace{\left[\sum_{h_1} P(h_1) P(x_1 \mid h_1) P(h_2 \mid h_1)\right]}_{\pi_2(h_2) \stackrel{\text{def}}{=}} P(x_2 \mid h_2) \prod_{t=3}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_3, \ldots, h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2) P(x_2 \mid h_2) P(h_3 \mid h_2)\right]}_{\pi_3(h_3) \stackrel{\text{def}}{=}} P(x_3 \mid h_3) \prod_{t=4}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \ldots \\
&= \sum_{h_T} \pi_T(h_T) P(x_T \mid h_T).
\end{aligned}$$

In general we have the *forward recursion* as

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t) P(x_t \mid h_t) P(h_{t+1} \mid h_t).$$

The recursion is initialized as $\pi_1(h_1) = P(h_1)$. In abstract terms this can be written as $\pi_{t+1} = f(\pi_t, x_t)$, where f is some learnable function. This looks very much like the update equation in the latent variable models we discussed so far in the context of RNNs!

Entirely analogously to the forward recursion, we can also sum over the same set of latent variables with a backward recursion. This yields:

$$
\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot P(h_T \mid h_{T-1}) P(x_T \mid h_T) \\
&= \sum_{h_1, \ldots, h_{T-1}} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_T} P(h_T \mid h_{T-1}) P(x_T \mid h_T)\right]}_{\rho_{T-1}(h_{T-1}) \stackrel{\mathrm{def}}{=}} \\
&= \sum_{h_1, \ldots, h_{T-2}} \prod_{t=1}^{T-2} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_{T-1}} P(h_{T-1} \mid h_{T-2}) P(x_{T-1} \mid h_{T-1}) \rho_{T-1}(h_{T-1})\right]}_{\rho_{T-2}(h_{T-2}) \stackrel{\mathrm{def}}{=}} \\
&= \ldots \\
&= \sum_{h_1} P(h_1) P(x_1 \mid h_1) \rho_1(h_1).
\end{aligned}
$$

We can thus write the *backward recursion* as

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \rho_t(h_t),$$

with initialization $\rho_T(h_T) = 1$. Both the forward and backward recursions allow us to sum over T latent variables in $\mathcal{O}(kT)$ (linear) time over all values of $(h_1, \ldots, h_T)$ rather than in exponential time. This is one of the great benefits of the probabilistic inference with graphical models. It is also a very special instance of a general message passing algorithm :cite:`Aji.McEliece.2000`. Combining both forward and backward recursions, we are able to compute

$$P(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j) \rho_j(h_j) P(x_j \mid h_j).$$

Note that in abstract terms the backward recursion can be written as $\rho_{t-1} = g(\rho_t, x_t)$, where g is a learnable function. Again, this looks very much like an update equation, just running backwards unlike what we have seen so far in RNNs. Indeed, hidden Markov models benefit from knowing future data when it is available. Signal processing scientists distinguish between the two cases of knowing and not knowing future observations as interpolation v.s. extrapolation. See the introductory chapter of the book on sequential Monte Carlo algorithms

Bidirectional Model

If we want to have a mechanism in RNNs that offers comparable look-ahead ability as in hidden Markov models, we need to modify the RNN design that we have seen so far. Fortunately, this is easy conceptually. Instead of running an RNN only in the forward mode starting from the first token, we start another one from the last token running from back to front. *Bidirectional RNNs* add a hidden layer that passes information in a backward direction to more flexibly process such information. :numref:`fig_birnn` illustrates the architecture of a bidirectional RNN with a single hidden layer.


Architecture of a bidirectional RNN. :label:`fig_birnn`

In fact, this is not too dissimilar to the forward and backward recursions in the dynamic programing of hidden Markov models. The main distinction is that in the previous case these equations had a specific statistical meaning. Now they are devoid of such easily accessible interpretations and we can just treat them as generic and learnable functions. This transition epitomizes many of the principles guiding the design of modern deep networks: first, use the type of functional dependencies of classical statistical models, and then parameterize them in a generic form.

Definition

Bidirectional RNNs were introduced by :cite:`Schuster.Paliwal.1997`. For a detailed discussion of the various architectures see also the paper :cite:`Graves.Schmidhuber.2005`. Let us look at the specifics of such a network.

For any time step t, given a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs in each example: d) and let the hidden layer activation function be ϕ. In the bidirectional architecture, we assume that the forward and backward hidden states for this time step are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$, respectively, where h is the number of hidden units. The forward and backward hidden state updates are as follows:

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned}$$

where the weights $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, $\text{and } \mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, and biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h} \text{ and } \mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ are all the model parameters.

Next, we concatenate the forward and backward hidden states $\overrightarrow{\mathbf{H}}_t$ and $\overleftarrow{\mathbf{H}}_t$ to obtain the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ to be fed into the output layer. In deep bidirectional RNNs with multiple hidden layers, such information is passed on as *input* to the next bidirectional layer. Last, the output layer computes the output $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

Here, the weight matrix $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ and the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer. In fact, the two directions can have different numbers of hidden units.

Computational Cost and Applications

One of the key features of a bidirectional RNN is that information from both ends of the sequence is used to estimate the output. That is, we use information from both future and past observations to predict the current one. In the case of next token prediction this is not quite what we want. After all, we do not have the luxury of knowing the next to next token when predicting the next one. Hence, if we were to use a bidirectional RNN naively we would not get a very good accuracy: during training we have past and future data to estimate the present. During test time we only have past data and thus poor accuracy. We will illustrate this in an experiment below.

To add insult to injury, bidirectional RNNs are also exceedingly slow. The main reasons for this are that the forward propagation requires both forward and backward recursions in bidirectional layers and that the backpropagation is dependent on the outcomes of the forward propagation. Hence, gradients will have a very long dependency chain.

In practice bidirectional layers are used very sparingly and only for a narrow set of applications, such as filling in missing words, annotating tokens (e.g., for named entity recognition), and encoding sequences wholesale as a step in a sequence processing pipeline (e.g., for machine translation). In :numref: `sec_bert` and :numref: `sec_sentiment_rnn`, we will introduce how to use bidirectional RNNs to encode text sequences.

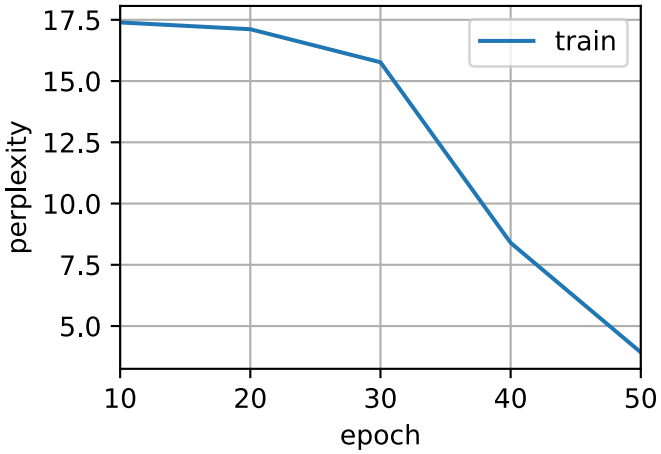
Training a Bidirectional RNN for a Wrong Application

If we were to ignore all advice regarding the fact that bidirectional RNNs use past and future data and simply apply it to language models, we will get estimates with acceptable perplexity. Nonetheless, the ability of the model to predict future tokens is severely compromised as the experiment below illustrates. Despite reasonable perplexity, it only generates gibberish even after many iterations. We include the code below as a cautionary example against using them in the wrong context.

```
import torch
from torch import nn
from d2l import torch as d2l

# Load data
batch_size, num_steps, device = 32, 35, d2l.try_gpu()
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Define the bidirectional LSTM model by setting `bidirectional=True`
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
```

```
%%time

# Train the model
num_epochs, lr = 50, 1 #500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 3.9, 126145.8 tokens/sec on cuda:0
time travellerrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
travellerrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
CPU times: user 4.85 s, sys: 197 ms, total: 5.05 s
Wall time: 5.29 s
```



The output is clearly unsatisfactory for the reasons described above. For a discussion of more effective uses of bidirectional RNNs, please see the sentiment analysis application in :numref:`sec_sentiment_rnn`.

Summary

- In bidirectional RNNs, the hidden state for each time step is simultaneously determined by the data prior to and after the current time step.
- Bidirectional RNNs bear a striking resemblance with the forward-backward algorithm in probabilistic graphical models.
- Bidirectional RNNs are mostly useful for sequence encoding and the estimation of observations given bidirectional context.
- Bidirectional RNNs are very costly to train due to long gradient chains.

9.5.Machine Translation and the Dataset

We have used RNNs to design language models, which are key to natural language processing. Another flagship benchmark is *machine translation*, a central problem domain for *sequence transduction* models that transform input sequences into output sequences. Playing a crucial role in various modern AI applications, sequence transduction models will form the focus of the

remainder of this chapter and :numref:`chap_attention`. To this end, this section introduces the machine translation problem and its dataset that will be used later.

Machine translation refers to the automatic translation of a sequence from one language to another. In fact, this field may date back to 1940s soon after digital computers were invented, especially by considering the use of computers for cracking language codes in World War II. For decades, statistical approaches had been dominant in this field :cite:`Brown.Cocke.Della-Pietra.ea.1988,Brown.Cocke.Della-Pietra.ea.1990` before the rise of end-to-end learning using neural networks. The latter is often called *neural machine translation* to distinguish itself from *statistical machine translation* that involves statistical analysis in components such as the translation model and the language model.

Emphasizing end-to-end learning, this book will focus on neural machine translation methods. Different from our language model problem in :numref:`sec_language_model` whose corpus is in one single language, machine translation datasets are composed of pairs of text sequences that are in the source language and the target language, respectively. Thus, instead of reusing the preprocessing routine for language modeling, we need a different way to preprocess machine translation datasets. In the following, we show how to load the preprocessed data into minibatches for training.

```
import os
import tensorflow as tf
from d2l import tensorflow as d2l
```

Downloading and Preprocessing the Dataset

To begin with, we download an English-French dataset that consists of [bilingual sentence pairs from the Tatoeba Project](). Each line in the dataset is a tab-delimited pair of an English text sequence and the translated French text sequence. Note that each text sequence can be just one sentence or a paragraph of multiple sentences. In this machine translation problem where English is translated into French, English is the *source language* and French is the *target language*.

```
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

def read_data_nmt():
    """Load the English-French dataset."""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[:75])
```

```
Downloading ../data/fra-eng.zip from http://d2l-data.s3-accelerate.amazonaws.
Go.	Va !
Hi.	Salut !
Run!	Cours !
Run!	Courez !
Who?	Qui ?
Wow!	Ça alors !
```

After downloading the dataset, we proceed with several preprocessing steps for the raw text data. For instance, we replace non-breaking space with space, convert uppercase letters to lowercase ones, and insert space between words and punctuation marks.

```
def preprocess_nmt(text):
    """Preprocess the English-French dataset."""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # Replace non-breaking space with space, and convert uppercase letters to
    # lowercase ones
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # Insert space between words and punctuation marks
    out = [
        ' ' + char if i > 0 and no_space(char, text[i - 1]) else char
        for i, char in enumerate(text)]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[:80])
```

```
go .	va !
hi .	salut !
run !	cours !
run !	courez !
who ?	qui ?
wow !	ça alors !
```

Tokenization

Different from character-level tokenization in :numref:`sec_language_model`, for machine translation we prefer word-level tokenization here (state-of-the-art models may use more advanced tokenization techniques). The following `tokenize_nmt` function tokenizes the the first `num_examples` text sequence pairs, where each token is either a word or a punctuation mark. This function returns two lists of token lists: `source` and `target`. Specifically, `source[i]` is a list of tokens from the i^{th} text sequence in the source language (English here) and `target[i]` is that in the target language (French here).

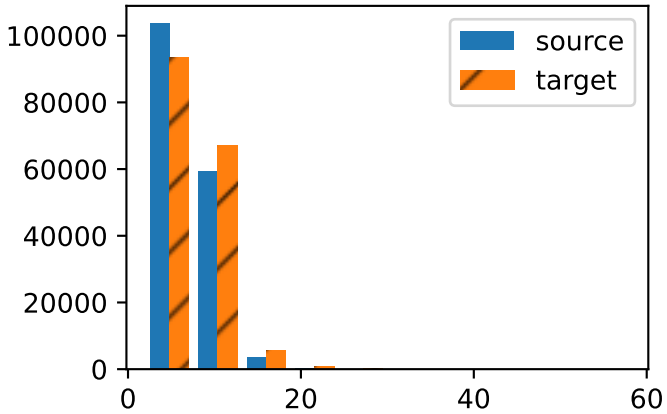
```
def tokenize_nmt(text, num_examples=None):
```

```
    """Tokenize the English-French dataset."""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[:6], target[:6]
```

```
([['go', '.'],
  ['hi', '.'],
  ['run', '!'],
  ['run', '!'],
  ['who', '?'],
  ['wow', '!']],
 [['va', '!'],
  ['salut', '!'],
  ['cours', '!'],
  ['courez', '!'],
  ['qui', '?'],
  ['ça', 'alors', '!']])
```

Let us plot the histogram of the number of tokens per text sequence. In this simple English-French dataset, most of the text sequences have fewer than 20 tokens.

```
d2l.set_figsize()
_, _, patches = d2l.plt.hist([[len(l)
                               for l in source], [len(l) for l in target]],
                             label=['source', 'target'])
for patch in patches[1].patches:
    patch.set_hatch('/')
d2l.plt.legend(loc='upper right');
```



Vocabulary

Since the machine translation dataset consists of pairs of languages, we can build two vocabularies for both the source language and the target language separately. With word-level tokenization, the vocabulary size will be significantly larger than that using character-level tokenization. To alleviate this, here we treat infrequent tokens that appear less than 2 times as the same unknown ("<unk>") token. Besides that, we specify additional special tokens such as for padding ("<pad>") sequences to the same length in minibatches, and for marking the beginning ("<bos>") or end ("<eos>") of sequences. Such special tokens are commonly used in natural language processing tasks.

```
src_vocab = d2l.Vocab(source, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)
```

```
10012
```

Loading the Dataset

Recall that in language modeling each sequence example, either a segment of one sentence or a span over multiple sentences, has a fixed length. This was specified by the `num_steps` (number of time steps or tokens) argument in :numref:`sec_language_model`. In machine translation, each example is a pair of source and target text sequences, where each text sequence may have different lengths.

For computational efficiency, we can still process a minibatch of text sequences at one time by *truncation* and *padding*. Suppose that every sequence in the same minibatch should have the same length `num_steps`. If a text sequence has fewer than `num_steps` tokens, we will keep appending the special "<pad>" token to its end until its length reaches `num_steps`. Otherwise, we will truncate the text sequence by only taking its first `num_steps` tokens and discarding the remaining. In this way, every text sequence will have the same length to be loaded in minibatches of the same shape.

The following `truncate_pad` function truncates or pads text sequences as described before.

```
def truncate_pad(line, num_steps, padding_token):
    """Truncate or pad sequences."""
    if len(line) > num_steps:
        return line[:num_steps]  # Truncate
    return line + [padding_token] * (num_steps - len(line))  # Pad

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])
```

```
[47, 4, 1, 1, 1, 1, 1, 1, 1, 1]
```

Now we define a function to transform text sequences into minibatches for training. We append the special “<eos>” token to the end of every sequence to indicate the end of the sequence. When a model is predicting by generating a sequence token after token, the generation of the “<eos>” token can suggest that the output sequence is complete. Besides, we also record the length of each text sequence excluding the padding tokens. This information will be needed by some models that we will cover later.

```
def build_array_nmt(lines, vocab, num_steps):
    """Transform text sequences of machine translation into minibatches."""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = torch.tensor([
        truncate_pad(l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
    return array, valid_len
```

Putting All Things Together

Finally, we define the `load_data_nmt` function to return the data iterator, together with the vocabularies for both the source language and the target language.

```
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """Return the iterator and the vocabularies of the translation dataset."""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

Let us read the first minibatch from the English-French dataset.

```
train_iter, src_vocab, tgt_vocab = load_data_nmt(batch_size=2, num_steps=8)
for X, X_valid_len, Y, Y_valid_len in train_iter:
    print('X:', tf.cast(X, tf.int32))
    print('valid lengths for X:', X_valid_len)
    print('Y:', tf.cast(Y, tf.int32))
    print('valid lengths for Y:', Y_valid_len)
    break
```

```
X: tf.Tensor(
[[86  8  4  3  1  1  1  1]
 [ 7  0  4  3  1  1  1  1]], shape=(2, 8), dtype=int32)
```

```
valid lengths for X: tf.Tensor([4 4], shape=(2,), dtype=int64)
Y: tf.Tensor(
[[0 4 3 1 1 1 1 1]
 [6 7 0 4 3 1 1 1]], shape=(2, 8), dtype=int32)
valid lengths for Y: tf.Tensor([3 5], shape=(2,), dtype=int64)
```

Summary

- Machine translation refers to the automatic translation of a sequence from one language to another.
- Using word-level tokenization, the vocabulary size will be significantly larger than that using character-level tokenization. To alleviate this, we can treat infrequent tokens as the same unknown token.
- We can truncate and pad text sequences so that all of them will have the same length to be loaded in minibatches.