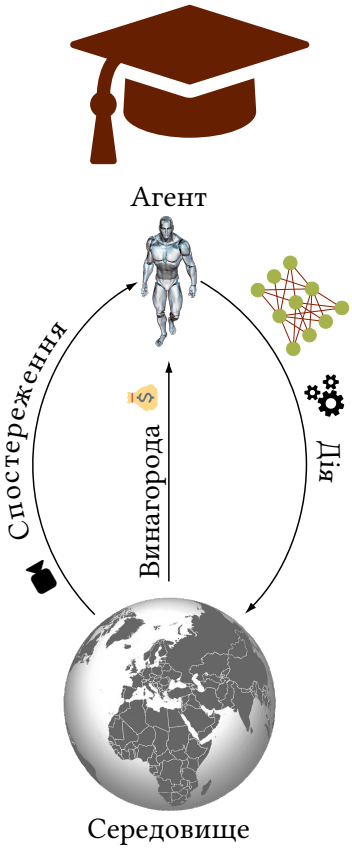
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

# Навчання нейронних мереж з підкріпленням

Методичні вказівки для виконання практичних робіт | Осінній семестр



## Інструктор


КОЧУРА Юрій Петрович

Кафедра ОТ, ФІОТ

@y_kochura

iuriy.kochura@gmail.com


## Особливості

Для магістрів 2-го курсу, осінь

123 – “Комп’ютерна інженерія”

Нормативна

Очна форма навчання

Українська, англійська

6 кредитів ЄКТС

7 лекцій

3 практичні роботи + проєкт

Екзамен

Розміщення курсу

## Опис

Навчання з підкріпленням (англ. reinforcement learning, RL) — це галузь машинного навчання, а також формалізм для автоматизованого прийняття рішень на основі взаємодій. За останні 5 років глибинне навчання з підкріпленням (deep RL) стало одним з найінтенсивніших напрямків досліджень у сфері штучного інтелекту. Сьогодні deep RL дозволяє досягати надлюдської продуктивності в ряді завдань: відео ігри, покер, а також у настільних іграх, включаючи ґо та шахи.

Цей курс познайомить Вас з сімейством статистичних алгоритмів, які вивчають оптимальну стратегію, метою якої є максимізація загальної винагороди, отриманої агентом при взаємодії з навколишнім середовищем.

## Потрібні навички

Для проходження цього курсу потрібно володіти наступними навичками:

- Базовий рівень володіння англійською мовою не нижче A2.
- Базові знання з лінійної алгебри та теорії ймовірностей.
- Досвід тренування глибинних мереж (ініціалізація, оптимізація, регуляризація, вибір методу та метрик для оцінки).

## Система оцінювання

| | |
|---|---|
| 30% | Практичні завдання (10% кожне) |
| 40% | Проєкт |
| 30% | Екзамен |

Важливо! Умова допуску до семестрового контролю (екзамену):

$$\text{Практичні завдання} + \text{Проєкт} \geq 42\%$$

Шкала оцінок КПІ ім. Ігоря Сікорського:

| | |
|---|---|
| A = 95–100 | Відмінно |
| B = 85–94 | Дуже добре |
| C = 75–84 | Добре |
| D = 65–74 | Задовільно |
| E = 60–64 | Достатньо |
| F < 60 | Незадовільно |
| Fx < 42 | Недопущений |
| Порушення кодексу честі | Усунений |

## Кодекс честі

Ви можете обговорювати завдання практичних робіт у групах. Однак, кожен студент/студентка повинен/повинна підготувати розв’язки завдань самостійно.

Під час проходження цього курсу Ви зобов’язані дотримуватись Кодекс честі КПІ ім. Ігоря Сікорського та усі наступні правила:

1. Кожен з Вас повинен відправляти на перевірку власно виконану роботу. Використання чужих розв’язків або програмного коду і представлення їх за свої напрацювання є плагіатом та серйозним порушенням основних академічних стандартів.
2. Ви не повинні ділитися своїми розв’язками з іншими студентами, а також просити інших ділитися своїми розв’язками з Вами.
3. Якщо Ви отримували допомогу у вирішенні певного завдання, Ви маєте вказати це у звіті, а саме: від кого та яку допомогу отримали.

# Практична 1: Перевернутий маятник (Cartpole), понг (Pong) та MountainCar

*"The difference between stupidity and genius is that genius has its limits."*
*"Різниця між геніальністю і дурістю в тому, що у першої є свої межі."*

– Альберт Ейнштейн

## Вступ

Виконуючи це завдання, Ви познайомитесь з тим як можна створити середовище різної складності та навчити агента дотримуватись в ньому певної стратегії.

## Опис завдання

**Завдання агента** – оптимізувати (максимізувати) загальну винагороду, отриману ним при взаємодії з навколишнім середовищем. На кожному кроці в момент часу $t$ агент:

- Отримує спостереження $O_t$ та винагороду $R_t$
- Виконує дію $A_t$

Середовище:

- Отримує дію $A_t$
- Продукує спостереження $O_{t+1}$ та винагороду $R_{t+1}$

**Винагорода** – це скалярний сигнал, який отримує агент у якості зворотного зв'язку від середовища. Вона показує, наскільки добре працює агент у момент часу $t$ відповідно до поставленої мети. Формально агент має максимізувати кумулятивну винагороду:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots \tag{1}$$

- $G_t$ називається загальною винагородою (return) – сума всіх винагород, які агент розраховує отримати при дотриманні стратегії від певного стану до кінця епізоду[1].

Навчання з підкріпленням базується на гіпотезі винагороди:

> "Будь-яка мета може бути формалізована як результат максимізації сукупної винагороди."

[1] Епізод – кожна спроба агента вивчити середовище.

Функції цінності:

- Цінність станів

  Очікувана сукупна винагорода, яку агент розраховує отримати від стану $s$ називається **цінністю (value)**:

$$v(s) = \mathbb{E}\left[G_t \mid S_t = s\right] = \\ = \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots \mid S_t = s\right] \tag{2}$$

- Цінність дій – Q-функція[2]

  Q-функція дозволяє оцінити цінність (якість) дій:

$$q(s,a) = \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right] = \\ = \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots \mid S_t = s, A_t = a\right] \tag{3}$$

Навчання з підкріпленням фундаментально відрізняється від контрольованого (з учителем) та неконтрольованого (без учителя) / напів-контрольованого навчання, оскільки ми тренуємо модель управляти діями нашого агента з метою знаходження ним оптимальної послідовності дій, які приведуть його до вирішення поставленого завдання з максимальним кінцевим значенням загальної винагороди. Іншими словами метою навчання агента є визначення найкращого наступного стану у який має перейти агент для отримання найбільшої кінцевої винагороди.

У цій практичній роботі ми сфокусуємось на створенні алгоритму навчання з підкріпленням для вивчення трьох різних середовищ з різним рівнем складності:

1. Перевернутий маятник (Cartpole): балансування стовпа, що стоїть на візку у вертикальному положенні, пересуваючи візок вліво та вправо.
2. Понг (Pong): виграйте ваших опонентів (інший штучний інтелект або людей) у грі Pong. Великопросторове середовище спостереження – навчання напряму з піксельних даних.
3. MountainCar: агент (автомобіль) знаходиться між двома пагорбами, його завдання заїхати на пагорб праворуч до прапорця, проте двигун агента недостатньо потужний, щоб піднятися на гору просто рухаючись вперед. Тому єдиний спосіб досягти успіху – це рухатися вперед-назад, щоб набрати потрібних обертів.

Для того, щоб змоделювати вище згадані середовища, ми будемо використовувати API Gym, розроблений компанією OpenAI. Gym пропонує кілька попередньо визначених середовищ для тренування та тестування агентів навчання з підкріпленням, включаючи класичні завдання для фізичного контролю, відеоігри Atari, а також симуляції роботів. Для інсталяції базової бібліотеки Gym використовуйте:

```
pip install gym
```

Додаткові деталі щодо інсталяції можна знайти тут: https://github.com/openai/gyminstallation. Google Colab має попередньо інстальовану Gym бібліотеку, версію якої можна перевірити наступним чином:

```
import gym
gym.__version__
```

У перевернутого маятника стовп прикріпленій за допомогою шарніру до візка, який може рухатись вліво-вправо без тертя. На початку епізоду стовп знаходиться у вертикальному положенні і метою є запобігання його падіння. Система контролюється шляхом прикладання сили зі значенням +1 чи -1 до візка. Винагорода

[2] Q – перша літеру англійського слова Quality, означає якість.

+1 надається за кожен момент часу, коли стовп залишається вертикальним. Епізод закінчується, коли стовп відхиляється більше ніж на 15 градусів від вертикалі, або ж якщо візок від'їзжає більше ніж на 2.4 одиниці від центра. Візуальна репрезентація візка зображена на рисунку 1.

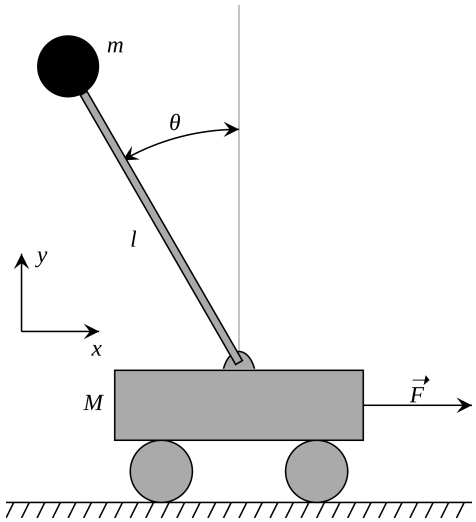



**Рис. 1:** Перевернутий маятник

Для ініціалізації середовища потрібно викликати функцію `make` з бібліотеки `gym` та передати в якості параметра назву потрібного середовища. Однією з проблем з якою ми можемо зіштовхнутися створюючи алгоритми навчання з підкріпленням, це те, що багато аспектів навчання за своєю сутністю є випадковими: ініціалізація стану гри, зміни в середовищі, дії агента. Тому гарною ідеєю є встановлення початкового `seed` для середовища, аби забезпечити деякий рівень контролю. Так само як можна використовувати `numpy.random.seed`, ми викликаємо таку саму функцію у `gym` з нашим середовищем, для того аби наше середовище мало однакові випадкові величини при різних запусках:

```
### Instantiate the Cartpole environment ###

env = gym.make("CartPole-v1")
env.seed(1)
```

Визначивши середовище та цілі гри, ми можемо подумати про те, які :

1. спостереження можуть допомогти визначити стан середовища
2. дії агент може виконувати у цьому середовищі

Спочатку, подумаємо про спостереження. У цьому середовищі нашими спостереженнями будуть:

1. позиція візка
2. швидкість візка
3. кут стовпа
4. швидкість повороту стовпа

Для того, щоб перевірити розмір простору, виведемо його на екран:

```
n_observations = env.observation_space
print("Environment has observation space =", n_observations)
```

```
Environment has observation space = Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
```

Далі, ми розглядаємо простір дій. На кожному кроці, агент може пересуватися або вліво або вправо. Ми знову ж таки можемо перевірити розмір простору, вивівши значення на екран:

```
n_actions = env.action_space.n
print("Number of possible actions that the agent can choose from =", n_actions)
```

```
Number of possible actions that the agent can choose from = 2
```

Тепер, коли ми задали середовище та зрозуміли розмірність просторів спостереження та дій агент, можемо визначати нашого агента. У глибинному навчанні з підкріпленням, агент задається глибинною нейронною мережею. Ця мережа буде брати на вхід спостереження середовища та видавати ймовірності виконання кожної з можливих дій. Оскільки Cartpole є простором з малим простором спостережень, для агента підійде проста повнозв'язна (feed-forward) нейронна мережа. Ми задамо її за допомогою Sequential API:

```
### Define the Cartpole agent ###

# Defines a feed-forward neural network
def create_cartpole_model():
    model = tf.keras.models.Sequential([
        # First Dense layer
        tf.keras.layers.Dense(units=32, activation='relu'),

        # Think about the space the agent needs to act in!
        tf.keras.layers.Dense(units=n_actions, activation=None)

    ])
    return model

cartpole_model = create_cartpole_model()
```

Тепер, коли ми задали основну архітектуру мережі, ми визначимо функцію дій, що виконуватиме прямий прохід по мережі на основі множини спостережень, а також збирає вихід з мережі. На основі цього буде прийматися рішення про наступний крок агента.

```
### Define the agent's action function ###

# Function that takes observations as input, executes a forward pass through model,
#   and outputs a sampled action.
# Arguments:
#   model: the network that defines our agent
#   observation: observation(s) which is/are fed as input to the model
#   single: flag as to whether we are handling a single observation or batch of
#     observations, provided as an np.array
# Returns:
#   action: choice of agent action
def choose_action(model, observation, single=True):
    # add batch dimension to the observation if only a single example was provided
    observation = np.expand_dims(observation, axis=0) if single else observation

    ''' Feed the observations through the model to predict the log probabilities of each possible action.'
    logits = model.predict(observation)

    '''Choose an action from the categorical distribution defined by the log
       probabilities of each possible action.'''
    action = tf.random.categorical(logits, num_samples=1)

    action = action.numpy().flatten()
```

```
    return action[0] if single else action
```

Тепер, коли ми визначили середовище та архітектуру мережі агента, а також функцію дій, ми готові перейти до наступного кроку:

1. **Ініціалізація середовища та агента:** тут ми опишемо різні спостереження та дії, що їх може виконати агент у середовищі.
2. **Визначення пам'яті агента**: це дасть змогу агенту запам'ятовувати свої попередні дії та винагороди.
3. **Задання алгоритму навчання**: за допомогою цього буде виконано підкріплення хороших дій та покарання поганих дій агента.

У навчанні з підкріпленням, тренування виконується завдяки взаємодії агента з середовищем та виконання дій агентом; епізодом називається послідовність дій, що закінчується у якомусь кінцевому стані, як-от падіння стовпа чи виїзд візка за межі. Агенту потрібно буде пам'ятати усі свої дії та спостереження, так що коли закінчується епізод, він зможе "підкріпити" хороші дії та "покарати" погані. Першим кроком є задання простого буферу пам'яті, що зберігає спостереження агента, його дії, а також отримані винагороди за конкретні епізоди.

```
### Agent Memory ###

class Memory:
    def __init__(self):
        self.clear()

  # Resets/restarts the memory buffer
    def clear(self):
        self.observations = []
        self.actions = []
        self.rewards = []

  # Add observations, actions, rewards to memory
    def add_to_memory(self, new_observation, new_action, new_reward):
        self.observations.append(new_observation)
        '''Update the list of actions with new action'''
        self.actions.append(new_action)
        # ['''TODO''']
        '''Update the list of rewards with new reward'''
        self.rewards.append(new_reward)


    def __len__(self):
        return len(self.actions)

# Instantiate a single Memory buffer
memory = Memory()
```

Ми майже готові почати навчати нашого агента! Наступним кроком є підрахунок винагород, які агент отримає за свої дії у середовищі. Оскільки ми (та агент) не знає напевно, коли закінчиться гра чи завдання (коли звалиться стовп), має сенс отримувати миттєву винагороду, а не замислюватись над тим, яка винагорода може бути отримана пізніше. Ця ідея схожа з отриманням відсотків від депозиту.

Для підрахунку очікуваної сумарної винагороди отриманої агентом з моменту часу $t$, ми сумуємо винагороду

з фактором знецінювання у межах епізоду навчання з проектуванням у майбутнє:

$$G\_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

- Коефіцієнт знецінювання $\gamma \in [0, 1]$ показує на цінність майбутніх винагород
- Чим менший коефіцієнт знецінювання, тим менше агент замислюється над вигодою від майбутніх своїх дій.

Зверніть увагу на форму цієї суми – нам потрібно буде подумати про те, як її реалізувати. Конкретніше, нам потрібно буде ініціалізувати масив нулями, з довжиною рівною кількості кроків, та заповнювати його реальними значеннями винагороди зі знижкою по ходу того, як ми ітеруємо по винагородам епізоду, що зберігаються у пам'яті агента. Остаточно нам важливо, які дії кращі у порівнянні з іншими діями у цьому епізоді – тому ми нормалізуємо підраховані винагороди за допомогою середнього та стандартного відхилення винагород по всім епізодам.

```
### Reward function ###

# Helper function that normalizes an np.array x
def normalize(x):
    x -= np.mean(x)
    x /= np.std(x)
    return x.astype(np.float32)

# Compute normalized, discounted, cumulative rewards (i.e., return)
# Arguments:
#   rewards: reward at timesteps in episode
#   gamma: discounting factor
# Returns:
#   normalized discounted reward
def discount_rewards(rewards, gamma=0.95):
    discounted_rewards = np.zeros_like(rewards)
    R = 0
    for t in reversed(range(0, len(rewards))):
        # update the total discounted reward
        R = R * gamma + rewards[t]
        discounted_rewards[t] = R

    return normalize(discounted_rewards)
```

Тепер ми можемо почати визначати алгоритм навчання, що буде використовуватися для підкріплення хороших дій агента та покарання за погані дії. У цій роботі ми сфокусуємося на методах стратегій градієнту, метою яких є максимізація вірогідності дій, що призводять до великої винагороди. Аналогічно, це означає що потрібно мінімізувати негативну вірогідність цих же дій. Ми досягаємо цього множенням вірогідностей на їх відповідну винагороду – таким чином підсилюючи силу дій з великою винагородою.

Оскільки функція логарифму монотонно зростає, це означає що мінімізація негативної вірогідності еквівалентна мінімізації негативного логарифму вірогідності. Згадаймо, що ми можемо легко підрахувати негативний логарифм вірогідності дискретних дій, обчисливши його softmax cross entropy. Як і у навчанні з учителем, можна використати стохастичний градієнтний спуск для отримання бажаної мінімізації.

```
### Loss function ###

```

```
# Arguments:
#   logits: network's predictions for actions to take
#   actions: the actions the agent took in an episode
#   rewards: the rewards the agent received in an episode
# Returns:
#   loss
def compute_loss(logits, actions, rewards):
    '''Complete the function call to compute the negative log probabilities'''
    neg_logprob = tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits=logits, labels=actions) # TODO

    '''Scale the negative log probability by the rewards'''
    loss = tf.reduce_mean( neg_logprob * rewards )
    return loss
```

Тепер використаємо функцію втрат для визначення кроку навчання агента:

```
### Training step (forward and backpropagation) ###

def train_step(model, loss_function, optimizer, observations, actions, discounted_rewards, custom_fwd_fn=No
    with tf.GradientTape() as tape:
        # Forward propagate through the agent network
        if custom_fwd_fn is not None:
            prediction = custom_fwd_fn(observations)
        else:
            prediction = model(observations)

        '''call the compute_loss function to compute the loss'''
        loss = loss_function(prediction, actions, discounted_rewards)

    '''run backpropagation to minimize the loss using the tape.gradient method.
            Unlike supervised learning, RL is *extremely* noisy, so you will benefit
            from additionally clipping your gradients to avoid falling into
            dangerous local minima. After computing your gradients try also clipping
            by a global normalizer. Try different clipping values, usually clipping
            between 0.5 and 5 provides reasonable results. '''
    grads = tape.gradient(loss, model.trainable_variables)
    grads, _ = tf.clip_by_global_norm(grads, 2)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
## Cartpole training! ##
## Note: stoping and restarting this cell will pick up training where you
#        left off. To restart training you need to rerun the cell above as
#        well (to re-initialize the model and optimizer)

if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
for i_episode in range(500):

    plotter.plot(smoothed_reward.get())
    # Restart the environment
    observation = env.reset()
    memory.clear()

```

```
    while True:
        # using our observation, choose an action and take it in the environment
        action = choose_action(cartpole_model, observation)
        next_observation, reward, done, info = env.step(action)
        # add to memory
        memory.add_to_memory(observation, action, reward)

        # is the episode over? did you crash or do so well that you're done?
        if done:
            # determine total reward and keep a record of this
            total_reward = sum(memory.rewards)
            smoothed_reward.append(total_reward)

            # initiate training - remember we don't know anything about how the
            #   agent is doing until it has crashed!
            g = train_step(cartpole_model, compute_loss, optimizer,
                       observations=np.vstack(memory.observations),
                       actions=np.array(memory.actions),
                       discounted_rewards = discount_rewards(memory.rewards))

            # reset the memory
            memory.clear()
            break
        # update our observatons
        observation = next_observation
```

Оскільки у агента немає попередніх знань про середовище, він почне навчатися балансувати стовп на візку, базуючись лише на зворотньому зв'язку, отриманому від середовища! Ми побачимо як агент поступово навчається дотримуватись стратегії для оптимізації балансування стовпа так довго, як тільки це можливо. Для цього ми будемо спостерігати, як змінюються загальні винагороди агента в залежності від епізодів. У ході навчання загальні винагороди мають зростати. Допоміжні функції, які будуть використані подано нижче:

```
# Helper functions
import matplotlib.pyplot as plt
import tensorflow as tf
import time
import numpy as np

from IPython import display as ipythondisplay
from string import Formatter




def display_model(model):
  tf.keras.utils.plot_model(model,
              to_file='tmp.png',
              show_shapes=True)
  return ipythondisplay.Image('tmp.png')


```

```
def plot_sample(x,y,vae):
    plt.figure(figsize=(2,1))
    plt.subplot(1, 2, 1)

    idx = np.where(y==1)[0][0]
    plt.imshow(x[idx])
    plt.grid(False)

    plt.subplot(1, 2, 2)
    _, _, _, recon = vae(x)
    recon = np.clip(recon, 0, 1)
    plt.imshow(recon[idx])
    plt.grid(False)

    plt.show()



class LossHistory:
  def __init__(self, smoothing_factor=0.0):
    self.alpha = smoothing_factor
    self.loss = []
  def append(self, value):
    self.loss.append( self.alpha*self.loss[-1] + (1-self.alpha)*value if len(self.loss)>0 else value )
  def get(self):
    return self.loss

class PeriodicPlotter:
  def __init__(self, sec, xlabel='', ylabel='', scale=None):

    self.xlabel = xlabel
    self.ylabel = ylabel
    self.sec = sec
    self.scale = scale

    self.tic = time.time()

  def plot(self, data):
    if time.time() - self.tic > self.sec:
      plt.cla()

      if self.scale is None:
        plt.plot(data)
      elif self.scale == 'semilogx':
        plt.semilogx(data)
      elif self.scale == 'semilogy':
        plt.semilogy(data)
      elif self.scale == 'loglog':
        plt.loglog(data)
      else:
        raise ValueError("unrecognized parameter scale {}".format(self.scale))
```

```
72
73        plt.xlabel(self.xlabel); plt.ylabel(self.ylabel)
74        ipythondisplay.clear_output(wait=True)
75        ipythondisplay.display(plt.gcf())
76
77        self.tic = time.time()
```

Задання оптимізатора, швидкості навчання агента тощо:

```
## Training parameters ##
## Re-run this cell to restart training from scratch ##

# Learning rate and optimizer
learning_rate = 1e-3
optimizer = tf.keras.optimizers.Adam(learning_rate)

# instantiate cartpole agent
cartpole_model = create_cartpole_model()

# to track our progress
smoothed_reward = LossHistory(smoothing_factor=0.95)
plotter = PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Rewards')
```

Навчання агента:

```
### Training step (forward and backpropagation) ###

def train_step(model, loss_function, optimizer, observations, actions, discounted_rewards, custom_fwd_fn=No
    with tf.GradientTape() as tape:
        # Forward propagate through the agent network
        if custom_fwd_fn is not None:
            prediction = custom_fwd_fn(observations)
        else:
            prediction = model(observations)

        '''call the compute_loss function to compute the loss'''
        loss = loss_function(prediction, actions, discounted_rewards)

    '''run backpropagation to minimize the loss using the tape.gradient method.
            Unlike supervised learning, RL is *extremely* noisy, so you will benefit
            from additionally clipping your gradients to avoid falling into
            dangerous local minima. After computing your gradients try also clipping
            by a global normalizer. Try different clipping values, usually clipping
            between 0.5 and 5 provides reasonable results. '''
    grads = tape.gradient(loss, model.trainable_variables)
    grads, _ = tf.clip_by_global_norm(grads, 2)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
## Cartpole training! ##
## Note: stoping and restarting this cell will pick up training where you
#        left off. To restart training you need to rerun the cell above as
#        well (to re-initialize the model and optimizer)

if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
for i_episode in range(500):
```

```
30
31      plotter.plot(smoothed_reward.get())
32      # Restart the environment
33      observation = env.reset()
34      memory.clear()
35
36      while True:
37          # using our observation, choose an action and take it in the environment
38          action = choose_action(cartpole_model, observation)
39          next_observation, reward, done, info = env.step(action)
40          # add to memory
41          memory.add_to_memory(observation, action, reward)
42
43          # is the episode over? did you crash or do so well that you're done?
44          if done:
45              # determine total reward and keep a record of this
46              total_reward = sum(memory.rewards)
47              smoothed_reward.append(total_reward)
48
49              # initiate training - remember we don't know anything about how the
50              #   agent is doing until it has crashed!
51              g = train_step(cartpole_model, compute_loss, optimizer,
52                         observations=np.vstack(memory.observations),
53                         actions=np.array(memory.actions),
54                         discounted_rewards = discount_rewards(memory.rewards))
55
56              # reset the memory
57              memory.clear()
58              break
59          # update our observatons
60          observation = next_observation
```

Деталі щодо середовищ можна переглянути за наступними посиланнями з офіційного ресурсу:

1. Pong: https://www.gymlibrary.dev/environments/atari/pong/?highlight=pongpong
2. Cart Pole: https://www.gymlibrary.dev/environments/classic_control/cart_pole/
3. Mountain Car: https://www.gymlibrary.dev/environments/classic_control/mountain_car/

## Завдання для виконання

Завдання 1. Дайте короткі власні відповіді на наступні питання:

1. Як показав себе агент на практиці для кожного середовища?
2. Чи можна витратити менше часу на тренування та все ще отримати гарний результат?
3. Чи вважаєте ви, що збільшуючи час навчання агента ми могли б покращити результат ще більше?
4. Як складність між Pong та Cartpole впливає на швидкість тренування та загальний результат?
5. Що можна змінити в агенті або процесі навчання для того аби покращити результат?

Завдання 2. Створити за аналогією попереднього прикладу середовище `MountainCar-v0` та навчити агента досягати поставленої мети у цьому середовищі.

## Оцінювання

Ваша оцінка за виконання практичної буде залежати від:

- 30% – подано короткі власні відповіді на питання
- 50% – створено середовище `MountainCar-v0` та навчено агента досягати поставлену мету у цьому середовищі
- 20% – звіт: описано процес створення середовища `MountainCar-v0` та процес навчання агента у цьому середовищі

Шаблон за яким потрібно підготувати звіт поданий нижче на наступних сторінках:

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

# Навчання з підкріпленням

## Практична робота # 1

## Перевернутий маятник (Cartpole), понг (Pong) та MountainCar

**Виконав(ла)**

Ім’я Прізвище
Група: IO-xxx
Курс: X

16 вересня 2021 р.

# 1 MountainCar

## 1.1 Визначення середовища MountainCar та агента

Зазначте інструментарій, який був використаний для моделювання середовища. Вкажіть як здійснюється доступ до середовища. Опишіть поставлену задачу, яку має виконати агент у середовищі MountainCar. Вкажіть, які спостереження може використовувати агент для визначення свого стану та які дії може виконувати агент у цьому середовищі. Виведіть розмір простору спостережень та розмір простору дій агента на екран. Визначте агента та поясніть як агент буде обирати, яку дію йому слід виконати для досягнення поставленої перед ним мети.

## 1.2 Визначення пам'яті агента

Опишіть процедуру та роль пам'яті агента, тобто для чого вона потрібна та як Ви її задавали. Якщо Ви хочете супроводжувати свій опис блоками коду – можете це зробити через зовнішній файл, наприклад так:

**Лістинг 1:** Перевірка на парність введеного числа.

```python
# Python program to check if the input number is odd or even.
# A number is even if division by 2 gives a remainder of 0.
# If the remainder is 1, it is an odd number.

num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

Або вставивши безпосередньо потрібні рядки коду в оточення `lstlisting`:

**Лістинг 2:** Перетин двох масивів.

```
def intersection(array_1, array_2):
  element_1 = set()
  intersected = []
  for i in range(0, len(array_1)):
    element_1.add(array_1[i])
  already_added = set()
  for j in range(0, len(array_2)):
    if array_2[j] in element_1 and array_2[j] not in already_added:
      intersected.append(array_2[j])
      # MISSING LINE HERE.
      already_added.add(array_2[j])
  return intersected
```

Без рамки та без підпису:

```
def intersection(array_1, array_2):
  element_1 = set()
  intersected = []
  for i in range(0, len(array_1)):
    element_1.add(array_1[i])
  already_added = set()
  for j in range(0, len(array_2)):
    if array_2[j] in element_1 and array_2[j] not in already_added:
      intersected.append(array_2[j])
      # MISSING LINE HERE.
      already_added.add(array_2[j])
  return intersected
```

## 1.3 Функція винагороди

Тут Ви повинні надати детальну інформацію про те, яку функцію винагороди використовували та як її задавали. Подайте формулу, програмну реалізацію, графік тощо.

Приклад формули. Сигмоїда задається наступним виразом:

$$P(y = 1|z) = \sigma(z) = \frac{1}{1 + e^{-z}}, \tag{1}$$

де $y$ – очікуване значення моделі.

## 1.4 Функція втрат

Тут Ви повинні надати детальну інформацію про те, яку функцію втрат використовували та як її задавали. Подайте формулу, програмну реалізацію, графік тощо.

## 1.5 Крок навчання агента

Подайте програмну реалізацію кроку навчання агента.

## 1.6 Навчання агента

Для представлення результатів навчання агента використовуйте графік залежності винагороди від кількості ітерацій агента. Деякі результати для кращого порівняння можете подати у вигляді таблиці. Приклад таблиці 1. Приклад рисунку 1. Остаточний результат навчання агента оцініть візуально по відео, так як показано у прикладах: перевернутий маятник та понг.

**Табл. 1:** Приклад таблиці.

| Ім'я оператора | | Синтаксис |
|---|---|---|
| Присвоєння | | a = b |
| Додавання | | a + b |
| Віднімання | | a - b |
| Унарний плюс | | +a |
| Унарний мінус | | -a |
| Множення | | a * b |
| Ділення | | a / b |
| Залишок від ділення | | a % b |
| Інкремент | префікс<br>суфікс | ++a<br>a++ |
| Декремент | префікс<br>суфікс | - -a<br>a- - |

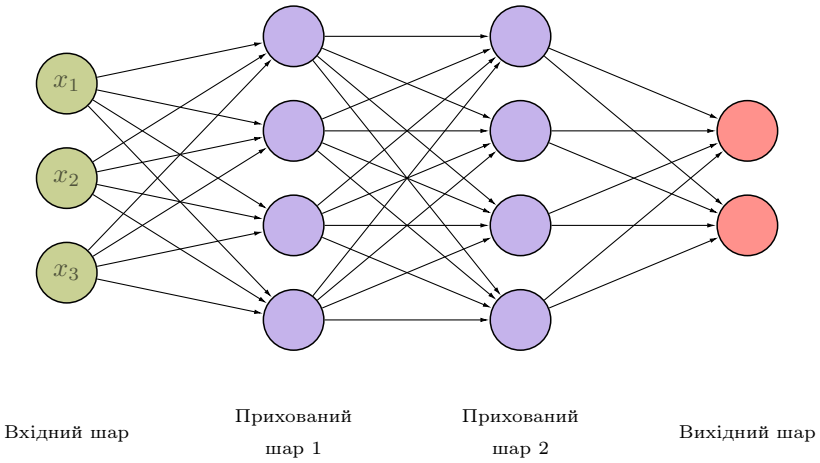



**Рис. 1:** Приклад представлення нейронної мережі.

## 1.7 Допомога

У цьому розділі потрібно залишити додаткові деталі про виконання цієї роботи. Тобто, якщо Ви обговорювали завдання або працювали над ним у групі – потрібно це зазначати: з ким працювали та що кожен учасник виконував. Якщо отримували допомогу – вказати від кого та яку допомогу було отримано. Якщо використовували додаткові матеріали (блокноти Kaggle, github тощо) – залишите посилання на джерела. Якщо завдання було виконано особисто Вами і Ви не отримували ніякої допомоги – так і напишіть. Використання будь-яких чужих матеріалів та представлення їх за свої напрацювання є плагіатом та серйозним порушенням основних академічних стандартів. Приклад посилання на два джерела одночасно [2, 3].

## 1.8 Висновки

Резюмуйте пророблену Вами роботу, відобразіть свої спостереження щодо налаштування та складності середовища, процесу навчання агента та отриманих результатів. Чи можна витратити менше часу на навчання агента та все ще показати гарний результат? Які речі можна змінити в агенті або в процесі навчання для того аби покращити результат?

## Література

[1] Chest X-ray. Kaggle. [Online]. Available: https://www.kaggle.com/c/cxray/data

[2] M. Mathur. Cnn using keras(100% accuracy). Kaggle. [Online]. Available: https://www.kaggle.com/madz2000/cnn-using-keras-100-accuracy

[3] R. Jain. Deep learning using sign langugage. Kaggle. [Online]. Available: https://www.kaggle.com/ranjeetjain3/deep-learning-using-sign-langugage

# Практична 2: $N$-рукий бандит

*"Efficiency is doing things right; effectiveness is doing the right things."*
*"Продуктивність – робити речі правильно; ефективність – робити правильні речі."*

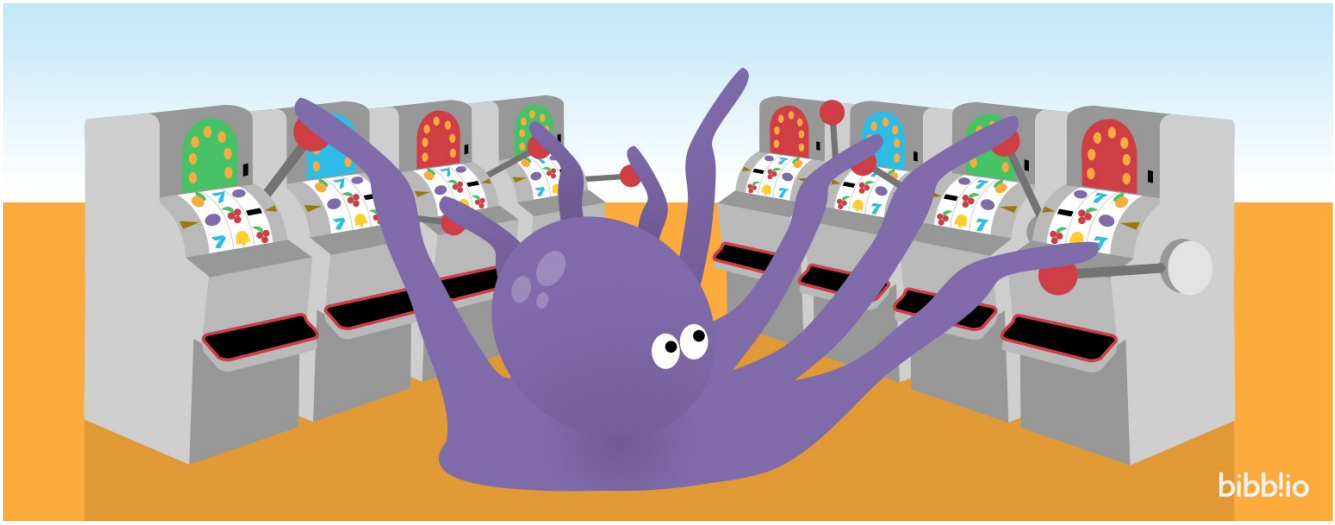
– Пітер Друкер

## Вступ

Виконуючи це завдання, Ви познайомитесь з тим як агент знаходить компроміс між вивчення поточної ситуації у середовищі та використанням раніше набутих знань.

## Опис завдання

Давайте розглянемо наступну ситуацію. Скажімо, Ви опинились у казино, де перед вами знаходиться 8 ігрових автоматів з кричущим надписом: "Грай безкоштовно! Максимальна виплата становить 10 доларів!" Вау, непогано! Ви заінтриговано запитуєте одну зі співробітниць, що відбувається, тому що це здається занадто дивним, щоб бути правдою, і вона каже: "Це справді правда, грайте безкоштовно скільки завгодно. Кожен ігровий автомат гарантовано дасть Вам винагороду від 0 до 10 доларів. Але слід пам'ятати, що кожен із цих 8 ігрових автоматів має різну середню виплату, тому спробуйте з'ясувати, який із них дає у середньому найбільшу винагороду і ви отримаєте багато грошей!"




Джерело зображення: Multi-Armed Bandits are the New A/B Tests

Що це за казино? Давайте просто з'ясуємо, як можна отримати якнайбільше грошей! До речі, однооруким бандитом називають ігровий автомат. У нього є одна рука (важіль), і він зазвичай краде ваші гроші. Ця ситуація відповідає проблемі 8-рукого бандита, але у загальному випадку можна розглядати задачу $N$-рукого

бандита, де $N$ – кількість ігрових автоматів. Задача багаторукого бандита є прикладом класу задач, які демонструють дилему компромісу між розвідкою[3] (`exploration`) та використанням[4] (`exploitation`) [1].

Сформулюємо нашу проблему більш формально. У нас є $N$ можливих дій (для цього випадку $N = 8$), де дія означає: потягнути за ручку або важіль певного ігрового автомата. Під час кожної гри ($k$) ми можемо вибрати лише один важіль, щоб це зробити. Після виконання дії $a$ ми отримаємо винагороду $R_k$ – винагорода отримана за гру $k$. Кожен важіль має унікальний розподіл ймовірностей для виплат грошей (винагород). Наприклад, якщо ми маємо 8 ігрових автоматів і граємо багато разів, то ігровий автомат № 3 може дати середню винагороду скажімо 9 доларів, тоді як ігровий автомат № 1 може дати середню винагороду лише 4 долари. Звичайно, оскільки винагорода за кожну гру є імовірнісною, можливо, що автомат № 1 випадково дасть нам винагороду в розмірі 10 доларів у якійсь грі. Але якщо ми зіграємо багато ігор, ми очікуємо, що у середньому ігровий автомат № 1 буде видавати меншу винагороду, ніж № 3.

Наша стратегія повинна полягати у тому, щоб зіграти кілька разів, вибираючи різні важелі та спостерігати за отриманою винагородою за кожну дію. У наступній грі ми хочемо обрати лише той важіль, який мав найбільшу середню винагороду. Таким чином, нам потрібно визначити середню винагороду за виконання дії $a$ у грі $k$, беручи до уваги винагороди, які були отримані у попередніх іграх для цієї дії. Позначимо середню винагороду як $Q_k(a)$:

$$Q_k(a) = \frac{R_1 + R_2 + \cdots + R_k}{k_a}$$

Псевдокод:

```
def exp_reward(action, history):
  rewards_for_action = history[action]
  return sum(rewards_for_action) / len(rewards_for_action)
```

Тобто, середня винагорода у грі $k$ за дію $a$ є середнім арифметичним усіх попередніх винагород, які ми отримали, виконуючи дію $a$. Таким чином, наші попередні дії та спостереження упливають на наші майбутні дії. Функцію $Q_k(a)$ називають функцією цінності (цінність дій), оскільки вона показує нам очікуване значення винагороди для заданої дії. Оскільки ми зазвичай позначаємо цю функцію символом $Q$, тому її також часто називають $Q$-функцією.

# Розвідка та використання

Коли ми вперше починаємо грати, нам потрібно грати в гру і спостерігати за винагородами, які ми отримуємо для різних автоматів. Ми можемо назвати цю стратегію **розвідкою**, оскільки ми, по суті, випадковим чином вивчаємо результати наших дій. Іншу стратегію, яку ми могли б застосувати – **використання**, яка означає, що ми використовуємо наші поточні знання про те, який автомат, здається, приносить найбільшу винагороду і продовжуємо грати на ньому. Наша загальна стратегія має включати деяку кількість **використань** (вибір найкращого важеля на основі того, що ми знаємо на даний момент) і певну кількість **розвідок** (вибір випадкових важелів, щоб ми могли дізнатися більше). Правильний баланс між використаннями та розвідками буде важливим для максимізації наших винагород.

Як ми можемо розробити алгоритм, який би визначав, який ігровий автомат має найбільшу середню винагороду? Ну, найпростішим алгоритмом було б просто вибрати дію, пов'язану з найвищим значенням $Q$:

$$a^* = \arg\max_a Q_k(a_i) \qquad \forall a_i \in A_k$$

Псевдокод:

[3] Вивчення поточної ситуації у середовищі.
[4] Використання раніше набутих знань.

```
def get_best_action(actions, history):
  exp_rewards = [exp_reward(action, history) for action in actions]
  return argmax(exp_rewards)
```

Ми використовуємо нашу вищезгадану функцію $Q_k(a)$ для усіх можливих дій і вибираємо ту дію, яка повертає максимальну середню винагороду. Оскільки $Q_k(a)$ залежить від запису наших попередніх дій та пов'язаних з ними винагород, цей метод не буде оцінювати ті дії, які ми ще не виконували. Таким чином, якщо ми, скажімо, спочатку спробували важелі № 1 та № 3 і помітили, що важіль № 3 дає нам більшу винагороду, тоді дотримуючись лише цього методу ми ніколи більше не подумаємо спробувати інший важіль, наприклад, № 6, який, насправді міг би давати найвищу середню винагороду. Цей метод, який дозволяє просто вибрати найкращий важіль, називається жадібним (`greedy`) або методом використання.

## $\varepsilon$-жадібна стратегія

Відкрити приклад у Colab, який буде розглядатись далі, можна за цим посиланням:

https://github.com/YKochura/rl-kpi/blob/main/homeworks/lab2/N_armed_Bandits.ipynb

Нам потрібно перевірити інші важелі (інші ігрові автомати), щоб знайти дійсно найкращу дію. Для того, щоб це зробити, потрібно просто модифікувати наш попередній алгоритм таким чином, щоб з ймовірністю $\varepsilon$ ми обирали дію $a$ випадковим чином, а решту часу з ймовірністю $1-\varepsilon$ ми обирали найкращий важіль на основі попереднього досвіду. Цей метод відомий як $\varepsilon$ (епсилон)-жадібна стратегія. У більшості випадків ми будемо грати жадібно ($\varepsilon < 0.5$), але іноді ми ризикуємо і вибираємо випадковий важіль, щоб побачити, що станеться. Результат, звичайно, вплине на наші майбутні жадібні дії.

```
import numpy as np
from scipy import stats
import random
import matplotlib.pyplot as plt

N = 8  # Кількість рук бандита (число ігрових автоматів)
probs = np.random.rand(N) # Приховані ймовірності, пов'язані з кожною рукою
eps = 0.1 # епсилон для епсилон-жадібного вибору дій
```

```
probs
```

```
array([0.84026687, 0.7182308 , 0.52330024, 0.52118617, 0.77030153,
       0.75181956, 0.13122341, 0.18899409])
```

У цьому прикладі ми будемо вирішувати проблему з 8-руким бандитом, тому $N = 8$. Числовий масив `probs` довжиною $N$, заповнений випадковими числами з плаваючою комою, які можна розглядати як ймовірності. Кожна позиція у масиві `probs` відповідає номеру важеля, тобто є можливою дією. Наприклад, перший елемент у масиві `probs` має позицію індексу 0, тому це дія 0, що відповідає важелю під номером 0. Кожен важіль має свою ймовірність, яка враховує сумарну винагороду, яку можна отримати потягнувши за важіль.

Спосіб, який ми обрали для реалізації розподілу ймовірності винагороди для кожного важелю, такий: кожний важіль матиме ймовірність, наприклад, 0.6, а максимальна винагорода становить 10 доларів. Ми налаштуємо цикл `for`, який буде пробігати до 8, і на кожному кроці він додаватиме 1.25 до винагороди, якщо випадкове число з плаваючою комою буде менше, ніж ймовірність важеля. Таким чином, у першому проході циклом створюється випадковий float (наприклад, 0.4), 0.4 менше 0.6, тому `reward += 1.25`. На наступній ітерації створюється ще один випадковий float (наприклад, 0.55), що також менше за 0.6, тому `reward += 1.25`. Це продовжується до тих пір, поки ми не завершимо 8 ітерацій, а потім повернемо остаточну загальну винагороду, яка може бути у діапазоні від 0 до 10. З ймовірністю важеля 0.6, середня винагорода за виконання цієї дії до нескінченності буде давати 6, але під час будь-якої окремої гри винагорода може бути більшою чи меншою.

```
def get_reward(prob, N=8):
    reward = 0;
```

```
    for i in range(N):
        if random.random() < prob:
            reward += 1.25
    return reward
```

```
reward_test = [get_reward(0.6) for _ in range(2000)]
```

```
np.mean(reward_test)
```

```
5.9675
```

Отриманий результат показує, що виконання цього коду 2000 разів з імовірністю 0.6 дійсно дає нам середню винагороду близько 6 (див. гістограму на рисунку 2).

```
plt.figure(figsize=(9,5), dpi=600)
plt.xlabel("Винагорода",fontsize=22)
plt.ylabel("# Спостереження",fontsize=22)
plt.hist(reward_test,bins=7)
```

```
(array([ 13.,  98., 263., 466., 529., 406., 225.]),
 array([ 0.        ,  1.42857143,  2.85714286,  4.28571429,  5.71428571,
         7.14285714,  8.57142857, 10.        ]),
 <BarContainer object of 7 artists>)
<Figure size 5400x3000 with 1 Axes>
```

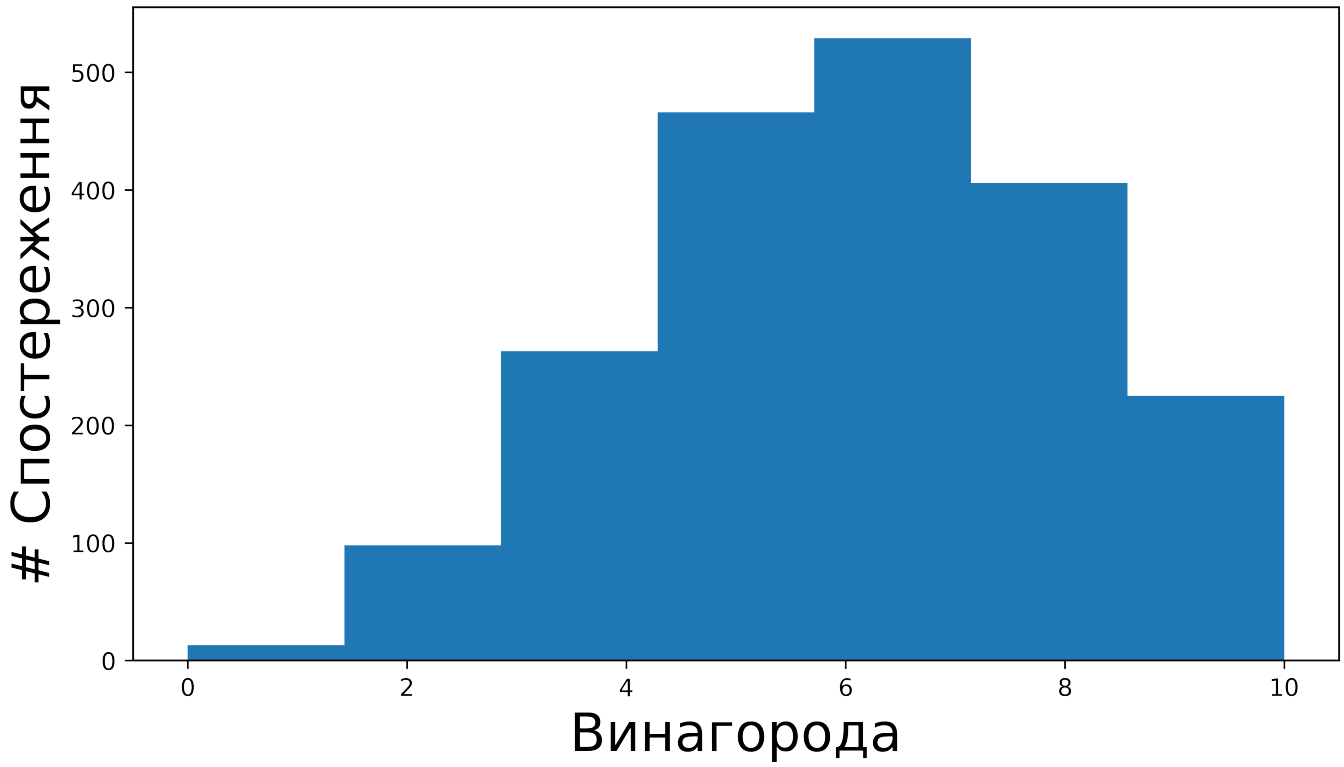



**Рис. 2:** Розподіл винагород для змодельованого $N$-рукого бандита з ймовірністю виплати 0.6

Наступна функція, яку ми визначимо – це наша жадібна стратегія вибору найкращого важелю. Нам потрібен спосіб для відстежування важелів, які були потягнуті і яка в результаті була винагорода. Ми могли б просто створити список і додати у цей список кортеж у вигляді спостережень (важіль, винагорода), наприклад, (2, 9), вказуючи, що ми вибрали важіль 2 і отримали винагороду 9. Цей список зростав би у продовж усієї гри.

Однак існує набагато простіший підхід, оскільки нам насправді потрібно відстежувати лише середню винагороду для кожного важелю (руки) – нам не потрібно зберігати кожне спостереження. Нагадаємо, що для обчислення середнього значення для списку чисел $x$ потрібно просто підсумувати усі значення цього списку $x_i$, а потім розділити отриману суму на кількість елементів у цьому списку:

$$\mu = \frac{1}{k}\sum_i x_i$$

В основному це математичний еквівалент циклу `for`, наприклад:

```
sum = 0
x = [4, 5, 6, 7]
for i in range(len(x)):
    sum = sum + x[i]

mu = sum / len(x)
mu
```

```
5.5
```

Якщо ми вже маємо середню винагороду $\mu$ для певного важеля, тоді ми можемо оновити це значення, коли отримаємо нову винагороду в іншій грі, перерахувавши середнє значення. Нам спочатку потрібно скасувати попереднє середнє значення, а потім перерахувати його. Щоб скасувати попереднє середнє значення, ми помножимо $\mu$ на загальну кількість до цього зіграних ігор $k$. Звичайно, це дасть нам лише суму, а не вихідний набір значень – ми не можете скасувати суму. Загальне число зіграних ігор – це те, що нам потрібно знати, щоб перерахувати середнє значення з урахуванням отриманого нового значення винагороди. Ми просто додаємо цю суму до нового значення винагороди $x_{k+1}$ і ділимо на $k+1$ – нова загальна кількість зіграних ігор:

$$\mu_{new} = \frac{k \cdot \mu_{old} + x_{k+1}}{k+1} \tag{5}$$

Ми можемо використовувати це рівняння, щоб постійно оновлювати середню винагороду для кожного важеля, коли ми збираємо нові дані. Таким чином нам потрібно відстежувати лише два числа для кожного важеля: $k$ – кількість отриманих винагород (зіграних ігор) та $\mu$ – поточне середнє значення винагороди.

Ми можемо легко зберегти це у масиві `numpy` $8 \times 2$ (припускаючи, що у бандита 8 рук):

```
# 8 дій x 2 стовпці
# Стовпці: Кількість зіграних ігор, Середня винагорода
record = np.zeros((N, 2))
record
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

У першому стовпці цього масиву буде зберігатися кількість перетягувань кожної руки автомата, а в другому стовпці зберігатиметься поточна середня винагорода. Давайте напишемо функцію для оновлення цього масиву, коли було виконано нову дію та отримано нову винагороду:

```python
def update_record(record, action, r):
    new_r = (record[action, 0] * record[action, 1] + r) / (record[action, 0] + 1)
    record[action, 0] += 1
    record[action, 1] = new_r
    return record
```

Ця функція приймає масив `record`, дію (`action`), яка є просто значенням індексу важеля і нове значення винагороди `r`. Щоб оновити середню винагороду ця функція просто реалізує математичну функцію (5), яку ми описали раніше, а потім збільшує лічильник на одиницю, який зберігає скільки разів цей важіль було використано.

Далі нам потрібно реалізувати функцію, яка вибере, який важіль слід потягнути. Ми хочемо, щоб ця функція обирала важіль, який асоціюється з найвищою середньою винагородою, тому все, що нам потрібно зробити – це знайти рядок у масиві `record` з найбільшим значенням у стовпці 1. Ми можемо це легко зробити, використавши вбудовану функцію `numpy argmax`, яка приймає масив, знаходить найбільше значення в цьому масиві та повертає його позицію у масиві (індекс):

```python
def get_best_arm(record):
    arm_index = np.argmax(record[:,1], axis=0)
    return arm_index
```

Тепер ми можемо перейти до основного циклу для гри з $N$-руким бандитом. Якщо випадкове число буде більше за параметр епсилон (`eps`), ми просто знаходимо найкращу дію за допомогою функції `get_best_arm` і виконуємо її. В іншому випадку ми обираємо випадкову дію, щоб забезпечити певний обсяг розвідки цього середовища. Після вибору важеля ми викликаємо функцію `get_reward`, яка поверне значенням винагороди для цього важеля у поточній грі (нове спостереження). Потім ми оновлюємо масив `record` цим новим спостереження. Ми повторюємо цей процес певну кількість разів, тим самим постійно оновлюючи масив `record`. Важіль з найбільшою ймовірністю винагороди в кінцевому підсумку буде обиратись найчастіше, оскільки цей автомат дасть найвищу середню винагороду (див. нижче масив `record` після 500 зіграних ігор).

Давайте зіграємо 500 разів. Сподіваємось, ми побачимо, що середня винагорода збільшується зі збільшенням кількості зіграних ігор.

```python
fig, ax = plt.subplots(figsize=(1,1), dpi=600)
ax.set_xlabel("Кількість зіграних ігор")
ax.set_ylabel("Середня винагорода")
fig.set_size_inches(9, 5)
rewards = [0]
for i in range(500):
    if random.random() > eps:
        choice = get_best_arm(record)
    else:
        choice = np.random.randint(N)
    r = get_reward(probs[choice])
    record = update_record(record, choice, r)
    mean_reward = ((i + 1) * rewards[-1] + r) / (i + 2)
    rewards.append(mean_reward)
ax.scatter(np.arange(len(rewards)), rewards, s=10)
```

```python
record
```

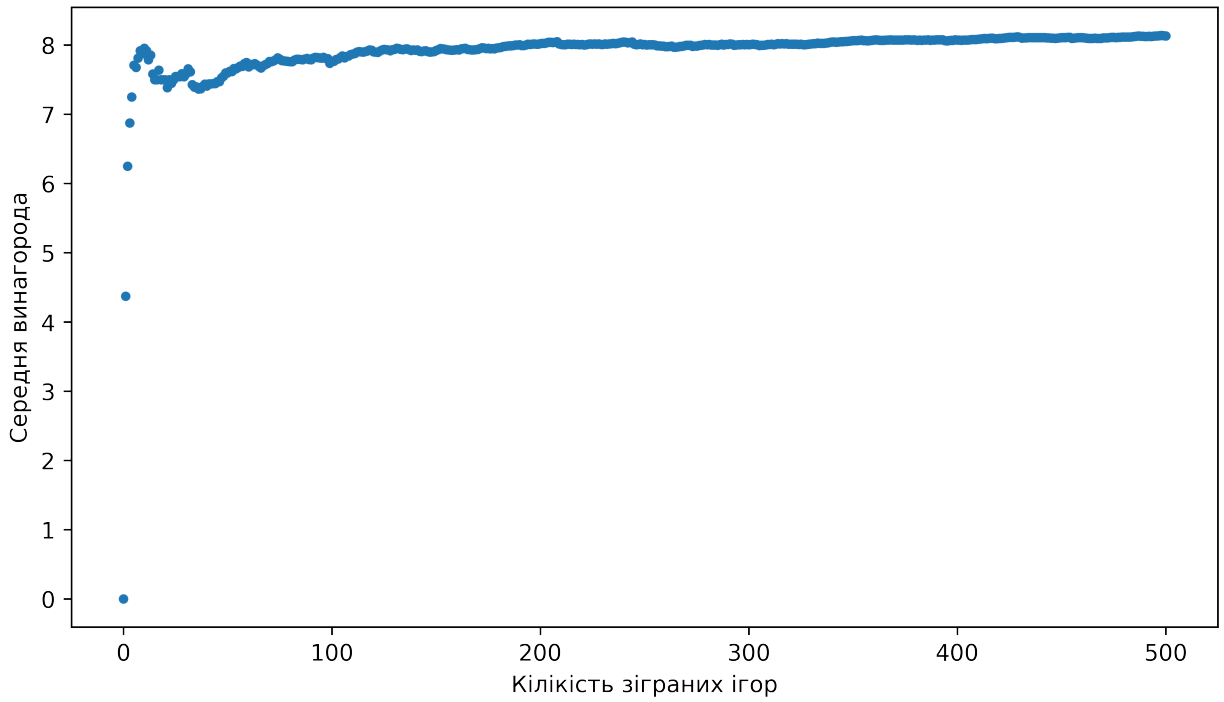



**Рис. 3:** Цей графік показує, що середня винагорода за кожну гру з часом збільшується. Це вказує на те, що ми успішно вчимося вирішувати проблему $N$-рукого бандита

```
array([[445.        ,   8.33426966],
       [  4.        ,   6.25      ],
       [  6.        ,   5.41666667],
       [  9.        ,   5.69444444],
       [ 22.        ,   8.23863636],
       [  9.        ,   7.77777778],
       [  1.        ,   1.25      ],
       [  4.        ,   1.25      ]])
```

Як ви можете бачити з рисунку 3, середня винагорода дійсно зростає зі збільшенням кількості зіграних ігор, потім виходить на насичення (знаходження автомату з найбільшою середньою винагородою). Наш алгоритм навчається, УРА! Процес навчання підкріплюється попередніми добре зіграними іграми! Цей алгоритм, як Ви могли помітити, є досить простим.

## Завдання для виконання

Давайте трохи ускладнимо попереднє завдання. Задача, яку ми розглянули вище, є стаціонарною, оскільки основні розподіли ймовірностей винагород для кожного автомату не змінювались з часом. Розгляньте випадок, коли розподіли ймовірностей винагород змінюються – нестаціонарна задача. У цьому випадку Вам потрібно модифікувати `update_record` для знаходження середньозваженого значення винагороди. Крім того, дослідіть для обох випадків: стаціонарної та нестаціонарної задачі про $N$-рукого бандита, вплив значення $\varepsilon$ на швидкість знаходження найкращого автомата, тобто автомата з найбільшою середньою або середньозваженою винагородою. Результати оформити та представити у вигляді звіту.

## Оцінювання

Ваша оцінка за виконання завдання буде залежати від:

- 20% – досліджено вплив параметра $\varepsilon$ на швидкість знаходження найкращого автомата для стаціонарної та нестаціонарної задачі про $N$-рукого бандита.
- 40% – реалізовано програмно нестаціонарну задачу про $N$-рукого бандита.
- 40% – підготовлено звіт у якому подана програмна реалізація нестаціонарної задачі про про $N$-рукого бандита, а також представлено дослідження впливу параметра $\varepsilon$ на швидкість знаходження найкращого автомата для стаціонарної та нестаціонарної задачі. Очікується формальний звіт, написаний в LaTeX.

  Шаблон за яким потрібно підготувати звіт поданий нижче на наступних сторінках:

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

# Навчання з підкріпленням

## Практична робота # 2

---

### N-рукий бандит

**Виконав(ла)**

Ім’я Прізвище
Група: IO-xxx
Курс: X

1 вересня 2022 р.

# 1 N-рукий бандит

## 1.1 Нестаціонарна задача

Подайте детально прокоментовану програмну реалізацію нестаціонарної задачі про $N$-рукого бандита. Представте використані формули / формулу. **Наприклад**. Сигмоїда задається наступним виразом:

$$P(y = 1|z) = \sigma(z) = \frac{1}{1 + e^{-z}}, \tag{1}$$

де $y$ – очікуване значення моделі.

Супроводжуйте свій опис блоками коду – можете це зробити через зовнішній файл, наприклад так:

**Лістинг 1:** Перевірка на парність введеного числа.

```python
# Python program to check if the input number is odd or even.
# A number is even if division by 2 gives a remainder of 0.
# If the remainder is 1, it is an odd number.

num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

Або вставивши безпосередньо потрібні рядки коду в оточення `lstlisting`:

**Лістинг 2:** Перетин двох масивів.

```
def intersection(array_1, array_2):
  element_1 = set()
  intersected = []
  for i in range(0, len(array_1)):
    element_1.add(array_1[i])
  already_added = set()
  for j in range(0, len(array_2)):
    if array_2[j] in element_1 and array_2[j] not in already_added:
      intersected.append(array_2[j])
      # MISSING LINE HERE.
      already_added.add(array_2[j])
  return intersected
```

Без рамки та без підпису:

```
def intersection(array_1, array_2):
  element_1 = set()
  intersected = []
  for i in range(0, len(array_1)):
    element_1.add(array_1[i])
  already_added = set()
  for j in range(0, len(array_2)):
    if array_2[j] in element_1 and array_2[j] not in already_added:
      intersected.append(array_2[j])
      # MISSING LINE HERE.
      already_added.add(array_2[j])
  return intersected
```

## 1.2 Дослідження параметра $\varepsilon$

Тут має бути подано Ваше дослідження впливу параметра $\varepsilon$ на швидкість знаходження найкращого автомата для стаціонарної та нестаціонарної задачі про $N$-рукого бандита. Подайте програмну реалізацію цього дослідження, графіки отриманих результатів тощо. Деякі результати для кращого порівняння можете подати у вигляді таблиці. Приклад таблиці 1. Приклад рисунку 1.

**Табл. 1:** Приклад таблиці.

| Ім'я оператора | | Синтаксис |
|---|---|---|
| Присвоєння | | a = b |
| Додавання | | a + b |
| Віднімання | | a - b |
| Унарний плюс | | +a |
| Унарний мінус | | -a |
| Множення | | a * b |
| Ділення | | a / b |
| Залишок від ділення | | a % b |
| Інкремент | префікс | ++a |
| | суфікс | a++ |
| Декремент | префікс | - -a |
| | суфікс | a- - |

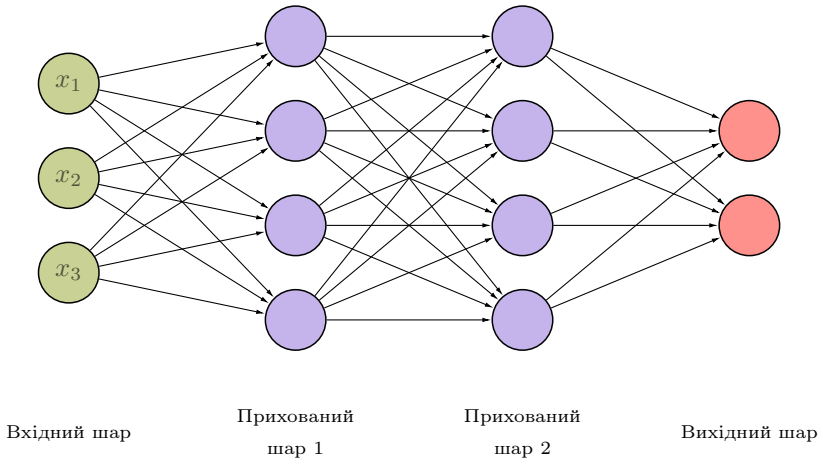



**Рис. 1:** Приклад представлення нейронної мережі.

## 1.3 Допомога

У цьому розділі потрібно залишити додаткові деталі про виконання цієї роботи. Тобто, якщо Ви обговорювали завдання або працювали над ним у групі – потрібно це зазначати: з ким працювали та що кожен учасник виконував. Якщо отримували допомогу – вказати від кого та яку допомогу було отримано. Якщо використовували додаткові матеріали (блокноти Kaggle, github тощо) – залишите посилання на джерела. Якщо завдання було виконано особисто Вами і Ви не отримували ніякої допомоги – так і напишіть. Використання будь-яких чужих матеріалів та представлення їх за свої напрацювання є плагіатом та серйозним порушенням основних академічних стандартів. Приклад посилання на два джерела одночасно [**?**, **?**].

## 1.4 Висновки

Резюмуйте пророблену Вами роботу, відобразіть свої спостереження щодо величини параметра $\varepsilon$ в $\varepsilon$-жадібній стратегії для стаціонарної та нестаціонарної задачі про $N$-рукого бандита. Чим відрізняється стаціонарна від нестаціонарної задачі про $N$-рукого бандита? Спробуйте оцінити складність цих двох алгоритмів.

## Література

[1] M. Mathur. Cnn using keras(100% accuracy). Kaggle. [Online]. Available: https://www.kaggle.com/madz2000/cnn-using-keras-100-accuracy

[2] R. Jain. Deep learning using sign langugage. Kaggle. [Online]. Available: https://www.kaggle.com/ranjeetjain3/deep-learning-using-sign-langugage

# Практична 3: Глибинне Q-навчання

*"Logic will get you from A to B. Imagination will take you everywhere."*
*"Логіка може провести Вас від пункту А до пункту Б, а уява – куди завгодно."*

– Альберт Ейнштейн

## Вступ

Виконуючи це завдання, Ви познайомитесь з тим як здійснювати передбачення найкращих дій та станів агента.

## Опис завдання

Q-навчання – це алгоритм навчання з підкріпленням на основі функції цінності, який прагне знайти найкращу дію, яку потрібно виконати з огляду на поточний стан. В Q-навчанні ми створюємо так звану Q-таблицю або матрицю, яка відповідає формі `[стан, дія]`, ініціалізувавши її нулями. Потім ми оновлюємо та зберігаємо наші значення Q після епізоду. Ця Q-таблиця стає довідковою таблицею для нашого агента, щоб вибрати найкращу дію на основі Q-значення. Наступним кроком є просто взаємодія агента з середовищем і оновлення пар стан-дія в нашій Q-таблиці `Q[стан, дія]`. Оновлення відбуваються після кожного кроку чи дії та завершуються, коли завершується епізод. Завершення епізоду настає тоді, коли досягнуто деякої кінцевої точки агентом (термінального стану). Термінальним станом, наприклад, може бути як досягнення кінця якоїсь гри, виконання бажаної мети тощо. Агент не дізнається багато після одного епізоду, але врешті-решт після достатнього навчання (кроків та епізодів) він зійдеться та дізнається оптимальні Q-значення.

Для оновлення Q-таблиці потрібно здійснити наступні кроки:

1. Агент починає з деякого стану, виконує дію, переходить в наступний стан та отримує винагороду
2. Агент обирає наступну дію, посилаючись на Q-таблицю з найвищим (максимальним) значенням або обирає випадково
3. Оновлює значення Q:

$$\boxed{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big]} \tag{6}$$

де $\alpha_t$ – швидкість навчання, $\gamma$ – коефіцієнт знецінювання.

## Завдання для виконання

Програмно реалізуйте алгоритм Q-навчання та детально прокоментуй його.

## Оцінювання

Ваша оцінка за виконання практичної буде залежати від:

- 60% – програмна реалізація алгоритму Q-навчання
- 40% – детально прокоментований скрип програмної реалізації

# Проєкт: інструкції для виконання

*"Той, хто вчиться, але не мислить – втратить себе, той, хто мислить, але не вчиться – занапастить себе."*

– Конфуцій

## Опис завдання

Над проєктною роботою можна працювати у групах (до двох людей) або самостійно. **Мета проєктної роботи** – застосовувати та розглянути передові прийоми та розробки в галузі штучного інтелекту, які дотичні до цього курсу та викликають у Вас зацікавленість.

Ваше перше завдання – вибрати тему проєкту. Ви можете вільно обрати тему самі, проте якщо складно визначитися з темою, будь ласка, повідомте мене і я запропоную для Вас кілька ідей.

Виконати можна один із трьох видів проєктів:

1. **Проєкт застосунку.** Виберіть напрямок, який Вас цікавить (наприклад, *медицина, мова, торгівля, енергетика, спорт, ігри, робототехніка тощо*) і дослідіть на практиці провідні напрацювання за обраним напрямком, що опубліковані у статтях, на github тощо.
2. **Алгоритмічний проєкт.** Виберіть проблему (задачу) або сімейство проблем і розробіть новий алгоритм навчання або новий варіант існуючого алгоритму для її вирішення.
3. **Теоретичний проєкт.** Доведіть деякі цікаві/нетривіальні властивості існуючого або нового алгоритму навчання.

Проєкти над якими Ви будете працювати можуть поєднувати елементи застосунку, алгоритмів та теорії.

Щоб визначитися із темою, Ви можете обирати або область застосування, яка Вас цікавить, або якийсь напрямок штучного інтелекту у якому хочете краще розібратися та вивчити. Отже, виберіть щось, чим зможете захопитись і приступайте до роботи. Для натхнення Ви можете переглянути деякі останні дослідження з машинного навчання. Дві основні конференції машинного навчання – це ICML та NeurIPS. Ви можете знайти статті з нещодавніх конференцій за цими гіперпосиланнями: ICML та NeurIPS.

Обравши цікаву тему, перегляньте існуючі дослідження з відповідної теми, здійснивши пошук ключових слів в академічній пошуковій системі, наприклад: http://scholar.google.com.

Іншим важливим моментом роботи над проєктом є визначення одного або декількох наборів даних, що підходять для вашої теми. Якщо ці дані потребують значної попередньої обробки відповідно до вашого завдання або Ви збираєтеся самостійно зібрати необхідні дані, майте на увазі, що це лише одна частина очікуваної проєктної роботи, яка може зайняти чимало вашого часу.

Відтворення результатів, які представлені у статті, може бути хорошим способом навчання. Однак, замість того, щоб просто тиражувати статтю, шляхом відтворення результатів, спробуйте використати

розглянутий у статті підхід в іншому застосуванні та порівняти результати або зробити якийсь аналіз того, як кожен компонент моделі впливає на кінцеву продуктивність.

Ваша проєктна робота повинна складатися з наступних частин:

- **Анотація**
- **Вступ**
- **Літературний огляд**
- **Набір даних та фічі**
- **Методи**
- **Експерименти & Результати**
- **Висновки**
- **Внесок**
- **Література**
- **Лістинг коду**

Кінцевий звіт (без двох останніх розділів: **Література** та **Лістинг коду**) повинен бути не більше 6 сторінок (включаючи усі рисунки і таблиці).

## Набори даних

Ви можете вільно користуватися наявними наборами даних для свого проєкту, що знаходяться у відкритому доступі або зібрати та створити власний.

- Kaggle – тут можна знайти деякі набори даних.
- Деякі набори даних для NLP
- OpenAI Gym – середовище для навчання з підкріпленням (reinforcement learning).

## Обчислювальні ресурси

Для виконання проєктної роботи Ви можете використовувати безкоштовно обчислювальні ресурси, які пропонують:

- Google Colab
- Kaggle

## Оцінювання

Під час оцінювання проєктної роботи буде братися до уваги:

- **Постановка завдання**. Чітко визначено завдання та зрозуміла мотивація для вирішення обраної проблеми. Новизна проблеми, технічна якість вирішення проблеми та значущість роботи. Виконано літературний огляд.
- **Дані та експерименти**. Чітко описали свій набір даних або середовище навчання та експерименти, які провели. Перерахували свої результати та метрики.
- **Методи**. Детально описано використані у роботі алгоритми навчання.
- **Висновок**. Прозора інтерпретація отриманих результатів.

Максимальна оцінка за виконання цього завдання становить 40 балів.

## Звіт проєкту

Очікується досить формальний звіт, який буде підготовлено в LaTeX. Шаблон за яким потрібно підготувати звіт проєкту подано нижче на наступних сторінках:

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

# Навчання з підкріпленням

## Проєктна робота

---

### Назва проєкту

### Виконали

Ім’я Прізвище
Група: IO-xxx
Курс: X

Ім’я2 Прізвище2
Група: IO-xxx
Курс: X

3 грудня 2021 р.

# 1 Анотація [≈ 1 абзац]

Анотація повинна бути розміром в один абзац, містити в собі мотивацію обраної теми та пояснення застосованого методу/методів і отриманих результатів.

# 2 Вступ [≈ 0.5 сторінки]

Поясніть проблему та чому вона важлива. За необхідності дайте трохи теорії. Чітко вкажіть, які вхідні та вихідні дані використовували. Наприклад: "Вхідними даними нашого алгоритму є {зображення, вік пацієнта, кількості опадів, відео у відтінках сірого тощо}. Потім ми використали {SVM, нейронну мережу, лінійну регресію тощо}, щоб вивести прогнозований {вік, ціну акцій, тип раку, музичний жанр тощо}." Це дуже важливо, оскільки різні команди, як правило, мають різні дані на вході/виході, що охоплюють різні способи застосувань. Якщо явно про це говорити – це полегшить читачу зрозуміти з чим Ви працювали.

1. Один
2. Два
3. Три
4. ...

- Один
- Два
- Три
- ...

# 3 Літературний огляд [≈ 0.5 сторінки]

Вам слід знайти існуючі статті, згрупувати їх за категоріями на основі їх підходів та обговорити їх сильні та слабкі сторони, а також те, чим вони схожі та відрізняються від вашої роботи. Google Scholar може допомогти з пошуком: http://scholar.google.com (Ви можете натиснути кнопку **"Cite** і це генерує посилання на матеріал у необхідному для вас форматі: MLA, APA, BibTeX тощо). Приклад посилання на джерела [1, 2].

# 4 Набір даних та фічі [≈ 0.5 сторінки]

Зазначте інструментарій, який був використаний для моделювання середовища. Вкажіть як здійснюється доступ до середовища. Опишіть поставлену задачу, яку має виконати агент у визначеному середовищі. Вкажіть, які спостереження може використовувати агент для визначення свого стану та які дії може виконувати у цьому середовищі. Виведіть розмір простору спостережень та розмір простору дій агента на екран. Визначте агента та поясніть як агент буде обирати, яку дію йому слід виконати для досягнення поставленої перед ним мети.

Якщо для реалізації проєкту Ви обрали методи, які засновані на навчанні з учителем (контрольоване навчання), опишіть свій набір даних: скільки прикладів було використано для навчання (training)/перевірки (validation)/тестування (test)? Чи виконували Ви попередню обробку (preprocessing), нормалізацію (normalization) або збільшення даних (data augmentation)? Якщо навчальними даними у Вас були зображення, вкажіть яка їх роздільна здатність? Додайте посилання на те, звідки ви брали дані для вирішення обраної задачі. Подайте на рисунку кілька прикладів із набору даних. Також Вам слід вказати фічі (ознаки), якими користувались. Фіча – це та ознака, від якої безпосередньо

залежить вихідний результат передбачення моделі. Наприклад, для визначення діабету у людини, такими фічами будуть: вік, стать, індекс маси тощо. Якщо витягували фічі за допомогою перетворень Фур'є, word2vec, гістограми орієнтованих градієнтів (HOG), PCA (Principal component analysis), ICA (Independent component analysis) тощо, обов'язково про це вкажіть. Приклад рисунку 1.

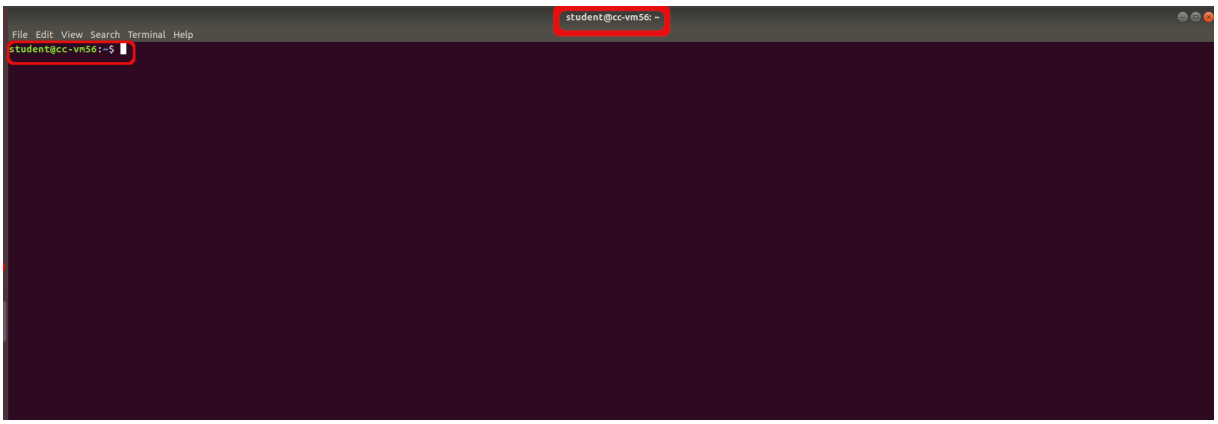



**Рис. 1:** Опис рисунку.

## 5 Методи [≈ 1-1.5 сторінки]

Опишіть використані у роботі алгоритми навчання. Обов'язково включіть відповідні математичні примітки, формули. Наприклад, Ви можете коротко включити мету, а також формулу оптимізації SVM або вказати активаційну функцію/функції, які використовувались у прихованих шарах та на виході. Для кожного апробованого алгоритму дайте короткий опис (≈ 1 абзац). Ви маєте показати своє розуміння того, як працюють ці алгоритми навчання.

## 6 Експерименти & Результати [≈ 1-3 сторінки]

Тут Ви маєте надати детальну інформацію про те, якими гіперпараметрами управляли (наприклад, чому використовували швидкість навчання (learning rate) X для градієнтного спуску, який був ваш розмір mini-batch?) та як Ви їх обирали. Чи робили Ви перехресну перевірку (cross-validation), якщо так, то скількох кратну? У k-кратній перехресній перевірці, вихідна вибірка випадковим чином розподіляється на k рівних підмножин. Перш ніж перераховувати свої результати, обов'язково вкажіть і поясніть, якими є ваші основні метрики: accuracy, precision, AUC тощо. Надайте рівняння для метрик. Для представлення результатів використовуйте таблиці та графіки з коротким поясненням та аналізом.

Якщо проєкт виконано за напрямком навчання з підкріпленням, опишіть проведені Вами експерименти. Для представлення результатів навчання агента використовуйте графік залежності винагороди від кількості ітерацій агента. Подані Вами рисунки і таблиці у звіті супроводжуйте коротким поясненням та аналізом. Деякі результати для кращого порівняння можете подати у вигляді таблиць. Приклад таблиці 1.

**Табл. 1:** Арифметичні оператори.

| Ім'я оператора | | Синтаксис |
|---|---|---|
| Присвоєння | | a = b |
| Додавання | | a + b |
| Віднімання | | a - b |
| Унарний плюс | | +a |
| Унарний мінус | | -a |
| Множення | | a * b |
| Ділення | | a / b |
| Залишок від ділення | | a % b |
| Інкремент | префікс | ++a |
| | суфікс | a++ |
| Декремент | префікс | - -a |
| | суфікс | a- - |

# 7 Висновки [≈ 1-2 абзаци]

Резюмуйте пророблену Вами роботу та повторіть ключові моменти отриманих результатів. Яку проблему вирішували та які алгоритми були найефективнішими для цього? Чому, на вашу думку, деякі алгоритми працювали краще за інші? Якби у Вас було більше часу, більше членів команди або більше обчислювальних ресурсів, що б Ви дослідили, спробували?

# 8 Внесок [≈ 1 абзац]

Цей розділ повинен містити інформацію про те, над чим кожен член команди працював та який зробив внесок у проєкт.

## Література

[1] T. I. A. for Research on Cancer. (2018) Latest global cancer data: Cancer burden rises to 18.1 million new cases and 9.6 million cancer deaths in 2018. [Online]. Available: https://www.who.int/cancer/PRGlobocanFinal.pdf

[2] S. Anisimov, D. Pandelidis, and V. Maisotsenko, "Numerical study of heat and mass transfer process in the maisotsenko cycle for indirect evaporative air cooling," *Heat Transfer Engineering*, pp. 1–40, January 28 2016. [Online]. Available: http://dx.doi.org/10.1080/01457632.2016.1142314

# Лістинг коду

Розмістіть тут програмну реалізацію цього проєкту.

Лістинг коду через зовнішній файл, наприклад так (`hello.c` та `test.py` знаходяться у директорії `code` шаблону LaTeX):

**Лістинг 1:** Назва лістингу.

```c
#include <stdio.h>

// main prints "hello world" to standard output.
int main(int argc, char **argv) {
  printf("hello world\n");
  return 0;
}
```

**Лістинг 2:** Перевірка на парність введеного числа.

```python
# Python program to check if the input number is odd or even.
# A number is even if division by 2 gives a remainder of 0.
# If the remainder is 1, it is an odd number.

num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

Або вставте безпосередньо потрібні рядки коду в оточення `lstlisting` так як показано нижче:

**Лістинг 3:** Перетин двох масивів.

```
def intersection(array_1, array_2):
  element_1 = set()
  intersected = []
  for i in range(0, len(array_1)):
    element_1.add(array_1[i])
  already_added = set()
  for j in range(0, len(array_2)):
    if array_2[j] in element_1 and array_2[j] not in already_added:
      intersected.append(array_2[j])
      # MISSING LINE HERE.
      already_added.add(array_2[j])
  return intersected
```

Без рамки та без підпису:

```
def intersection(array_1, array_2):
  element_1 = set()
  intersected = []
  for i in range(0, len(array_1)):
    element_1.add(array_1[i])
  already_added = set()
  for j in range(0, len(array_2)):
    if array_2[j] in element_1 and array_2[j] not in already_added:
      intersected.append(array_2[j])
      # MISSING LINE HERE.
      already_added.add(array_2[j])
  return intersected
```

# Література

[1] L. Weng. (2018) The multi-armed bandit problem and its solutions. [Online]. Available: https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html