MINISTRY EDUCATION AND SCIENCES UKRAINE
NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
"IGOR SIKORSKY KYIV
POLYTECHNIC INSTITUTE"

Gordienko Yu.G., Kochura Yu.P.

GENETIC ALGORITHMS

Synopsis of lectures

Tutorial
for master's degree holders
according to the educational program "Software engineering of computer systems»
specialties 121 "Software engineering"
according to the educational program "Computer systems and networks»
specialty 123 "Computer engineering"
according to the educational program "Information management systems and technologies»
specialties 126 "Information systems and technologies»

Electronic educational publication

APPROVED
at the meeting of Computer Engineering department,
protocol No. 10 on 05/25/2022

2022

Генетичні алгоритми

-

GENETIC ALGORITHMS

-

Lecture 01. Introduction

Content

Recommended Sources - Books
(some of them are used here!)

Books (classic):

Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press. **<- inventor of GA(!)**, **the highest number of citations for GA-publication** by Google Scholar!

Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press. <- **classic textbook**, **the highest number of citations for GA-textbook** by Google Scholar!

Books (with codes at github):

Wirsansky, E. (2020). *Hands-On Genetic Algorithms with Python*. Packt Publishing

Sheppard, C. (2019). *Genetic Algorithms with Python* (self-published).

Recommended Sources - Papers
(some of them are used here!)

Holland, J. H. (1992). *Genetic algorithms*. Scientific American, 267(1), 66-73. **<- inventor of GA(!) <- Just for Fun! :)**

Katoch, S., Chauhan, S. S., & Kumar, V. (2020). *A review on genetic algorithm: past, present, and future*. Multimedia Tools and Applications, 1-36.

García-Martínez, C., Rodríguez, F. J., & Lozano, M. (2018). *Genetic Algorithms*, Handbook of Heuristics, 2018, p. 431-464.

Content

- Recommended Sources
- **What are Genetic Algorithms (GAs)?**
- GA Analogy with IT
- Components of GA
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- Limitations of GAs
- When to use GAs

What are genetic algorithms?

Genetic algorithms (**GA**) are a family of search algorithms inspired by the principles of natutral evolution.

Imitating the natural selection and reproduction, GAs can produce high-quality solutions for various **problems**:

- **search,**
- **optimization,**
- **learning.**

Analogy to natural evolution allows GAs **to overcome some problems that are hard for traditional** algorithms, especially for cases with:

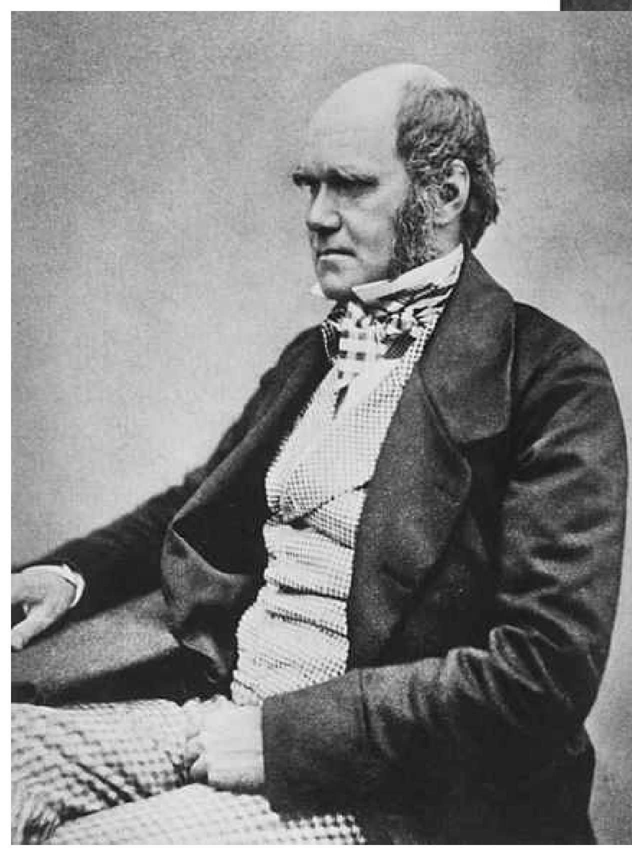
- **large number of parameters** and
- **complex mathematical representations**.

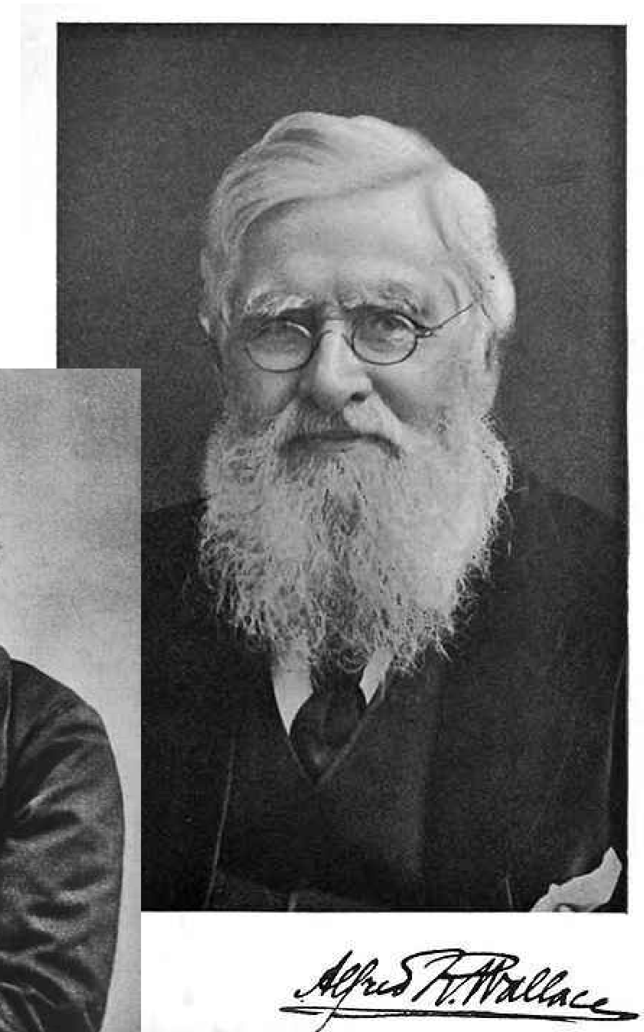
Theory behind GAs - Darwinian evolution

GAs implement a simplified version of the **Darwinian natural evolution**.

The principles of the Darwinian evolution:

- **Variation**
- **Inheritance**
- **Selection**





Theory behind GAs - Darwinian evolution

Variation:

The **traits** (**attributes**) of individual specimens belonging to a population **may vary**.

As a result, the **specimens differ** from each other to some degree,
for example, in:

- their **behavior** or
- their **appearance**.

Theory behind GAs - Darwinian evolution

Inheritance:

Some **traits** are consistently **passed** on from specimens **to their offspring**.

As a result, **offspring resemble their parents** more **than** they resemble **unrelated specimens**.

Theory behind GAs - Darwinian evolution

Selection:

Populations typically **struggle** for resources within their given environment.

The **specimens** with **traits** that are **better adapted** to the environment:

- **will be more successful** at surviving, and
- **will contribute more offspring** to the next generation.

Theory behind GAs - Darwinian evolution

Resume:

Evolution **maintains** a population of individual **specimens** that **vary** from each other.

Those who are **better adapted** to their environment have a **greater chance of surviving**, breeding, and **passing** their traits **to the next generation**.

This way, as generations go by, **species become more adapted** to their **environment** and to the **challenges** presented to them.

Content

- Recommended Sources
- What are Genetic Algorithms (GAs)?
- **GA Analogy with IT**
- Components of GA
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- Limitations of GAs
- When to use GAs

GA analogy with IT

GAs should find the **optimal solution** for a problem.

Darwinian evolution maintains a population of **individual specimens**,

BUT(!) ... GAs maintain a population of **candidate solutions** (**individuals**), for that given problem.

The **individuals** are iteratively evaluated and used to create a new generation of **individuals**.

Those who are **better** at solving this problem have a **greater** chance of being selected and passing their qualities to the next generation of **individuals**.

This way ... with generations ... **individuals** get better at solving the problem at hand.

Content

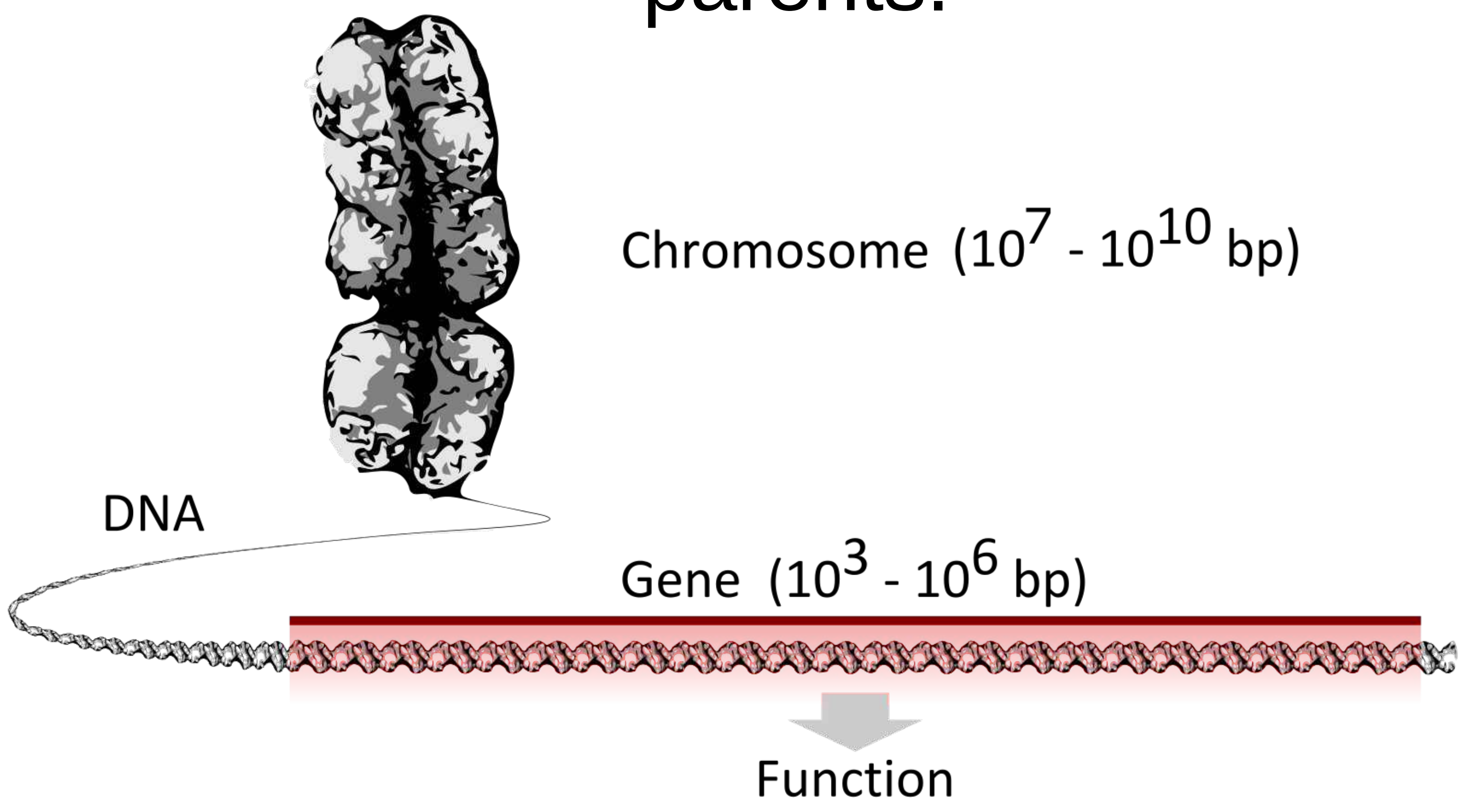
- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- **Components of GA**
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- Limitations of GAs
- When to use GAs

GA analogy with IT - Main Components

- Genotype
- Population
- Fitness function
- Selection
- Crossover
- Mutation

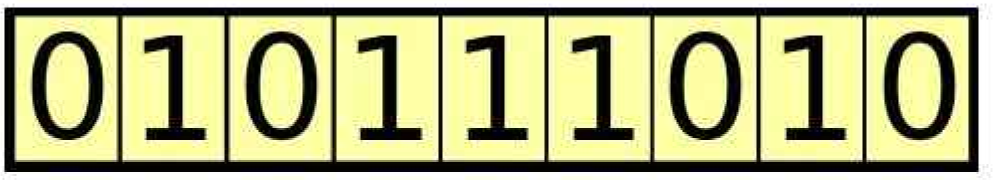
GA analogy with IT -
Main Components - Genotype

- **In biology: genotype** is a collection of **genes** that are grouped into **chromosomes**. If two specimens breed to create offspring, each **chromosome** of the offspring will **carry a mix of genes** from both parents.



GA analogy with IT - Main Components - Genotype

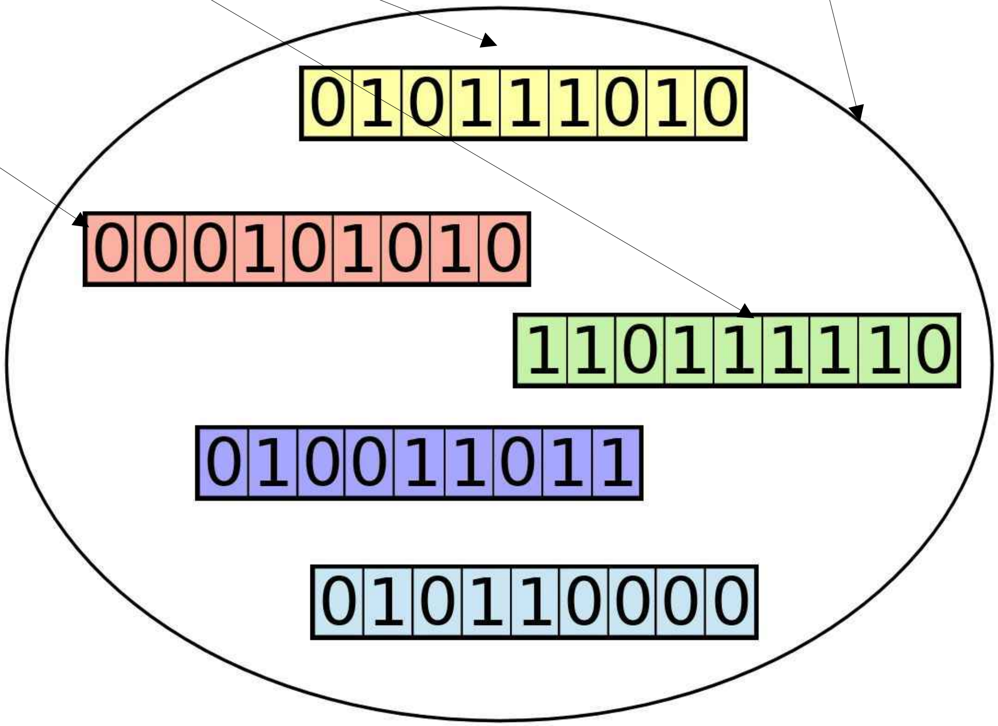
- **In IT** (GAs):
  - each **individual** is represented by **‘IT-chromosome’** that can be expressed as a **binary string**, where **each bit** represents a single **gene**.



Main Components - Population

- GAs always maintain a **population** of individuals -> a **collection** of candidate solutions for the problem. Individual -> chromosome, population -> collection of chromosomes.

The population **represents the current** generation and **evolves over time** when the current generation is replaced by a new one.



Main Components - Fitness function

At each iteration of the GA, the individuals are **evaluated** by a **fitness function** (also called the *target function*). This is the function we seek to optimize or the problem we attempt to solve.

Individuals who achieve a **better** fitness score represent **better** solutions and are more likely to be **chosen to reproduce** and be **represented** in the next generation.

Over time, the **quality** of the solutions **improves**, the **fitness** values **increase.** The process **can stop** once a solution is **found** with a **satisfactory** fitness value.

Main Components - Selection

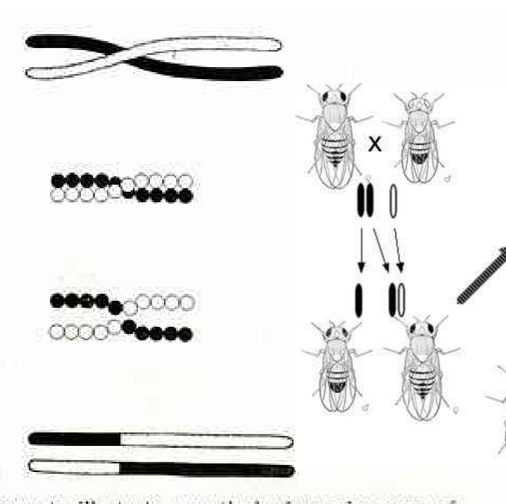
Selection process is used to **determine** which of the individuals in the population will get to **reproduce** and **create the offspring** that will form the next generation.

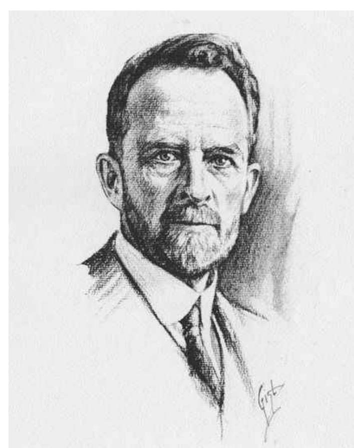
This is based on the fitness score of the individuals. Those with **higher** score values are **more likely** to be **chosen and pass** their genetic material to the next generation.

Individuals with **low fitness** values can still be chosen, but **with lower probability**. This way, their genetic material is not completely excluded.

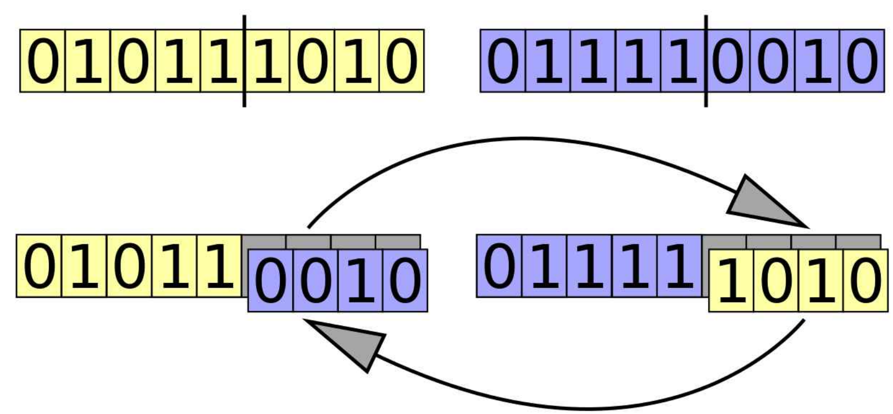
Main Components - Crossover

To create a pair of new individuals, two parents are usually chosen from the current generation, and parts of their chromosomes are **interchanged** (**crossover** or **recombination**) to create two new chromosomes representing the offspring.





Thomas Hunt Morgan's
(Nobel Prize - 1933) illustration of
crossing over (1916)

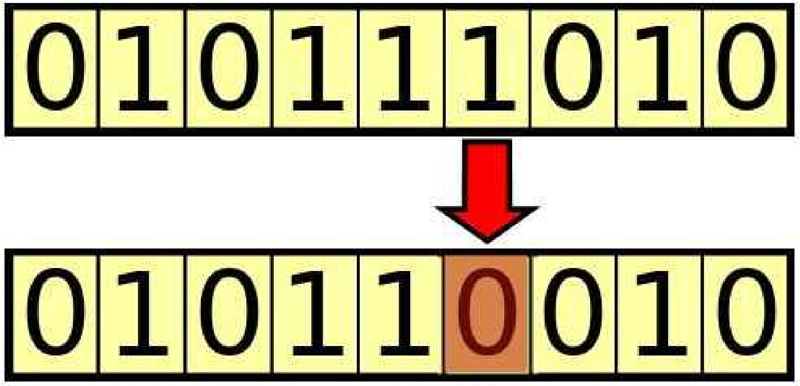


IT GA- version
of crossover (recombination)

Main Components - Mutation

The aim of **mutation** (as an operator) is to periodically and randomly **refresh the population**, **introduce new patterns** into the chromosomes, and **encourage search** in uncharted areas of the solution space.

A **mutation** can be as a **random change** in a gene, for example, flipping a bit in a binary string.



Content

- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- Components of GA
- **Main Hypothesis behind GAs**
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- Limitations of GAs
- When to use GAs

Main Hypothesis behind GAs

The **building-block hypothesis** -> the **optimal** solution to the problem is **assembled** of small building **blocks**, and as we bring **more** of these building blocks together, we get **closer** to this **optimal** solution.

Individuals in the population with the **desired building blocks** are identified by their **superior scores**.

The **repeated selection/crossover** result in the better individuals conveying these building blocks to the next generations, while possibly combining them with other successful building blocks.

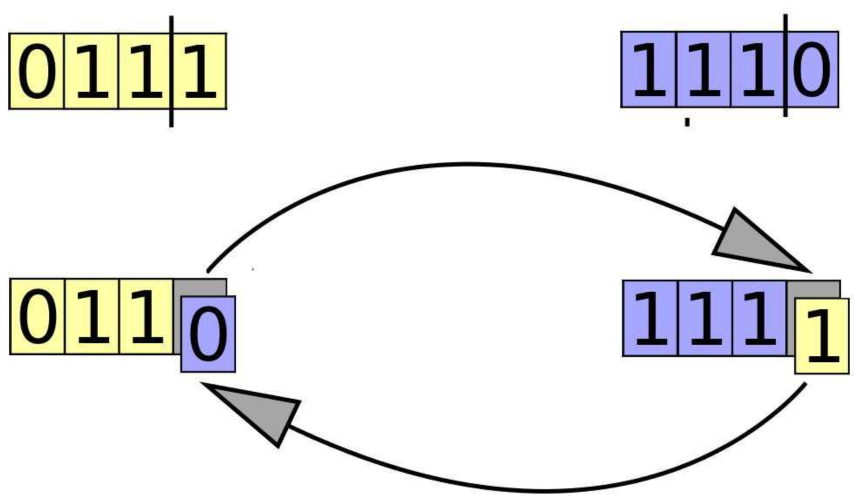
This creates **genetic pressure**, thus **guiding** the population **toward** having more individuals with the building blocks **that form the optimal** solution.

Main Hypothesis - Example

We have a population of 4-digit binary strings.

Aim: to find the string with the **largest** possible sum of digits.

Start: The digit 1 appearing at any of the 4 string positions will be a good building block.



The algorithm progresses will identify solutions that have these building blocks and bring them together. Each **new** generation will have **more** individuals with 1 values in various positions, ultimately resulting in the string 1111, which combines all the desired building blocks.

Holland's Schema Theorem

Schema is a **pattern** (or template) that can be found within the chromosomes.
It represents (as a **regular expression** with wildcards) a subset of chromosomes that have a certain similarity among them.
<u>Example:</u> if the set of chromosomes is represented by binary strings of length 4, the schema 1*01 represents all those chromosomes that have a 1 in the leftmost position, 01 in the rightmost two positions, and either a 1 or a 0 in the second from left position, since the * represents a wildcard value.

John Henry Holland
(February 2, 1929 – August 9, 2015)

“He is a **founding father of the complex systems approach**. In particular, he developed genetic algorithms and learning classifier systems”.

SANTA FE INSTITUTE

He was a member of the Board of Trustees and Science Board of the Santa Fe Institute and a fellow of the **World Economic Forum**.

THE FRANKLIN INSTITUTE

He received the 1961 **Louis E. Levy Medal** from The Franklin Institute, and the **MacArthur Fellowship** (unofficially known as the "Genius Grant") in 1992.

WORLD ECONOMIC FORUM

Holland's Schema Theorem

For each schema, one can assign two metrics:

Order:
The number of digits that are fixed (not wildcards!)

The following table provides several examples of four-digit binary schemata and their measurements:

Schema	Order	Defining Length
1101	4	3
1*01	3	3
*101	3	2
*1*1	2	2
**01	2	1
1***	1	0
****	0	0

Each chromosome in the population corresponds to multiple schemata in the same way that a given string matches regular expressions. The chromosome **1101**, for example,

Holland's Schema Theorem

The fundamental theorem of GAs:

The frequency of schemata of **low order**, **short defining length**, and **above-average fitness** increases exponentially in successive generations.

In other words: the **smaller**, **simpler** building blocks that represent the attributes **that make a solution better** will become **increasingly present** in the population as the GA progresses.

Content

- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- Components of GA
- Main Hypothesis behind GAs
- **Differences between GAs and Traditional Algorithms**
- Advantages of GAs
- Limitations of GAs
- When to use GAs

Differences GAs from Traditional Algorithms

The key characteristics of GAs distinguishing them from traditional algorithms are:

- Maintaining a **population** of solutions
- Using a **genetic representation** of the solutions
- Utilizing the outcome of a **fitness** function
- Exhibiting a **probabilistic** behavior

Differences GAs from Traditional Algorithms - Maintaining a Population of Solutions

GA **operates over a population** of candidate solutions (individuals) rather than a single candidate. GA **works with a set** of individuals that form the current generation. **Each iteration** of the GA **creates** the next generation of **set** of individuals.

In contrast, most other search algorithms **maintain a single solution** and iteratively modify it in search of the best solution.

<u>Example:</u> The ***gradient descent algorithm*** (widely used in ML/DL) iteratively **works with the current** solution (moves it in the direction of steepest descent, defined by the negative of the function's gradient).

Differences GAs from Traditional Algorithms -

Genetic Representation of Solutions

Traditional algorithms: operate directly on candidate solutions,

GAs: operate on their representations (or coding), often referred to as **chromosomes**.

Example: a chromosome is a fixed binary string.

The **genetic operations** are used for chromosomes:

- **Crossover** is interchanging chromosome parts between two parents.
- **Mutation** is modifying parts of the chromosome.

A side effect: GAs are **not aware** of what the chromosomes represent and do **not interpret** them.

Differences GAs from Traditional Algorithms - Result of Fitness Function

Fitness function (FF) represents (estimate) the problem we would like to solve.

<u>Aim of GAs:</u> to find the individuals that yield the **highest score** when this FF is calculated for them.

<u>Traditional algorithms:</u> **use the derivatives** or any other information related to FF.

<u>GAs:</u> **only** consider the **value** obtained by the FF. This allows to use FFs that are hard or impossible to mathematically differentiate.

Differences GAs from Traditional Algorithms - Probabilistic Behavior

Traditional algorithms: are **deterministic**.

GAs: the rules are **probabilistic**.

Example: when selecting the individuals that will be used to create the next generation, the **probability** of selecting a given individual **increases with** the individual's **fitness**, but there is still a random element in making that choice.

Mutation is probability-driven, usually makes changes at random location(s) in the chromosome.

Crossover can have a probabilistic element as well.

Despite the probabilistic nature, **GA is not random**; instead, it **uses the random** aspect to direct the search toward areas in the search space where there is a better chance to improve the results.

Content

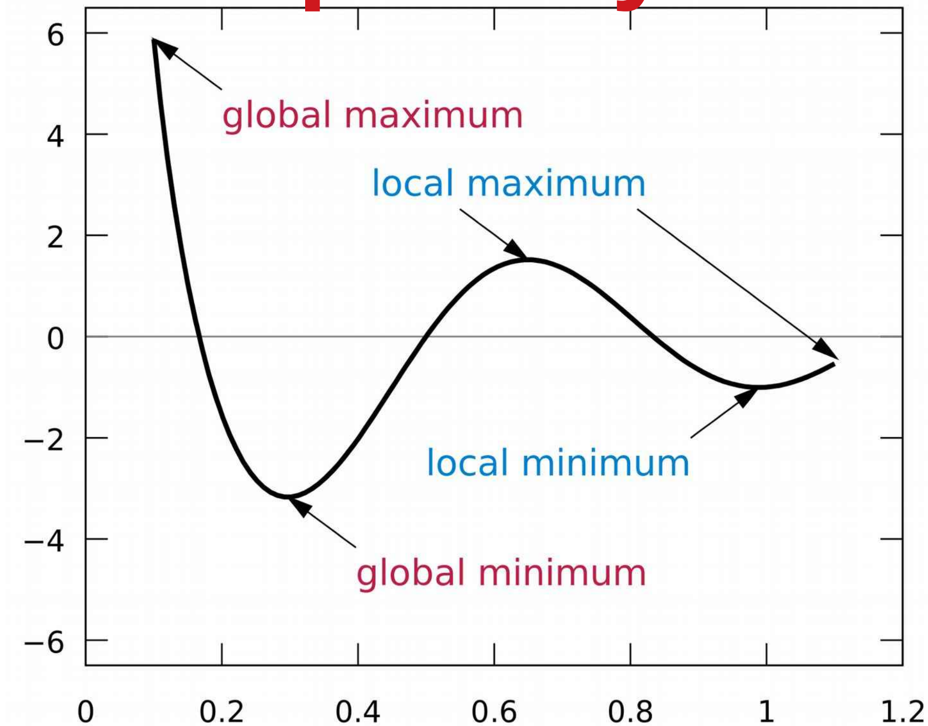
- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- Components of GA
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- **Advantages of GAs**
- Limitations of GAs
- When to use GAs

Advantages of GAs

- **Global** optimization capability
- Handling problems with a **complex** mathematical representation
- Handling problems that **lack** mathematical representation
- **Resilience** to noise
- Support for **parallelism** and distributed processing
- Suitability for **continuous learning**

Advantages of GAs - Global Optimization Capability

<u>Traditional algorithms (gradient-based)</u>:
may stuck in a local maximum rather than finding the global one
->
because near a local maximum, any small change will degrade the score.



<u>GAs:</u> are more likely to find the global maximum due to:
1 - the use of a **population** of candidate solutions,
2 - **crossover** and **mutation** that will, in many cases, result in candidate solutions that are **distant** from the previous ones.

This is **true if** we maintain the **diversity** of the population and **avoid premature convergence**.

Advantages of GAs - Complex Problems

GAs **need only the output** of FF for each individual and are not concerned with other aspects of the FF such as derivatives.

- That is why GAs **can be effective** for problems with
  - **complex mathematical representations** or
    - functions that are **hard** or impossible to **differentiate,**
  - problems with a **large number of parameters**,
    - problems with a **mix of parameter types** (combination of continuous and discrete parameters).

Advantages of GAs - Problems without Mathematical Representation

Assume that the FF score is based on human opinion.

Example:

to find the most attractive color palette for a website.

Solution:

- to try **different color combinations** and ask users to rate the attractiveness of the site;
- to apply GAs to search for the **best scoring combination** while using this opinion-based score as the fitness function outcome.

GA will do it, **despite FF has NO mathematical representation** and there is **NO way to calculate the score** directly from a given color combination.

Advantages of GAs - Resilience to Noise

Some problems present **noisy behavior**:

- even for **similar input** parameter values,

the **output** value may be **somewhat different** every time it's measured.

Examples:

- data go from sensor outputs, or
- FF score is based on human opinion.

Noisy behavior can **ruin** many **traditional** algorithms, but **GAs** are generally **resilient** to it, due to the **repetitive** operation of **reassembling** and **reevaluating** the individuals.

Advantages of GAs - Parallelism

GAs by their definition are **ready to parallelization** and **distributed processing**.

- **FF** is **independently** calculated for **each** individual, which means **all the individuals** in the population can be **evaluated concurrently**.

- Genetic operations of **selection**, **crossover**, and **mutation** can each be performed **concurrently** on individuals and pairs of individuals in the population.

That is why **GAs are natural candidates** for **distributed** and **cloud-based** implementation

Advantages of GAs - Continuous Learning

In nature, **evolution never stops**.
But it is dubious ... :) ... look around.

As the environmental **conditions change**, the **population will adapt** to them.
Similarly, **GAs can operate continuously** in an ever-changing environment, and at any point in time, the **best current solution can be fetched and used.**
But what about time?
For GAs to be effective, the **changes** in the environment **need to be slow** in relation to the generation **turnaround rate** of the GA-based search.

Content

- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- Components of GA
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- **Limitations of GAs**
- When to use GAs

Limitations of GAs

- The need for special definitions
- The need for hyperparameter tuning
- Computationally-intensive operations
- The risk of premature convergence
- No guaranteed solution

Limitations of GAs - Special Definitions

- To apply GAs to a given problem, we need to create a suitable representation for GAs and **define**:
  - **FF** and **chromosome** structure,
  - **genetic operators** (selection, crossover, and mutation) that will work for this problem.
- **This is challenging and time-consuming process!**

- **BUT ... GAs have already been applied to countless different types of problems**, and many of these definitions have been standardized.
- In other lectures some types of real-life problems will be presented that can be solved using GAs.

Limitations of GAs - Hyperparameter Tuning

The behavior of GAs is controlled by a set of **hyperparameters**, such as the **population size** and **mutation rate**, etc.

When applying GAs to the problem, **there are no exact rules (!)** for making these choices.

However, **this is true also for ... nearly all traditional search and optimization algorithms!** After doing some experimentation of your own, you will be able to make sensible choices for these values.

Limitations of GAs - Computationally-Intensive Operations

Operating on (potentially **large and very large**) populations and the **repetitive** nature of GAs can be **computationally intensive**, as well as **time consuming** before a good result is reached.

These can be alleviated by:

- a **good choice** of hyperparameters,
- implementing **parallel processing**,
- and **caching** the intermediate results (in some cases).

Limitations of GAs - Risk of Premature Convergence

If the fitness of one individual is **much higher** than the rest of the population, it may be **duplicated enough** that it **takes over** the entire population.

This can lead to the GA getting **prematurely stuck** in a **local extremum, instead of** finding the **global one.**

To prevent this from occurring,
it is important to **maintain the diversity** of the population.

Limitations of GAs - No Guaranteed Solution

- The use of GAs does **not guarantee** that the global extremum for the problem at hand **will be found**.

- However, this is almost **true for ... any traditional** search and optimization algorithm, unless it is an analytical solution for a particular type of problem.

- Generally, GAs, when used appropriately, are known to **provide good solutions** within a **reasonable amount of time**.

Content

- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- Components of GA
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- Limitations of GAs
- **When to use GAs**

Use Cases of GAs

- GAs are **best suited for** the following types of problems:
  - with **complex** mathematical representation
  - with **no** mathematical representation
  - involving a **noisy** environment
  - involving an environment that **changes over time**

Lecture 1 - DEMO A - Introduction to Genetic Algorithms

(long version) based on (C) Eyal Wirsansky work

In this lecture we introduce **DEAP — a powerful and flexible evolutionary computation** framework capable of solving real-life problems using **genetic algorithms (GA)**.

Brief Content:

- introduction,
- installation,
- main modules: *creator* and *toolbox*,
- components needed for the GA workflow,
- the simplest example, *the OneMax problem*, so called the Hello World of genetic algorithms.

By the end of this lecture you will know:

- the DEAP framework and its modules,
- the concepts of creator and toolbox in the DEAP framework,
- the simplest example of GA,
- how to create a GA solution using the DEAP framework,
- how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the OneMax problem using a GA coded with the DEAP framework,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

In these and other lectures, we will use various Python packages:

- NumPy
- Matplotlib
- Seaborn

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries
import random
import numpy
```

```
import matplotlib.pyplot as plt
```

Install DEAP by *pip* with the following code:

```
# Install DEAP
!pip install deap
```

```
Collecting deap
  Downloading https://files.pythonhosted.org/packages/0a/eb/2bd0a32e3ce757fb2
     |████████████████████████████████| 163kB 8.7MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
Installing collected packages: deap
Successfully installed deap-1.3.1
```

```
# Import DEAP
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
```

Example: OneMax problem

Constants

```
# Let's declare constants that set the parameters for the problem and control the 

# problem constants:
ONE_MAX_LENGTH = 100  # length of bit string to be optimized

# GA constants:
POPULATION_SIZE = 200
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.1   # probability for mutating an individual
MAX_GENERATIONS = 50
```

Reproducibility of Results

One important aspect of the GA is the use of probability, which introduces a random element to the behavior of the algorithm.

However, **for reproducibility of results**, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results.

To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code:

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Toolbox class

The **Toolbox** class is used as a container for functions (or operators), and enables us to create new operators by aliasing and customizing existing functions.

```
toolbox = base.Toolbox()
```

```
# For example, suppose we have a function, multiply() , defined as follows:
def multiply(a, b):
  return a*b

# Using toolbox, we can now create a new operator, incrementByFive(),
# which customizes the sumOfTwo() function as follows:
toolbox.register("MultiplyBy", multiply, b=5)

# examples:
A = toolbox.MultiplyBy(10)
print('toolbox.MultiplyBy(10) =', A)
B = multiply(10,5)
print('multiply(10,5) =', B)
```

```
toolbox.MultiplyBy(10) = 50
multiply(10,5) = 50
```

Let's create the *zeroOrOne* operator, which customizes the *random.randomint(a, b)* function.

This function normally returns a random integer *N* such that $a \leq N \leq b$.

By fixing the two arguments, *a* and *b*, to the values 0 and 1 the *zeroOrOne* operator will randomly return either the value 0 or the value 1 when called later in the code.

```
# create an operator that randomly returns 0 or 1:
toolbox.register("zeroOrOne", random.randint, 0, 1)

# examples:
A = toolbox.zeroOrOne()
print('zeroOrOne =', A)
B = toolbox.zeroOrOne()
print('zeroOrOne =', B)
C = toolbox.zeroOrOne()
print('zeroOrOne =', C)
D = toolbox.zeroOrOne()
print('zeroOrOne =', D)
```

```
zeroOrOne = 0
```

```
zeroOrOne = 0
zeroOrOne = 1
zeroOrOne = 0
```

Fitness class

Next, we need to create the *Fitness* class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the FitnessMax strategy, using a weights tuple with a single positive weight, as shown in the following code.

```
# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
A = base.Fitness.weights
print(A)
```

```
None
```

In DEAP, the *Individual* class is used to represent each of the population's individuals. This class is created with the help of the creator tool. In our case, list serves as the base class, which is used as the individual's chromosome. The class is augmented with the fitness attribute, initialized to the *FitnessMax* class that we defined earlier

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)
#creator.create("Individual", array.array, typecode='b', fitness=creator.FitnessMa
```

Next, register the *individualCreator* operator, which creates an instance of the *Individual* class, filled up with random values of either 0 or 1 . This is done by customizing the previously defined *zeroOrOne* operator.

Since the objects generated by the *zeroOrOne* operator are integers with random values of either 0 or 1 , the resulting *individualCreator* operator will fill an *Individual* instance with 100 randomly generated values of 0 or 1.

```
# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",   # Register the individualCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 creator.Individual,    # The container type (Individual) in which
                 toolbox.zeroOrOne,     # The function used to generate objects (=
                 ONE_MAX_LENGTH)        # The number of objects we want to generat
```

Register the *populationCreator* operator that creates a list of individuals.

```
# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",   # Register the populationCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 list,                  # The container type (list) in which the r
                 toolbox.individualCreator) # The function used to generate object
```

Define the function *oneMaxFitness* that computes the number of 1s in the individual.

```
# fitness calculation:
# compute the number of '1's in the individual
def oneMaxFitness(individual):
    return sum(individual),  # return a tuple,
                             # fitness values in DEAP are represented as tuples,
                             # and therefore a comma needs to follow when a single
```

Define the *evaluate* operator as an alias to the *oneMaxfitness()* function we defined earlier.

```
# create the evaluate alias for calculating the fitness (by a DEAP convention)
toolbox.register("evaluate", oneMaxFitness)
```

Genetic operators

The genetic operators are typically created by aliasing existing functions from the tools module and setting the argument values as needed.

Note: The *mutFlipBit* function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use the argument value (*indpb* parameter) as the probability of flipping (applying the not operator to) the attribute value. This value is independent of the mutation probability, which is set by the *P_MUTATION* constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the *mutFlipBit* function is called for a given individual in the population.

```
# genetic operators:

# Tournament selection with tournament size of 3:
toolbox.register("select", tools.selTournament, tournsize=3)

# Single-point crossover:
toolbox.register("mate", tools.cxOnePoint)

# Flip-bit mutation:
# indpb: Independent probability for each attribute to be flipped
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/ONE_MAX_LENGTH)
```

GA workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)
generationCounter = 0
```

Long version

```
# calculate fitness tuple for each individual in the population:
fitnessValues = list(map(toolbox.evaluate, population)) # use map() to apply the ev
for individual, fitnessValue in zip(population, fitnessValues):
  individual.fitness.values = fitnessValue
```

```
# extract the first value out of each fitness for gathering statistics:
fitnessValues = [individual.fitness.values[0] for individual in population]
```

```
# initialize statistics accumulators:
maxFitnessValues = []
meanFitnessValues = []
```

```
# main evolutionary loop:
# stop if max fitness value reached the known max value
# OR if number of generations exceeded the preset value:
while max(fitnessValues) < ONE_MAX_LENGTH and generationCounter < MAX_GENERATIONS:
  # update counter:
  generationCounter = generationCounter + 1

  # apply the selection operator, to select the next generation's individuals:
  offspring = toolbox.select(population, len(population))
  # clone the selected individuals:
  offspring = list(map(toolbox.clone, offspring))

  # apply the crossover operator to pairs of offspring:
  for child1, child2 in zip(offspring[::2], offspring[1::2]):
    if random.random() < P_CROSSOVER:
      toolbox.mate(child1, child2)
      del child1.fitness.values
      del child2.fitness.values

      for mutant in offspring:
        if random.random() < P_MUTATION:
          toolbox.mutate(mutant)
          del mutant.fitness.values

  # calculate fitness for the individuals with no previous calculated fitness valu
  freshIndividuals = [ind for ind in offspring if not ind.fitness.valid]
  freshFitnessValues = list(map(toolbox.evaluate, freshIndividuals))
  for individual, fitnessValue in zip(freshIndividuals, freshFitnessValues):
    individual.fitness.values = fitnessValue
```

```
# replace the current population with the offspring:
population[:] = offspring

# collect fitnessValues into a list, update statistics and print:
fitnessValues = [ind.fitness.values[0] for ind in population]

maxFitness = max(fitnessValues)
meanFitness = sum(fitnessValues) / len(population)
maxFitnessValues.append(maxFitness)
meanFitnessValues.append(meanFitness)
print("- Generation {}: Max Fitness = {}, Avg Fitness = {}".format(generationCou

# find and print best individual:
best_index = fitnessValues.index(max(fitnessValues))
print("Best Individual = ", *population[best_index], "\n")
```

```
- Generation 1: Max Fitness = 62.0, Avg Fitness = 52.59
Best Individual =  0 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 1 1

- Generation 2: Max Fitness = 64.0, Avg Fitness = 55.205
Best Individual =  0 1 1 1 0 0 0 1 1 1 0 1 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0 1

- Generation 3: Max Fitness = 67.0, Avg Fitness = 56.88
Best Individual =  1 1 0 1 1 1 0 1 0 1 1 0 1 0 1 1 1 1 0 0 1 0 0 1 1 1 1 0

- Generation 4: Max Fitness = 71.0, Avg Fitness = 58.425
Best Individual =  1 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 1 1

- Generation 5: Max Fitness = 69.0, Avg Fitness = 59.77
Best Individual =  0 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1

- Generation 6: Max Fitness = 73.0, Avg Fitness = 61.53
Best Individual =  1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1

- Generation 7: Max Fitness = 73.0, Avg Fitness = 62.525
Best Individual =  1 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0 0 1 1 1 0 1 1 1 0 1

- Generation 8: Max Fitness = 74.0, Avg Fitness = 63.23
Best Individual =  1 1 0 1 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1

- Generation 9: Max Fitness = 74.0, Avg Fitness = 63.76
Best Individual =  1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1

- Generation 10: Max Fitness = 74.0, Avg Fitness = 64.165
Best Individual =  1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 0 1 0 0 0 1 1 1 1 1 1

- Generation 11: Max Fitness = 75.0, Avg Fitness = 64.23
Best Individual =  1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1

- Generation 12: Max Fitness = 75.0, Avg Fitness = 64.83
Best Individual =  1 0 1 0 1 1 1 0 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1

- Generation 13: Max Fitness = 78.0, Avg Fitness = 65.225
Best Individual =  1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 1

- Generation 14: Max Fitness = 80.0, Avg Fitness = 65.355
Best Individual =  1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1
```

```
- Generation 15: Max Fitness = 75.0, Avg Fitness = 65.87
Best Individual =  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1

- Generation 16: Max Fitness = 76.0, Avg Fitness = 65.705
Best Individual =  0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 0 0 1 0 1

- Generation 17: Max Fitness = 75.0, Avg Fitness = 65.845
Best Individual =  1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1

- Generation 18: Max Fitness = 79.0, Avg Fitness = 66.02
Best Individual =  1 0 1 0 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1

- Generation 19: Max Fitness = 80.0, Avg Fitness = 66.44
Best Individual =  1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1
```

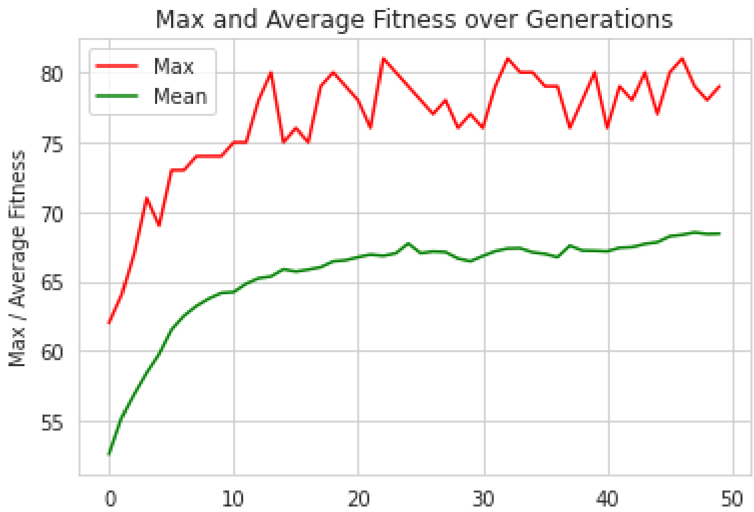
You should get the following output:

– Generation 1: Max Fitness = 62.0, Avg Fitness = 52.59 Best Individual = 0 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 1 1 1 0 0 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 1 1 1 1 0 1 0

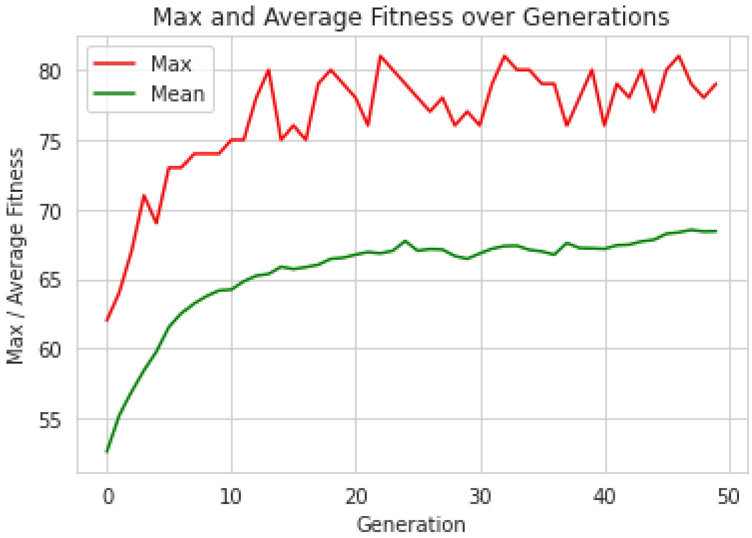
...

– Generation 50: Max Fitness = 79.0, Avg Fitness = 68.43 Best Individual = 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1

```
# Plot statistics:
sns.set_style("whitegrid")
plt.plot(maxFitnessValues, color='red', label='Max')
plt.plot(meanFitnessValues, color='green', label='Mean')
plt.xlabel('Generation')
plt.ylabel('Max / Average Fitness')
plt.title('Max and Average Fitness over Generations')
plt.legend()
plt.show()
```



You should get the following output:


Max and Average Fitness over Generations
Max
Mean
Max / Average Fitness
Generation
80
75
70
65
60
55
0
10
20
30
40
50

Lecture 1 - DEMO B - Introduction to Genetic Algorithms

(short version of code implementation) based on (C) Eyal Wirsansky work

In this lecture we introduce **DEAP — a powerful and flexible evolutionary computation** framework capable of solving real-life problems using **genetic algorithms (GA)**.

Brief Content:

- introduction,
- installation,
- main modules: *creator* and *toolbox*,
- components needed for the GA workflow,
- the simplest example, *the OneMax problem*, so called the Hello World of genetic algorithms.

By the end of this lecture you will know:

- the DEAP framework and its modules,
- the concepts of creator and toolbox in the DEAP framework,
- the simplest example of GA,
- how to create a GA solution using the DEAP framework,
- how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the OneMax problem using a GA coded with the DEAP framework,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

In these and other lectures, we will use various Python packages:

- [NumPy]
- [Matplotlib]
- [Seaborn]

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries
import random
import numpy
```

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

Install DEAP by *pip* with the following code:

```
# Install DEAP
!pip install deap
```

```
    Collecting deap
      Downloading https://files.pythonhosted.org/packages/0a/eb/2bd0a32e3ce757fb2
         |████████████████████████████████| 163kB 8.2MB/s
    Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
    Installing collected packages: deap
    Successfully installed deap-1.3.1
```

```
# Import DEAP
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
```

Example: OneMax problem

Constants

```
# Let's declare constants that set the parameters for the problem and control the 

# problem constants:
ONE_MAX_LENGTH = 100  # length of bit string to be optimized

# GA constants:
POPULATION_SIZE = 200
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.1   # probability for mutating an individual
MAX_GENERATIONS = 50
```

Reproducibility of Results

One important aspect of the GA is the use of probability, which introduces a random element to the behavior of the algorithm.

However, **for reproducibility of results**, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results.

To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code:

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Toolbox class

The **Toolbox** class is used as a container for functions (or operators), and enables us to create new operators by aliasing and customizing existing functions.

```
toolbox = base.Toolbox()
```

```
# For example, suppose we have a function, multiply() , defined as follows:
def multiply(a, b):
  return a*b

# Using toolbox, we can now create a new operator, incrementByFive(),
# which customizes the sumOfTwo() function as follows:
toolbox.register("MultiplyBy", multiply, b=5)

# examples:
A = toolbox.MultiplyBy(10)
print('toolbox.MultiplyBy(10) =', A)
B = multiply(10,5)
print('multiply(10,5) =', B)
```

```
toolbox.MultiplyBy(10) = 50
multiply(10,5) = 50
```

Let's create the *zeroOrOne* operator, which customizes the *random.randomint(a, b)* function.

This function normally returns a random integer *N* such that $a \leq N \leq b$.

By fixing the two arguments, *a* and *b*, to the values 0 and 1 the *zeroOrOne* operator will randomly return either the value 0 or the value 1 when called later in the code.

```
# create an operator that randomly returns 0 or 1:
toolbox.register("zeroOrOne", random.randint, 0, 1)

# examples:
A = toolbox.zeroOrOne()
print('zeroOrOne =', A)
B = toolbox.zeroOrOne()
print('zeroOrOne =', B)
C = toolbox.zeroOrOne()
print('zeroOrOne =', C)
```

```
D = toolbox.zeroOrOne()
print('zeroOrOne =', D)
```

```
zeroOrOne = 0
zeroOrOne = 0
zeroOrOne = 1
zeroOrOne = 0
```

Fitness class

Next, we need to create the *Fitness* class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the FitnessMax strategy, using a weights tuple with a single positive weight, as shown in the following code.

```
# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
A = base.Fitness.weights
print(A)
```

```
None
```

In DEAP, the *Individual* class is used to represent each of the population's individuals. This class is created with the help of the creator tool. In our case, list serves as the base class, which is used as the individual's chromosome. The class is augmented with the fitness attribute, initialized to the *FitnessMax* class that we defined earlier

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)
#creator.create("Individual", array.array, typecode='b', fitness=creator.FitnessMa
```

Next, register the *individualCreator* operator, which creates an instance of the *Individual* class, filled up with random values of either 0 or 1 . This is done by customizing the previously defined *zeroOrOne* operator.

Since the objects generated by the *zeroOrOne* operator are integers with random values of either 0 or 1 , the resulting *individualCreator* operator will fill an *Individual* instance with 100 randomly generated values of 0 or 1.

```
# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",   # Register the individualCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 creator.Individual,    # The container type (Individual) in which
                 toolbox.zeroOrOne,     # The function used to generate objects (=
                 ONE_MAX_LENGTH)        # The number of objects we want to generat
```

Register the *populationCreator* operator that creates a list of individuals.

```
# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",   # Register the populationCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 list,                  # The container type (list) in which the r
                 toolbox.individualCreator) # The function used to generate object
```

Define the function *oneMaxFitness* that computes the number of 1s in the individual.

```
# fitness calculation:
# compute the number of '1's in the individual
def oneMaxFitness(individual):
    return sum(individual),  # return a tuple,
                             # fitness values in DEAP are represented as tuples,
                             # and therefore a comma needs to follow when a single
```

Define the *evaluate* operator as an alias to the *oneMaxfitness()* function we defined earlier.

```
# create the evaluate alias for calculating the fitness (by a DEAP convention)
toolbox.register("evaluate", oneMaxFitness)
```

Genetic operators

The genetic operators are typically created by aliasing existing functions from the tools module and setting the argument values as needed.

Note: The *mutFlipBit* function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use the argument value (*indpb* parameter) as the probability of flipping (applying the not operator to) the attribute value. This value is independent of the mutation probability, which is set by the *P_MUTATION* constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the *mutFlipBit* function is called for a given individual in the population.

```
# genetic operators:

# Tournament selection with tournament size of 3:
toolbox.register("select", tools.selTournament, tournsize=3)

# Single-point crossover:
toolbox.register("mate", tools.cxOnePoint)

# Flip-bit mutation:
# indpb: Independent probability for each attribute to be flipped
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/ONE_MAX_LENGTH)
```

GA workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

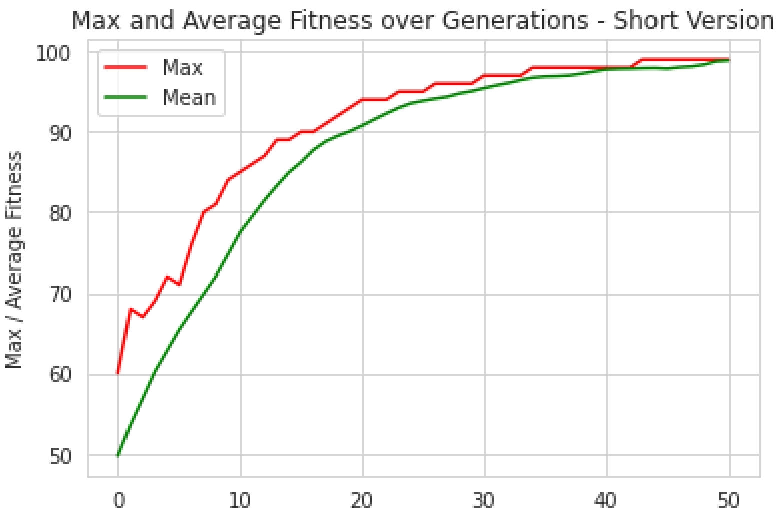
Short version

```
# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# perform the Genetic Algorithm flow:
population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P_CROSSOVER, mu
                                           stats=stats, verbose=True)

# Genetic Algorithm is done - extract statistics:
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

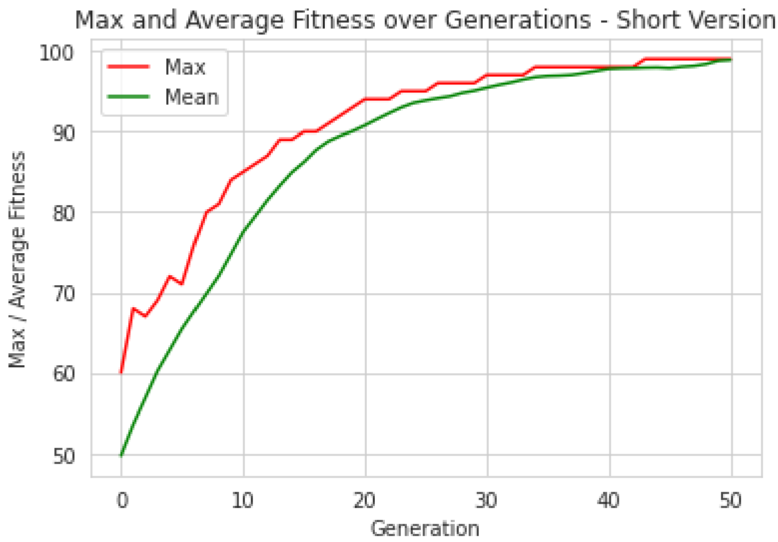
```
gen	nevals	max	avg
0  	200   	60 	49.705
1  	190   	68 	53.56 
2  	175   	67 	56.87 
3  	179   	69 	60.21 
4  	175   	72 	62.825
5  	184   	71 	65.45 
6  	178   	76 	67.68 
7  	187   	80 	69.865
8  	189   	81 	72.055
9  	184   	84 	74.765
10 	185   	85 	77.515
11 	181   	86 	79.485
12 	190   	87 	81.49 
13 	181   	89 	83.27 
14 	184   	89 	84.94 
15 	189   	90 	86.22 
16 	176   	90 	87.725
17 	176   	91 	88.79 
18 	182   	92 	89.485
19 	185   	93 	90.065
20 	182   	94 	90.765
21 	170   	94 	91.535
22 	179   	94 	92.28 
23 	178   	95 	92.985
24 	181   	95 	93.545
25 	189   	95 	93.855
26 	174   	96 	94.125
27 	179   	96 	94.36 
28 	186   	96 	94.78 
29 	185   	96 	95.055
30 	185   	97 	95.43 
```

```
31        186       97        95.775
32        187       97        96.075
33        179       97        96.435
34        176       98        96.745
35        187       98        96.885
36        186       98        96.93
37        190       98        97.015
38        175       98        97.245
39        171       98        97.515
40        179       98        97.78
41        188       98        97.845
42        188       98        97.87
43        178       99        97.925
44        174       99        97.95
45        176       99        97.87
46        185       99        98.04
47        184       99        98.14
48        184       99        98.37
49        187       99        98.79
50        185       99        98.885
```

```
# Plot statistics:
sns.set_style("whitegrid")
plt.plot(maxFitnessValues, color='red', label='Max')
plt.plot(meanFitnessValues, color='green', label='Mean')
plt.xlabel('Generation')
plt.ylabel('Max / Average Fitness')
plt.title('Max and Average Fitness over Generations - Short Version')
plt.legend()
plt.show()
```



You should get the following output:



Colab paid products - Cancel contracts here

Основи еволюційних обчислень
-
Evolutionary Computing Basics
-
Lecture 02. Overview

(based on Alan Turing, Holland, Khaled Rasheed, Ben Phillips, Eyal Wirsansky, and others works)

... in previous lecture ...

Content

- Recommended Sources
- What are Genetic Algorithms (GAs)?
- GA Analogy with IT
- Components of GA
- Main Hypothesis behind GAs
- Differences between GAs and Traditional Algorithms
- Advantages of GAs
- Limitations of GAs
- When to use GAs

Content — this lecture

- **Recommended Sources**
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA

Recommended Sources - Books

(the same as for GA!)

Books (classic):

Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press. **<- inventor of GA(!)**, **the highest number of citations for GA-publication** by Google Scholar!

Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press. <- **classic textbook**, **the highest number of citations for GA-textbook** by Google Scholar!

Books (with codes at github):

Wirsansky, E. (2020). *Hands-On Genetic Algorithms with Python*. Packt Publishing

Sheppard, C. (2019). *Genetic Algorithms with Python* (self-published).

Recommended Sources - Papers
(the same as for GA!)

Holland, J. H. (1992). *Genetic algorithms*. Scientific American, 267(1), 66-73. **<- inventor of GA(!) <- Just for Fun! :)**

Katoch, S., Chauhan, S. S., & Kumar, V. (2020). *A review on genetic algorithm: past, present, and future*. Multimedia Tools and Applications, 1-36.

García-Martínez, C., Rodríguez, F. J., & Lozano, M. (2018). *Genetic Algorithms*, Handbook of Heuristics, 2018, p. 431-464.

Content

- Recommended Sources
- **What is Evolutionary Computing (EC)**
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA

What is Evolutionary Computing (EC)?

EVOLUTION		IT
Environment	⟷	Problem
Individual	⟷	Candidate Solution (Individual)
Fitness	⟷	Quality

Fitness → chances for survival and reproduction

Quality → chance for seeding new solutions

What is EC — Metaphor (nature-IT)

A **population** of **individuals** exists in an environment with **limited** resources.

Competition for those **resources** causes **selection** of those **fitter** individuals that are **better adapted** to the environment.

These **individuals** act as **seeds** for the **new generation** of individuals through some **variation operations** (for example, GA like recombination and mutation).

The new individuals have their **fitness evaluated** and compete (possibly also with parents) **for survival**.

Natural selection causes a **rise in the fitness** of the population

Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- **EC History**
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA

EC — History — Founders

1948, Turing:
“genetical or evolutionary search”
1962, Bremermann
optimization through evolution and recombination
1964, Rechenberg
evolution strategies
1965, L. Fogel, Owens and Walsh
evolutionary programming
1975, Holland
genetic algorithms
1992, Koza
genetic programming

EC — History — Community

1985:
first international **conference** (ICGA)

1990:
first international **conference in Europe** (PPSN)

1993:
first scientific EC **journal** (MIT Press)

1997:
launch of European EC **Research Network** EvoNet

EC — History — NOW!

- 3 **major** EC conferences
  + 10 **small** related ones
- 3 scientific **core EC journals**
- **750-1000 papers** published in 2003
- numerous applications
- numerous consultancy and R&D firms

EC — History — Lessons

Nature has always served as a source of inspiration for engineers and scientists

The best problem solver known in nature is:

- the (human) brain that created "the wheel, New York, wars and so on" (after Douglas Adams' Hitch-Hikers Guide)
- the evolution mechanism that created the human brain (after Darwin's Origin of Species)

Answer 1 ☾ neurocomputing

Answer 2 ☾ evolutionary computing

EC — Current Needs

Developing, analyzing, applying
problem solving methods (algorithms)
is a central theme
in mathematics and computer science.

Why?

- **Time** for careful problem analysis decreases
- **Complexity** of the current problems increases

Resume:

Robust problem solving technology needed!

Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- **Problem Types for EC**
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA

EC — Problem Types

We have
a model, inputs and outputs
of our system
and look for different entities:

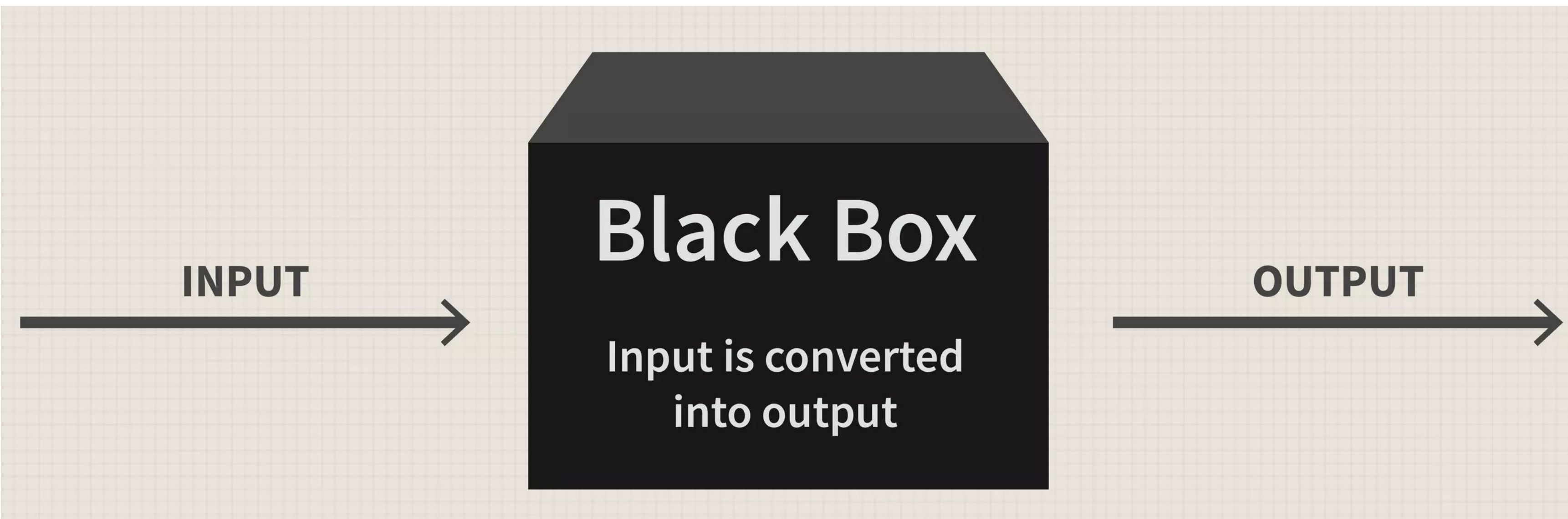
- optimization,

- modeling,

- simulation.

Problem Types – Optimization

We **have the model** of our system and
seek inputs that give us a specified goal:

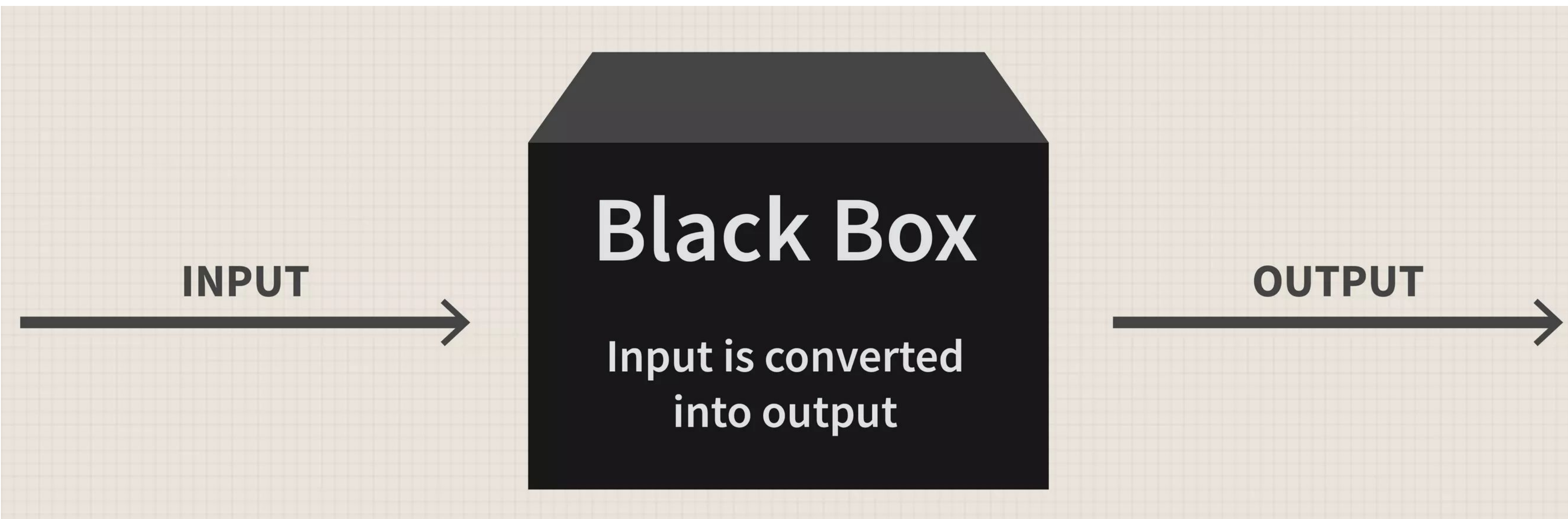


Input? **The model is known!**

We **look for inputs** to reach the specified goal, for example:
- time table for KPI (rozklad.kpi.ua - fantastic!),
- software/hardware design specifications,
- etc.

Problem Types – Modeling

We **have the corresponding input/output sets** of our system and **seek model** that give us a specified goal:

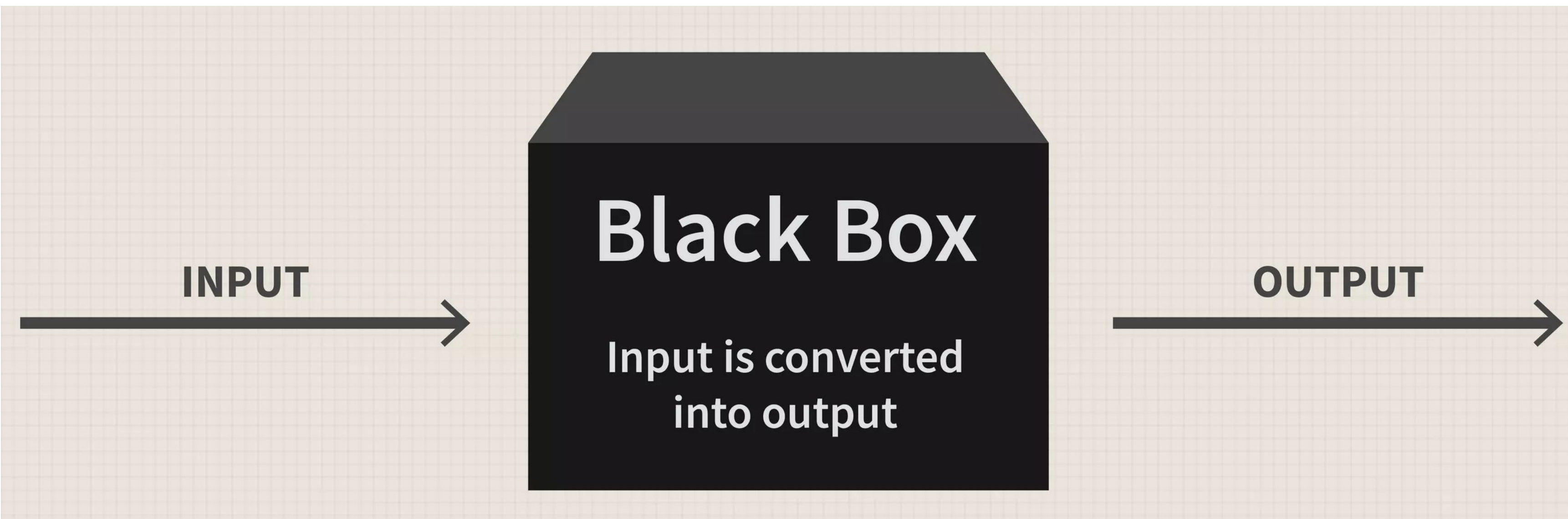


The input is known! **Model?** **The output is known!**

The model should deliver the **correct output** for every **known input**, for example:
- machine learning models,
- deep learning models.

Problem Types – Simulation

We **have the model** of our system and look for the **outputs** that will appear under different **inputs**:



The input is known! The model is known! Output?

It is used to **investigate scenarios** the **evolving** dynamic environments:

- evolutionary economics,
- geo-politics,
- military planning,
- artificial life

Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- **What is Evolutionary Algorithm (EA)**
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA

Again:
What is EC — Metaphor (nature-IT)

EVOLUTION		IT
		-
Environment	⟷	Problem
Individual	⟷	Candidate Solution (Individual)
Fitness	⟷	Quality

Fitness → chances for survival and reproduction

Quality → chance for seeding new solutions

What is Evolutionary Algorithms (EA) — Metaphor (nature-IT)

EAs is the category of **“generate and test”** algorithms.

They are stochastic, **population-based** algorithms.

Variation (genetic?) **operators** (recombination and mutation) **create** the necessary **diversity** and thereby facilitate **novelty**.

Selection reduces(!) diversity
and
acts as a **force pushing quality**.

EA — History and Types

Different types of EAs have been associated with different representations:

- **Binary** strings : Genetic Algorithms (**GA**)
- **Real**-valued vectors : Evolution Strategies (**ES**)
- **Finite state** Machines: Evolutionary Programming (**EP**)
- LISP **trees**: Genetic Programming (**GP**)

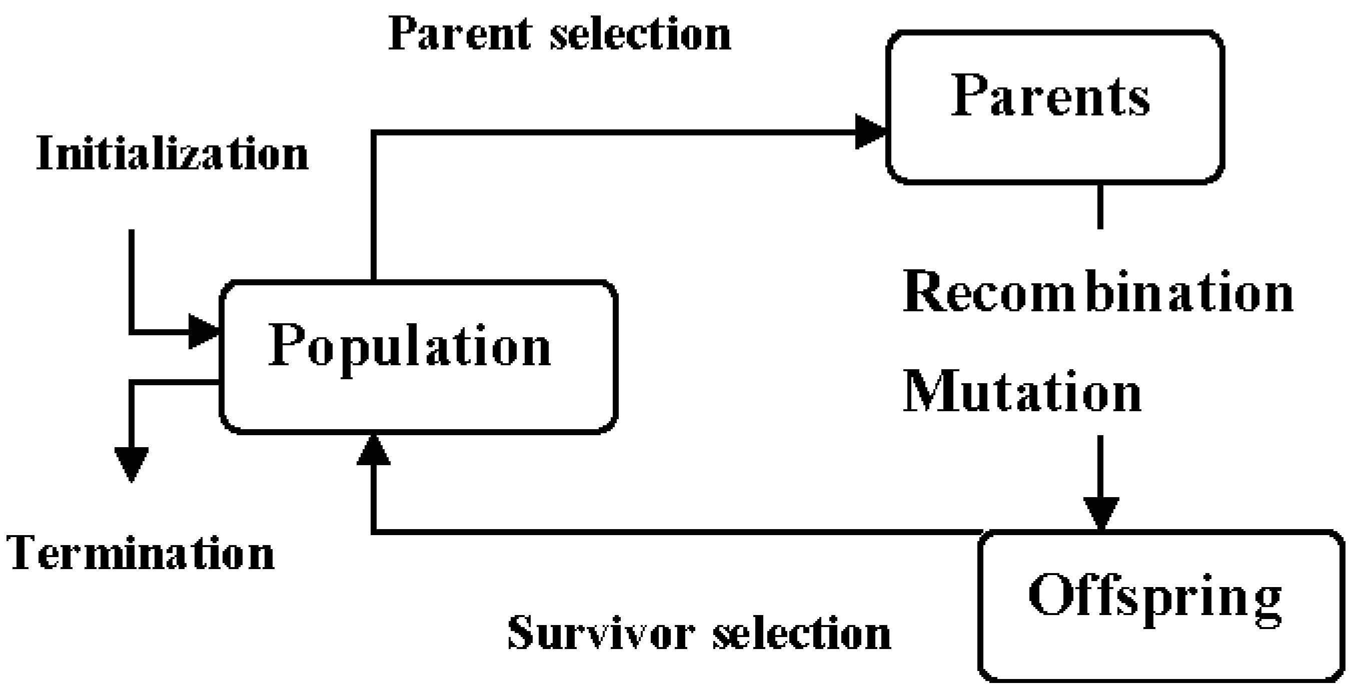
- These differences are largely irrelevant, best strategy
  - choose **representation** to suit **problem**
  - choose **variation operators** to suit **representation**

- **Selection** operators **only use fitness**
  - and so
  - are **independent of representation**.

Content

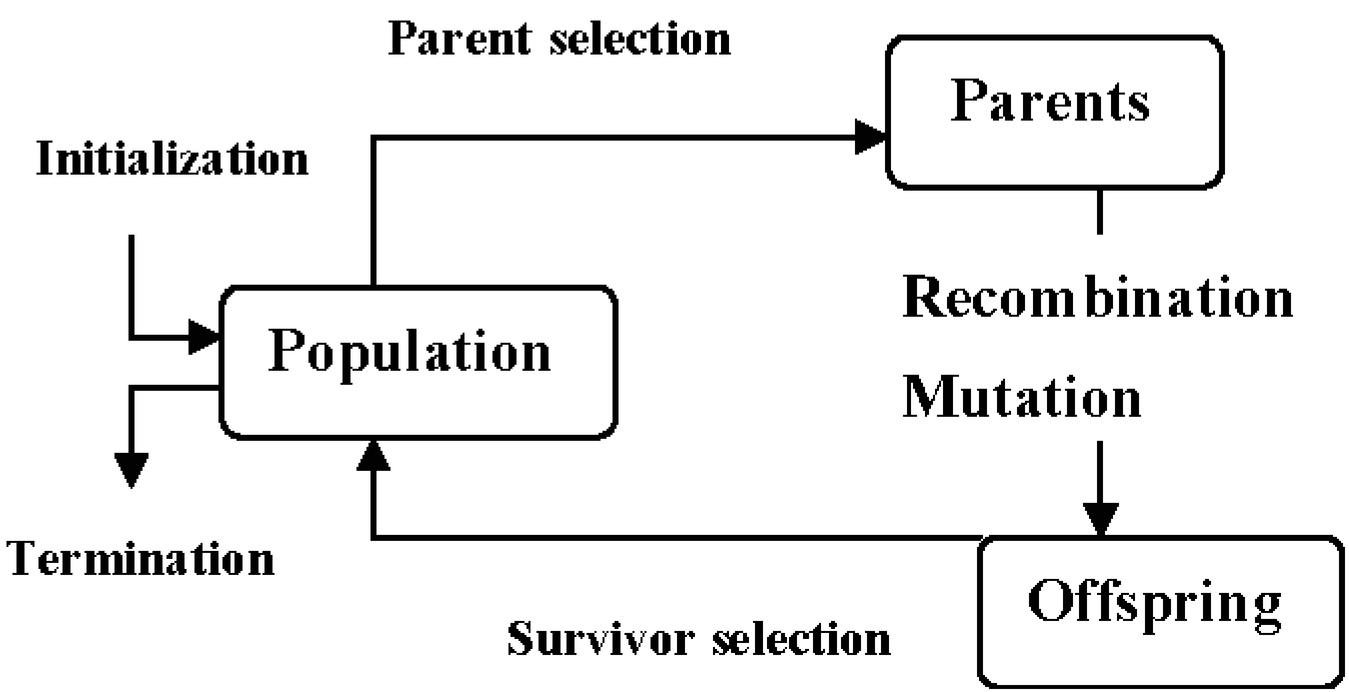
- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA

EA — General Scheme ...

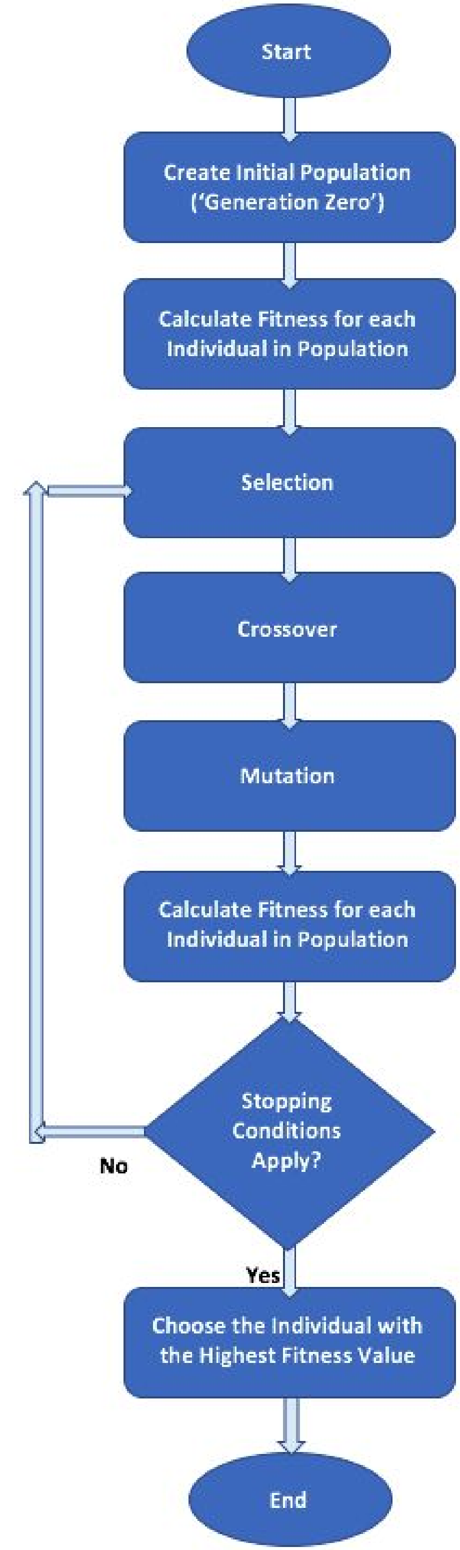


... and ...

EA — General Scheme ...



... and Workflow ---------------------->



EA — Workflow — Terminology

Candidate solutions (**individuals**) exist in **phenotype space**.

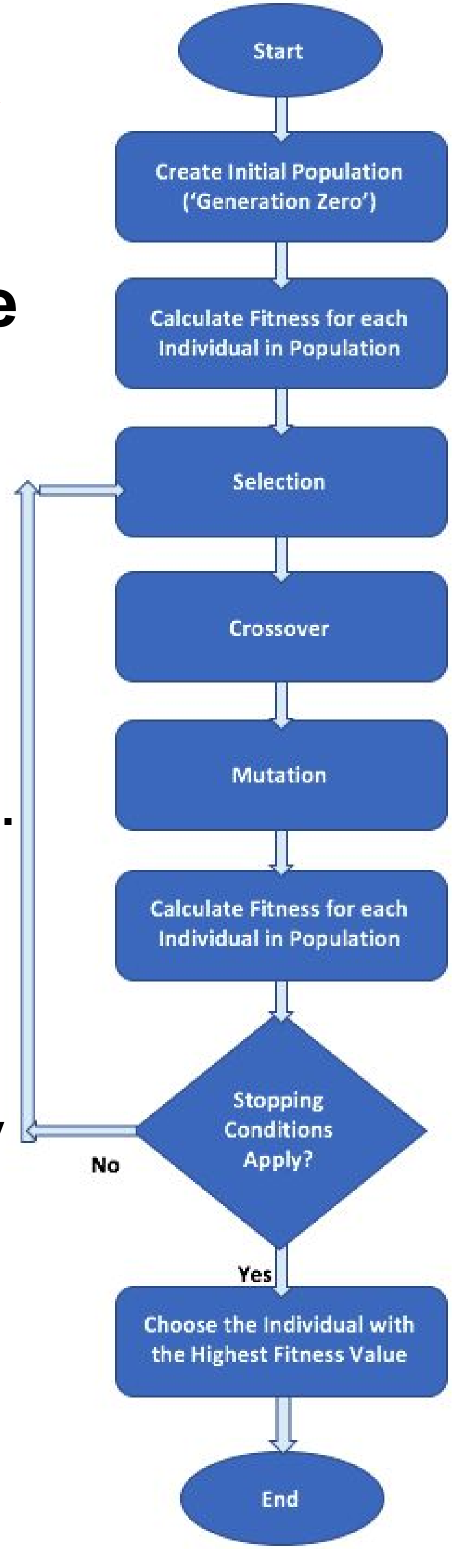
They are **encoded** in **chromosomes**, which exist in **genotype space**.

Encoding: phenotype->genotype (not always 1-to-1).

Decoding: genotype->phenotype (must be 1-to-1).

Chromosomes contain **genes**, which are in (usually fixed) positions called **loci** (sing. **locus**) and have a value (allele).

To find the **global optimum**, every feasible solution must be representable in genotype space!



EA — Workflow — Population

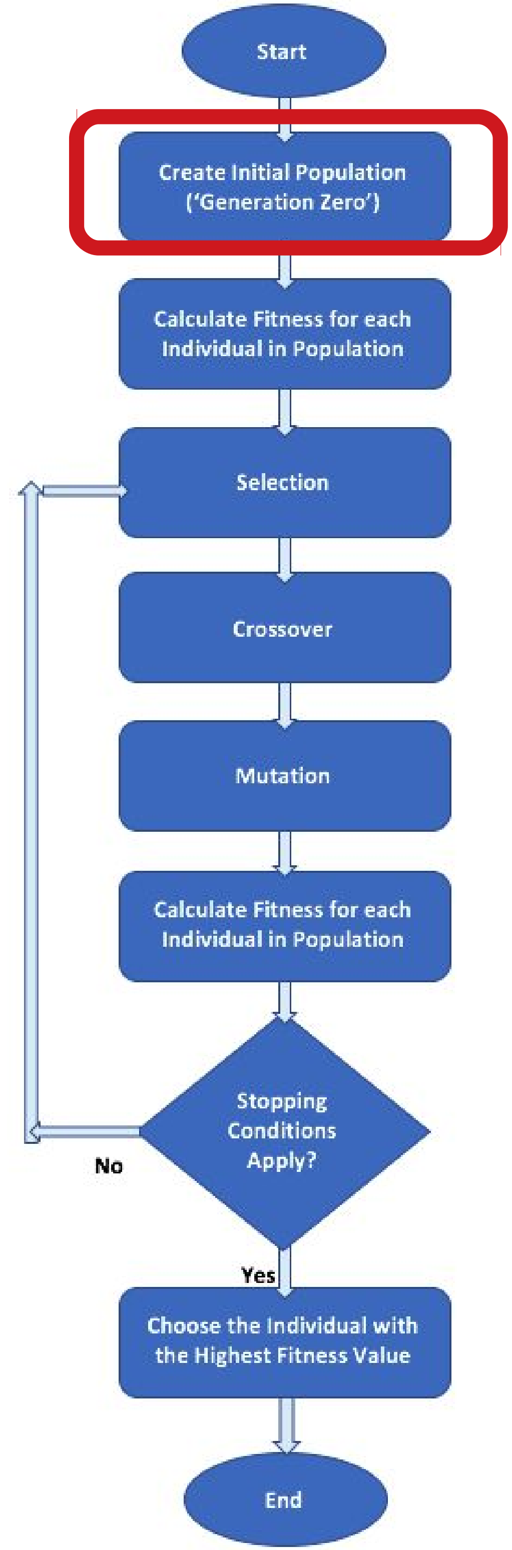
Has (representations of) **possible solutions.**

Usually has a **fixed size** and is a **multi-set of genotypes.**

Some sophisticated EAs also assert a **spatial structure** on the population e.g., a grid.

Selection operators work with **whole population** into account i.e., reproductive probabilities are relative to current generation.

Diversity of a population refers to the number of **different fitnesses / phenotypes / genotypes** present (note not the same thing).



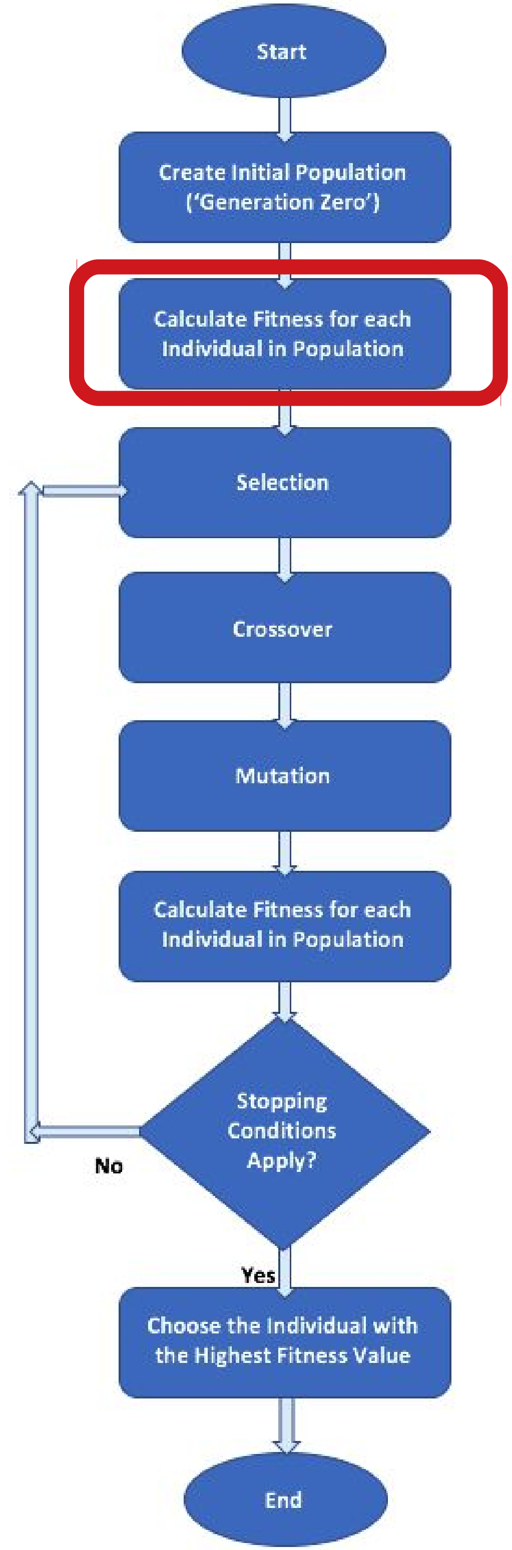
EA — Workflow — Fitness

Represents the requirements that the **population** should **adapt** to some **criteria** like **quality** function or **objective** function.

Assigns a single **real-value**d fitness to each **phenotype** which forms the **basis for selection.**

So the **more diversity** (different values) **the better.**

Typically **fitness** is assumed to be **maximized, but** ... some problems can be formulated as **minimization** problems.

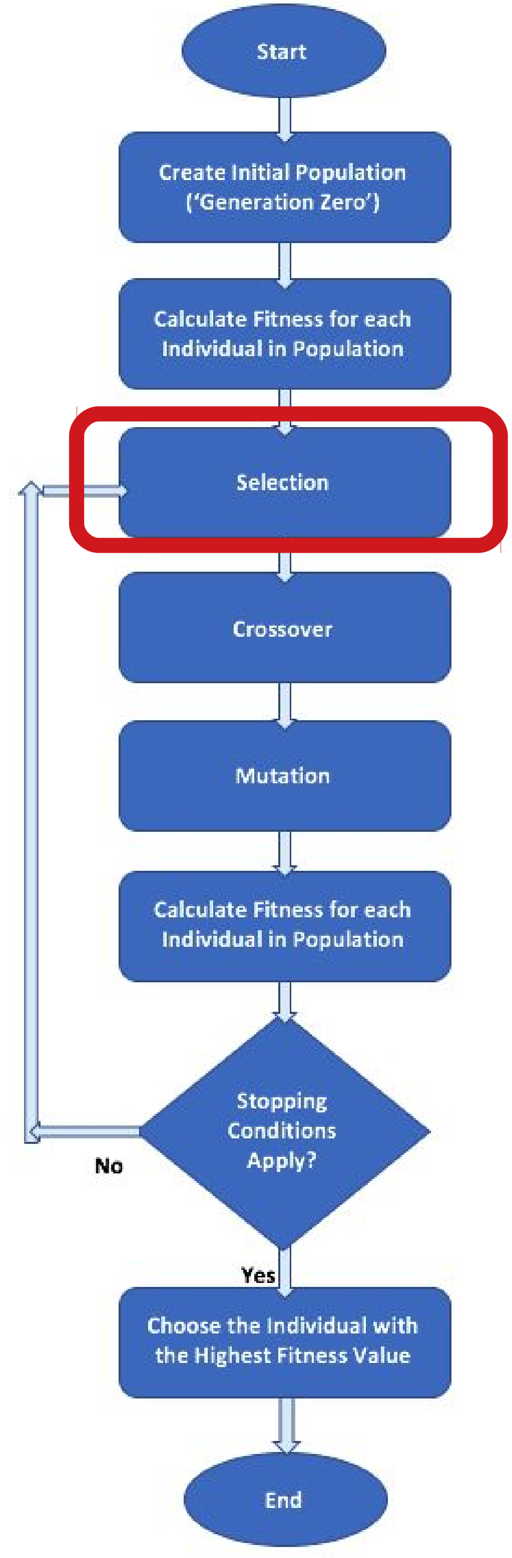


EA — Workflow — Selection

Assigns variable **probabilities** of individuals acting as parents depending on their fitnesses.

Usually **probabilistic:**
higher quality solutions **more** likely to become parents than **lower** quality, but ... not guaranteed.
Even **worst** in current population usually has **non-zero probability** of becoming a parent.

This **stochastic** nature
can **aid** escape from **local optima**!



EA — Workflow — Variation Operators

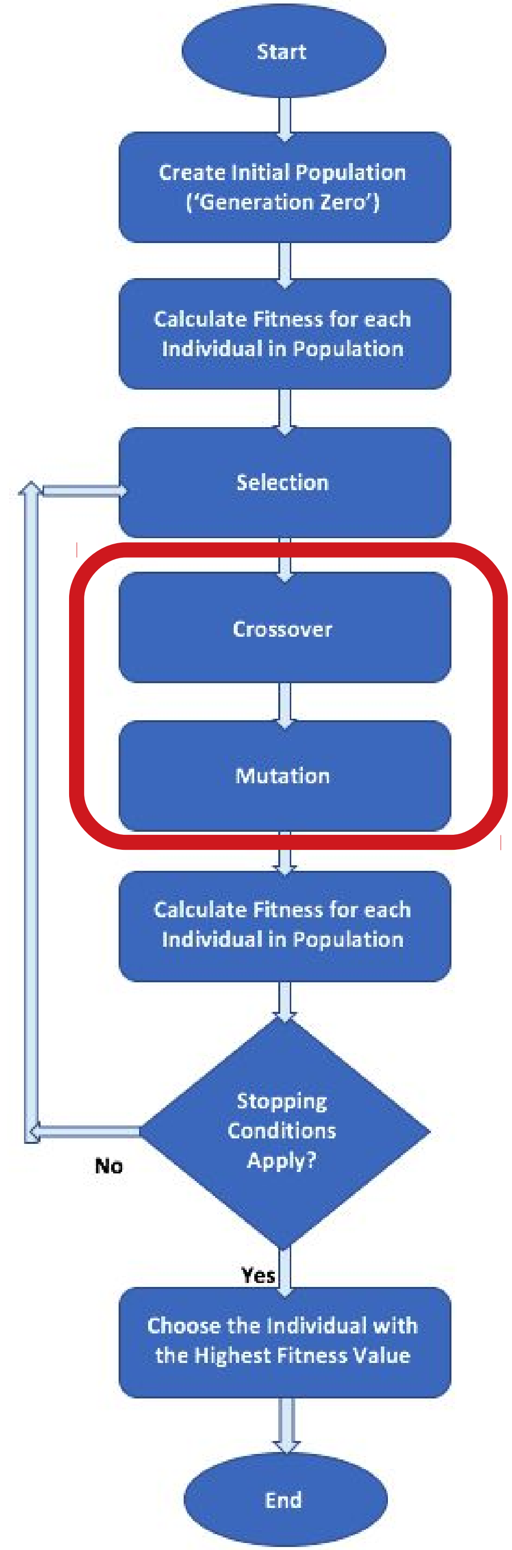
The **main aim** is
to **generate** new **candidate solutions**.

Usually divided into types according to their ***arity*** (number of inputs):

- arity = 1 -> mutation operators
- arity > 1 -> recombination operators
- arity = 2 -> crossover operators

The relative importance of recombination and mutation is debated intensively now, but most EAs use both of them.

Choice of particular variation operators is representation dependant.



Workflow - Variation Operators - Crossover

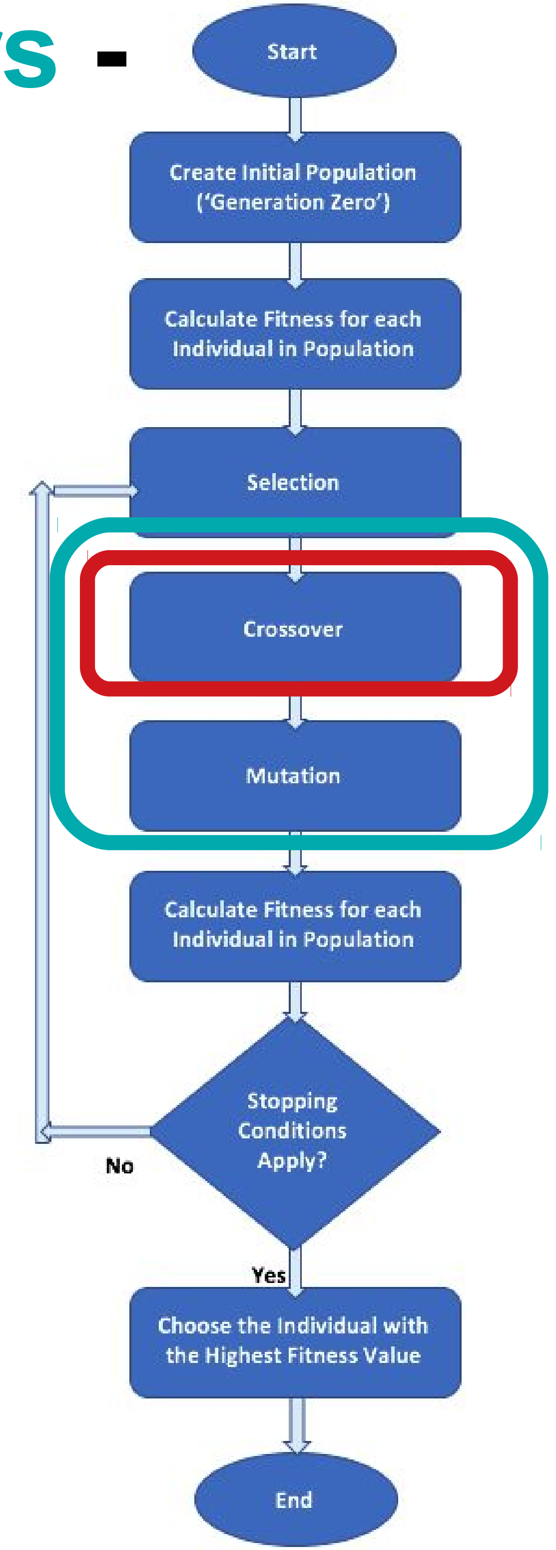
Crossover or **Recombination**

Merges information from **parents** into **offspring**.

Choice of what information to merge is **stochastic**.

Most **offspring** may be **worse** or the **same** as the **parents**.
Hypothesis: some can be **better** by **combining elements** of genotypes that **lead to good** traits.

Metaphor from nature:
it has been **successfully used** by breeders of plants and livestock!



Workflow - Variation Operators - Mutation

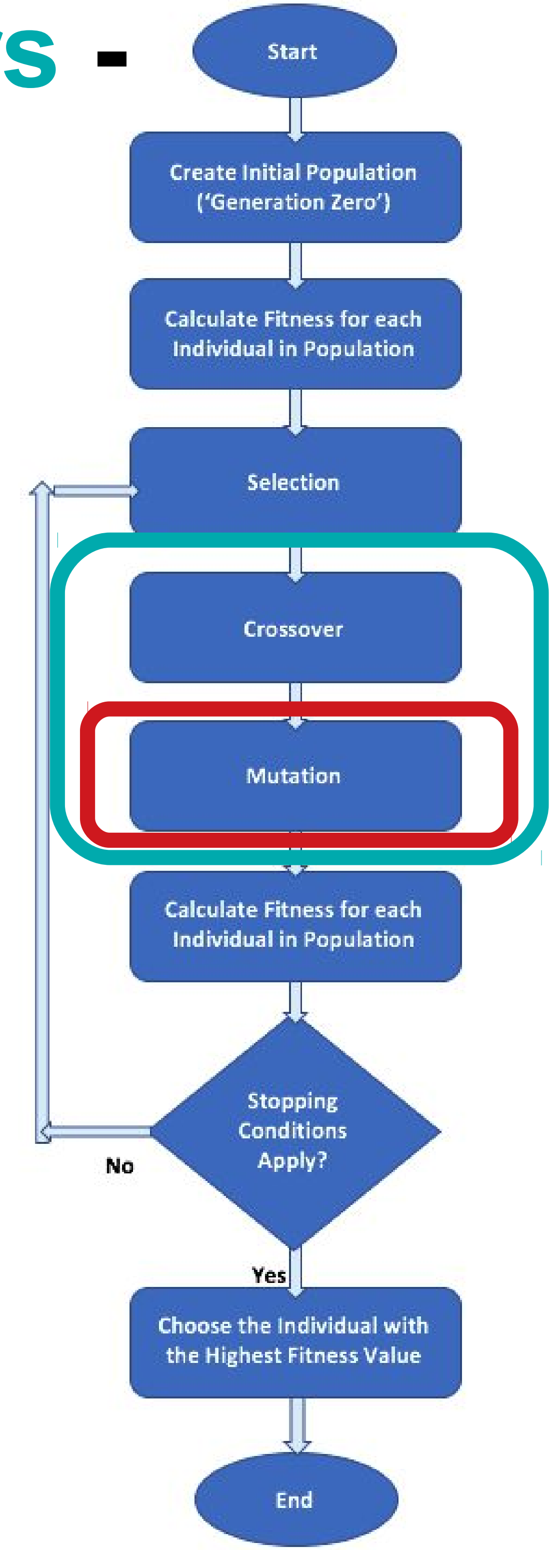
Operates on one genotype and **delivers another**.

Element of **randomness** is essential and differentiates it from other unary heuristic operators.

It depends on representation and dialect:

- **Binary GAs** – background operator responsible for preserving and introducing diversity,
- **EP** for FSM's/ continuous variables – only search operator,
- **GP** – hardly used.

May **guarantee** connectedness of search space and hence **convergence** proofs.



EA — Workflow — Start/Stop

Start

Initialization usually done at **random.**

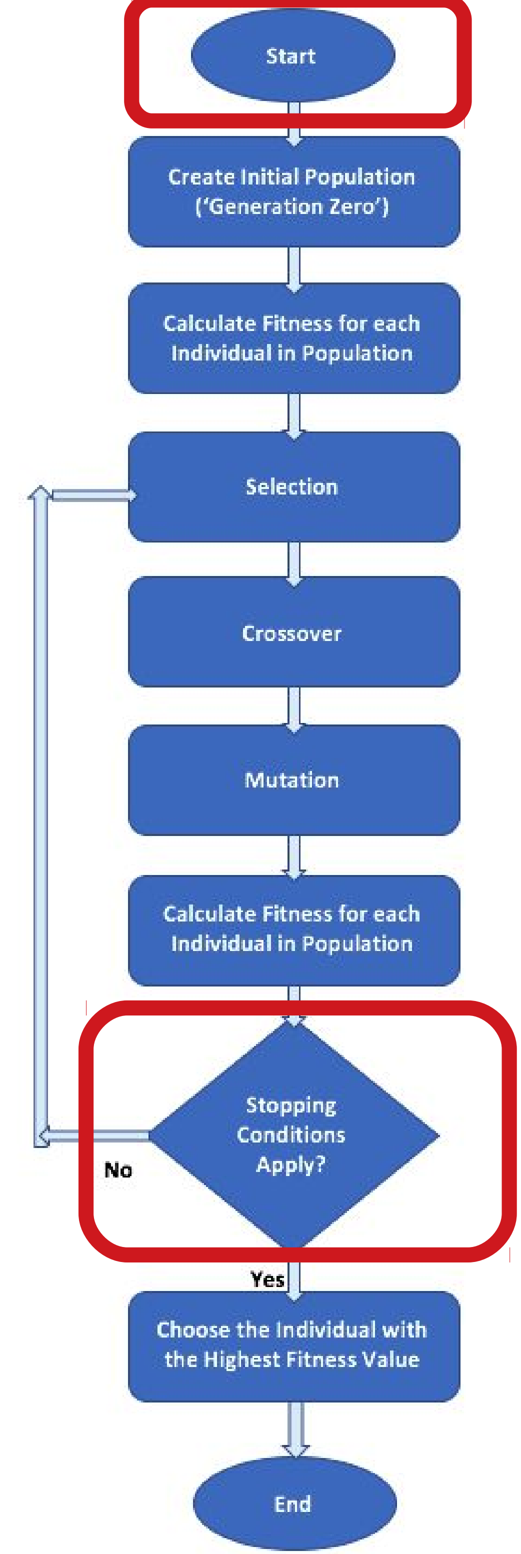
It should be even spread and mixture of possible allele values.

It can include **existing** solutions, or use problem-specific **heuristics**, to "**seed**" the population (care should be taken!)

Stop

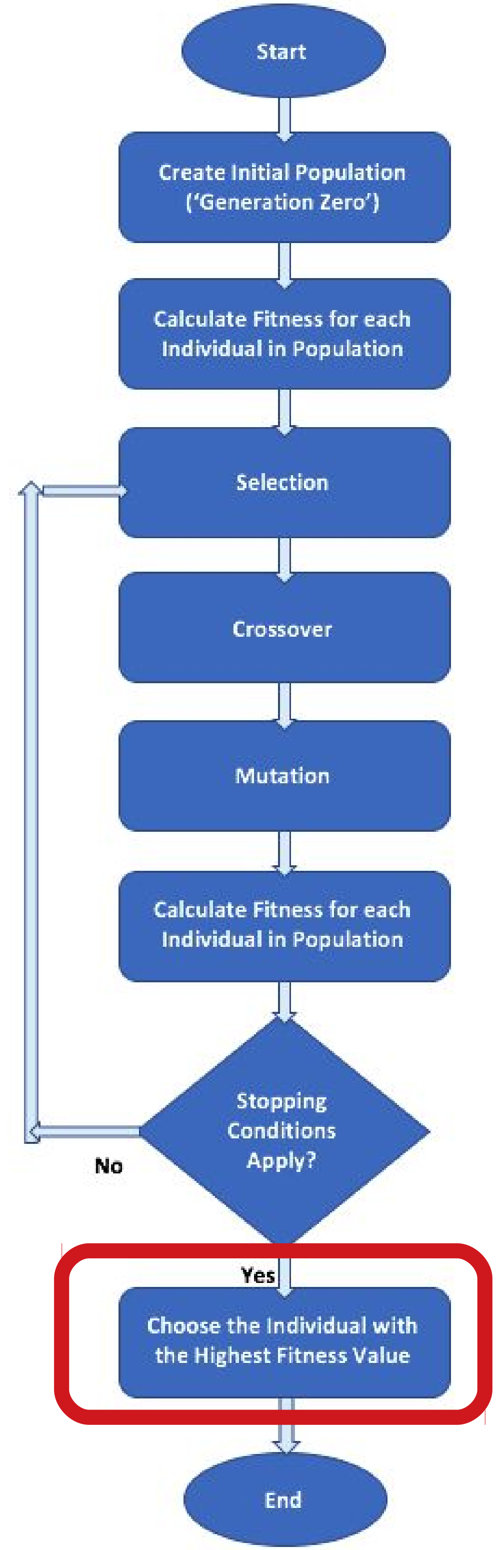
Termination condition checked every generation:

- some **planned** (known/assumed) **fitness**,
- some maximum allowed **number of generations**,
- some **minimum** level of **diversity**,
- some specified **number of generations without** fitness **improvement**.



EA — Workflow — End

Choose
the **individual**
with
the **highest fitness** value.

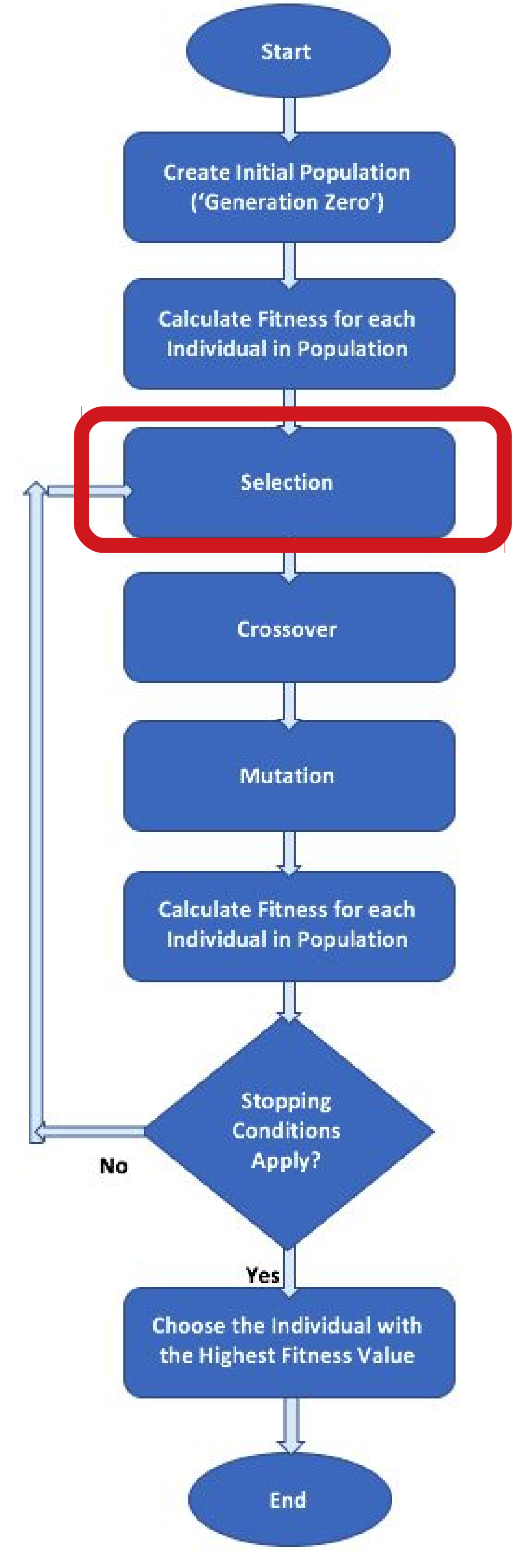


Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- **Selection — in details now**
- Crossover
- Mutation
- Real-coded EA

EA — Workflow — Selection Methods

- Roulette wheel selection (fitness proportionate selection — FPS)
- Stochastic universal sampling (SUS)
- Rank-based selection
- Tournament selection



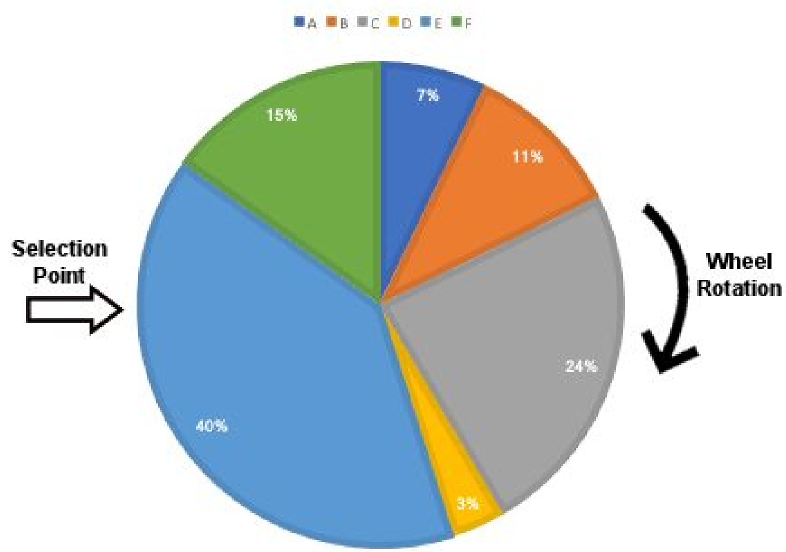
Workflow —

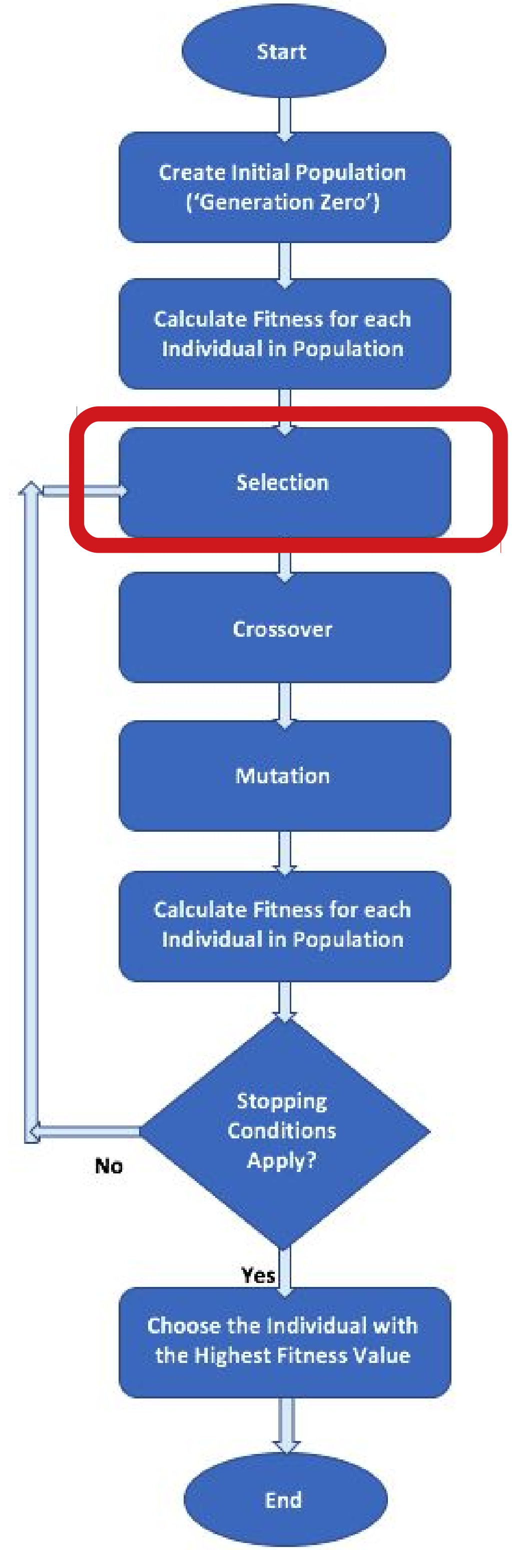
Roulette Wheel Selection

Probability for selecting an individual is directly **proportionate** to its **fitness value**.

This is comparable to using a roulette wheel in a casino and assigning each individual a portion of the wheel proportional to its fitness value

Individual	Fitness	Relative portion
A	8	7%
B	12	11%
C	27	24%
D	4	3%
E	45	40%
F	17	15%

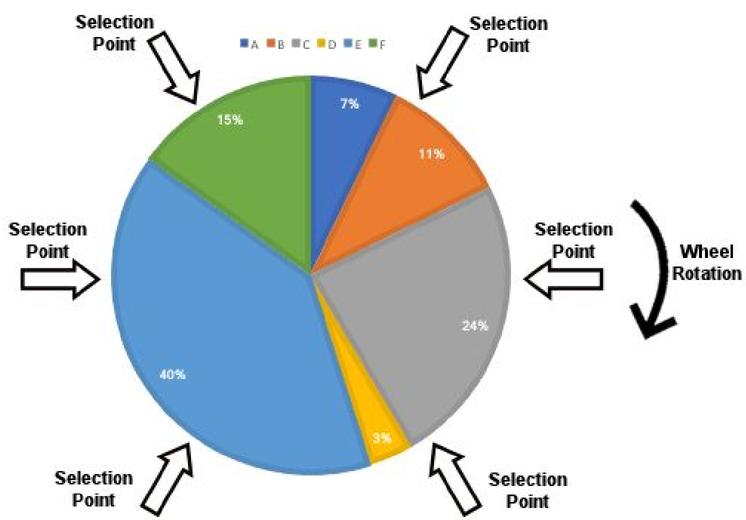


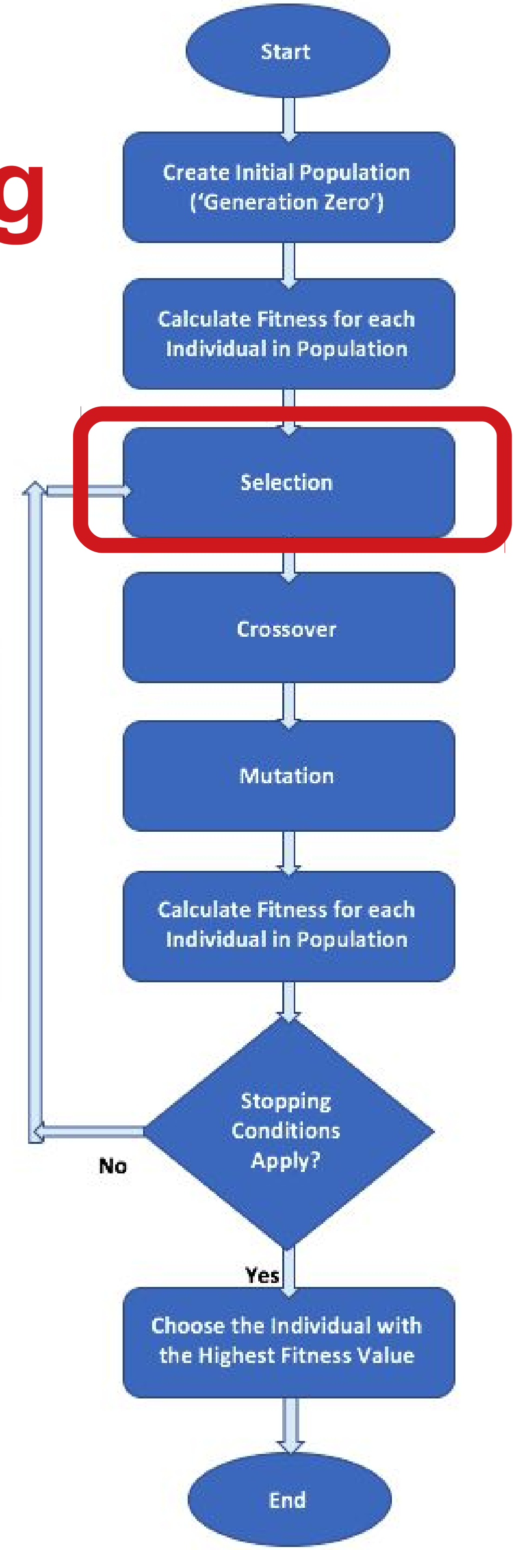


Workflow —
Stochastic Universal Sampling

Instead of a single selection point and turning the roulette wheel *N* times until all needed *N* individuals have been selected, we **turn** the wheel **only 1 time** and use ***N* selection points** that are equally spaced around the wheel

Individual	Fitness	Relative portion
A	8	7%
B	12	11%
C	27	24%
D	4	3%
E	45	40%
F	17	15%

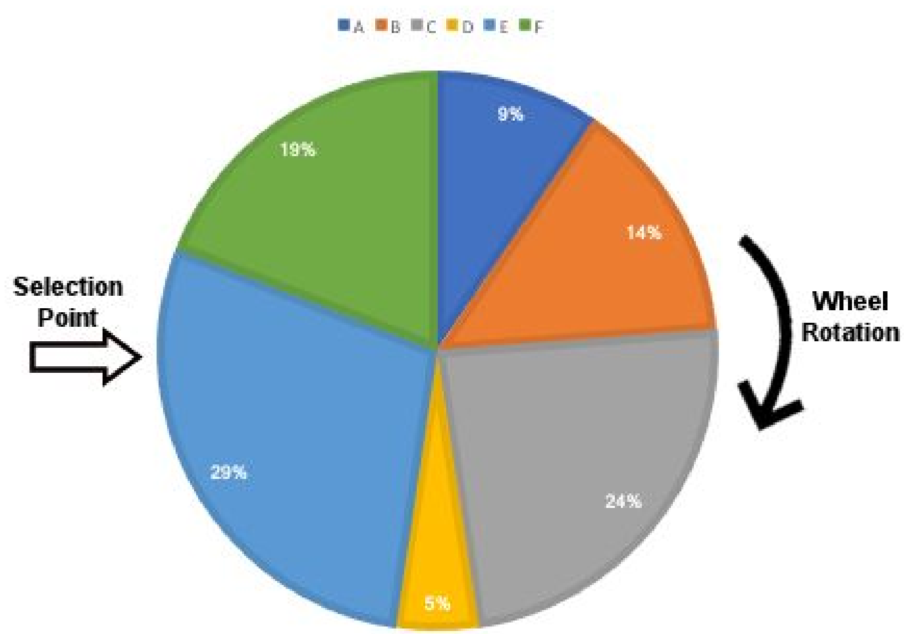


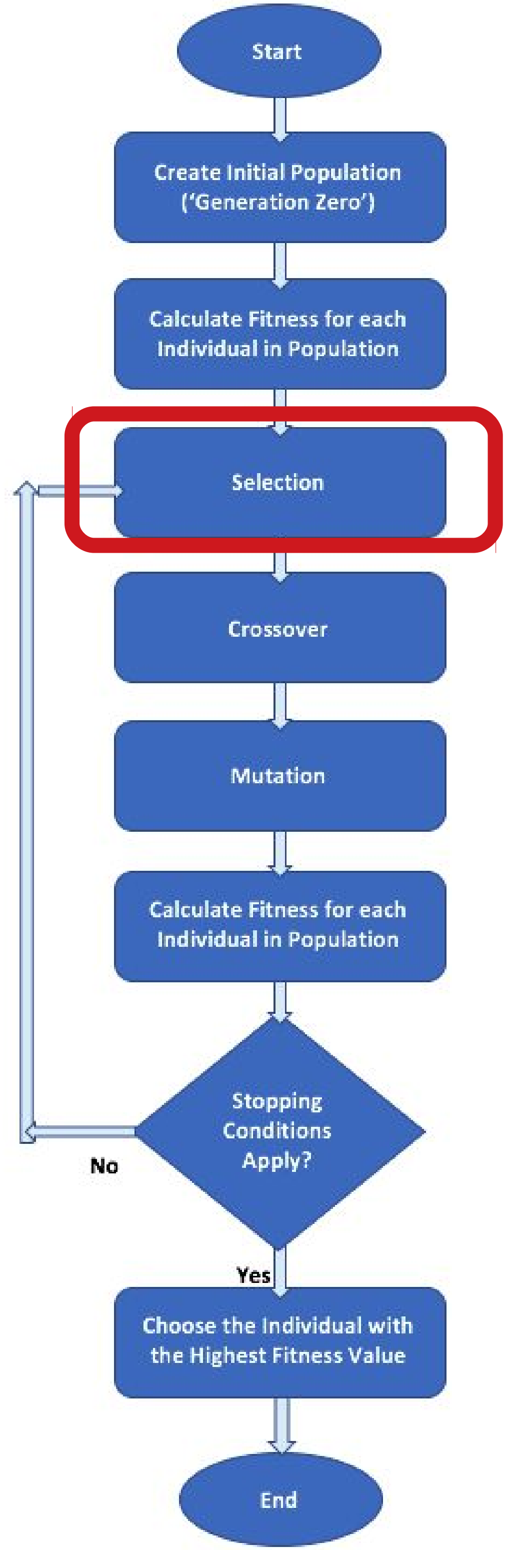


Workflow —
Rank-based Selection

The **fitness** is used **to sort** the individuals: each individual is given a **rank** for its **position** and **wheel-portion**, and the roulette probabilities are calculated based on these ranks.

Individual	Fitness	Rank	Relative portion
A	8	2	9%
B	12	3	14%
C	27	5	24%
D	4	1	5%
E	45	6	29%
F	17	4	19%



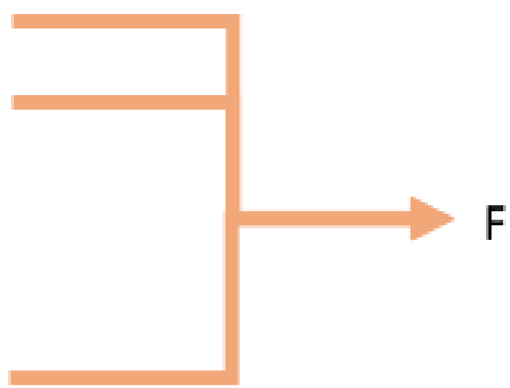


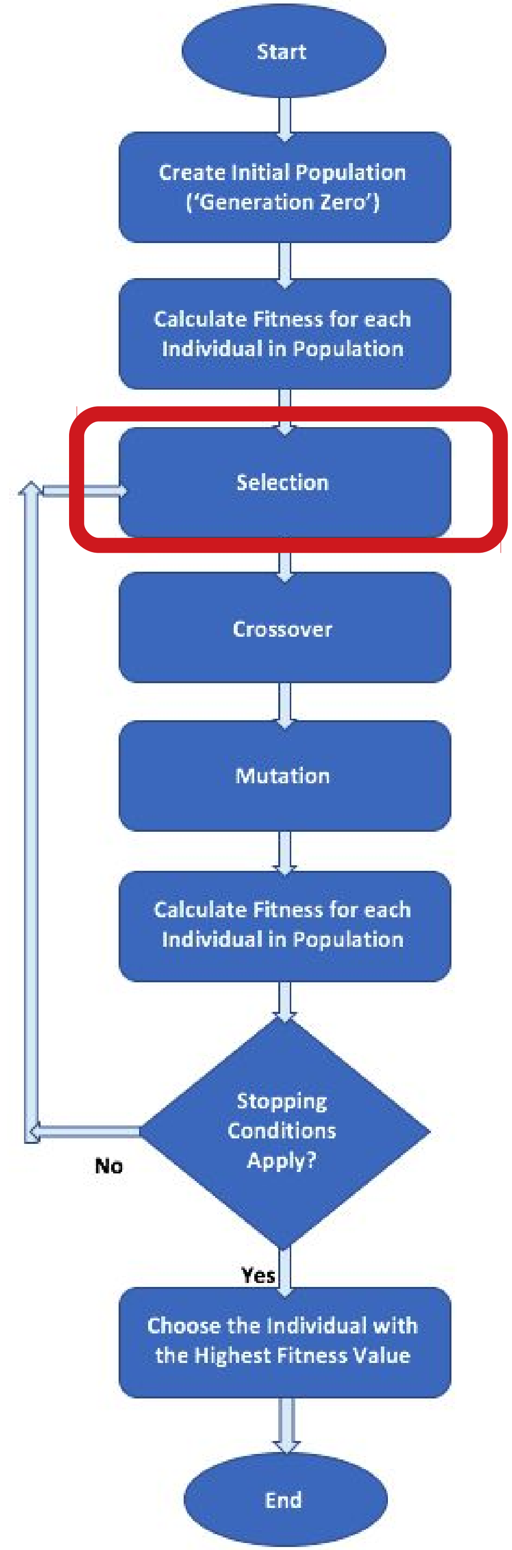
Workflow — Tournament Selection

In each round of the tournament selection method, two or more **individuals** are randomly **picked** from the population, and the **one** with the **highest** fitness score **wins** and gets **selected**.

The number of individuals participating at each tournament selection round (three in this figure) is suitably called ***tournament size***. The **larger** the tournament size, the **higher** the chances that the **best** individuals will **be selected**.

Individual	Fitness
A	8
B	12
C	27
D	4
E	45
F	17





Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- **Crossover — in details now**
- Mutation
- Real-coded EA

Workflow - Variation Operators - Crossover

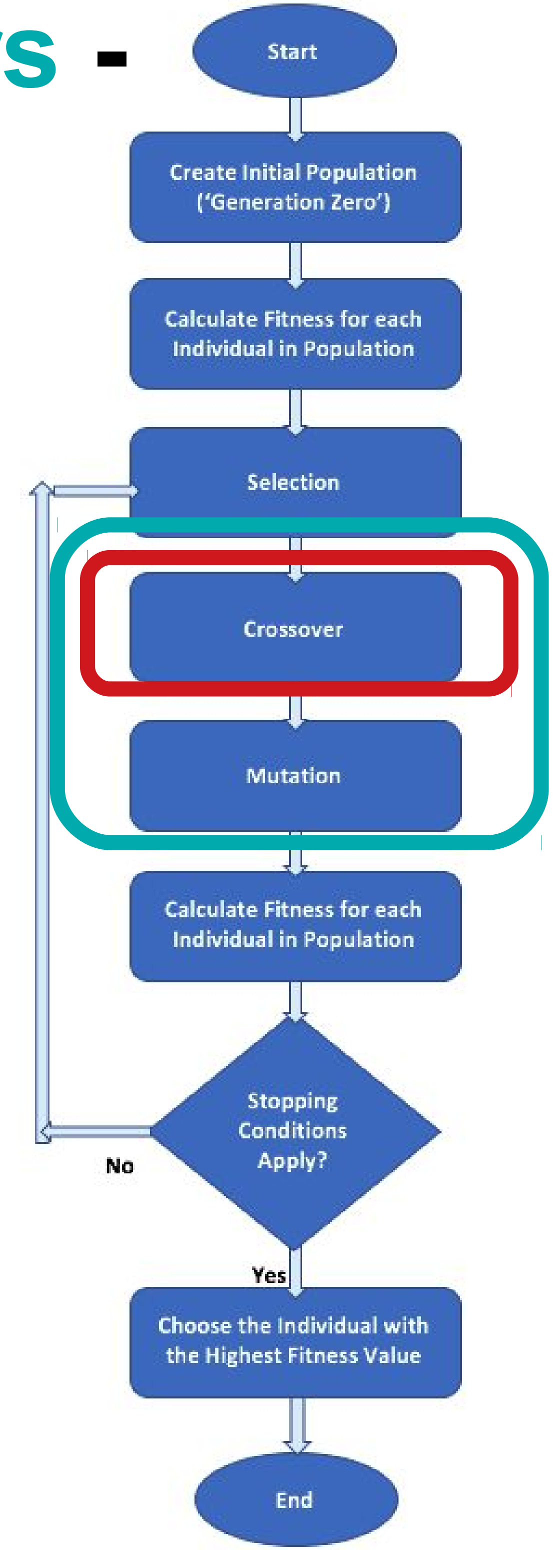
Crossover or **Recombination**

Merges information from **parents** into **offspring**.

Choice of what information to merge is **stochastic**.

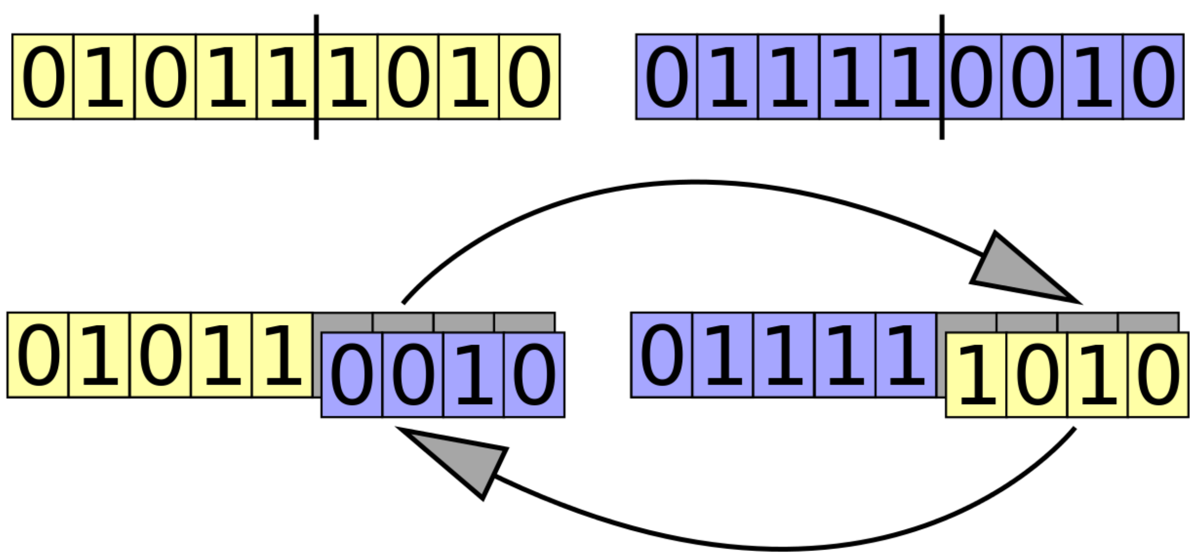
Most **offspring** may be **worse** or the **same** as the **parents**.
Hypothesis: some can be **better** by **combining elements** of genotypes that **lead to good** traits.

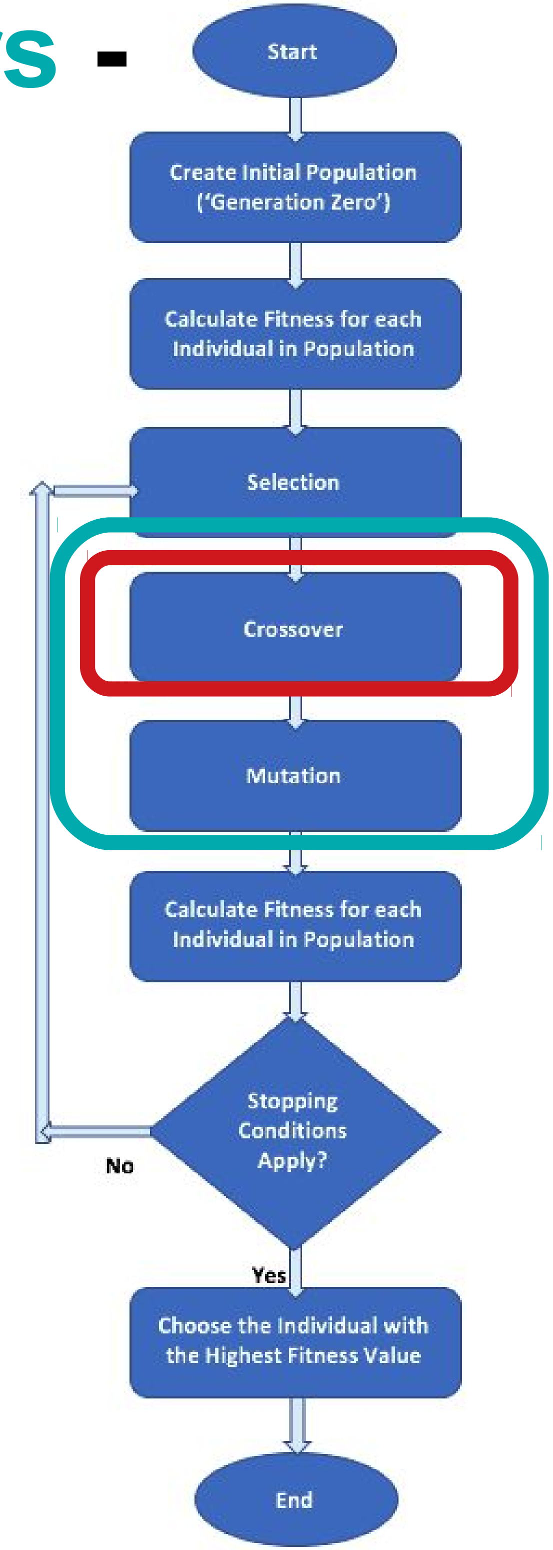
Metaphor from nature:
it has been **successfully used** by breeders of plants and livestock!



Workflow - Variation Operators - Crossover - Sinlge-point

The **crossover point** (or **cut point**) on the chromosomes of both parents is **selected randomly**.Genes to the right of that point are swapped between the two parent chromosomes. As a result, we **get two offsprings**, where each of them **carry** some genetic information **from both** parents.

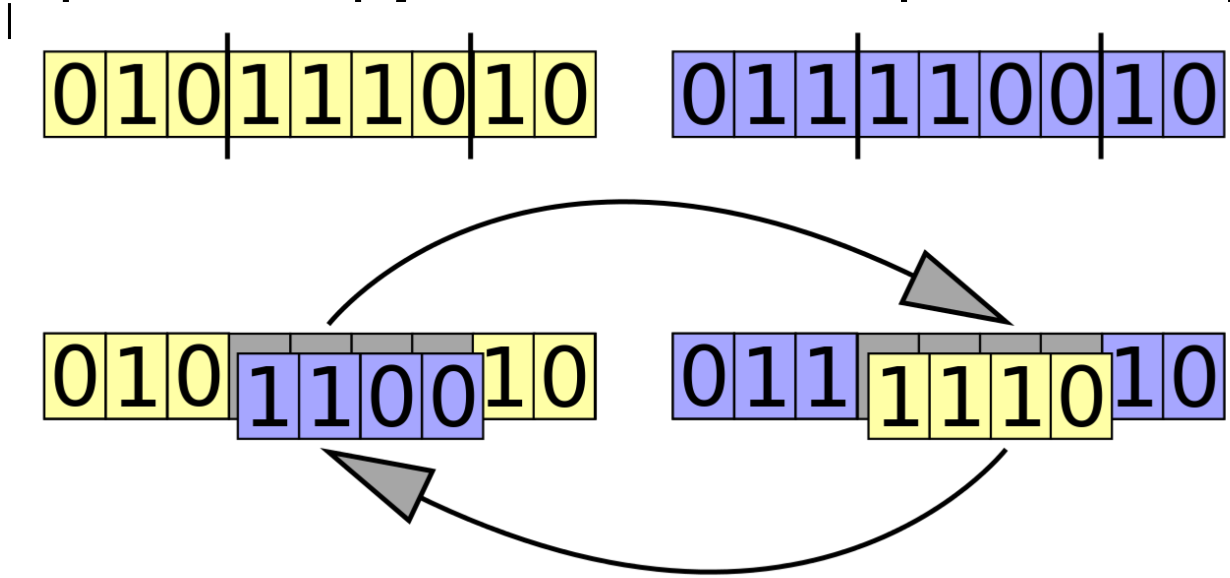


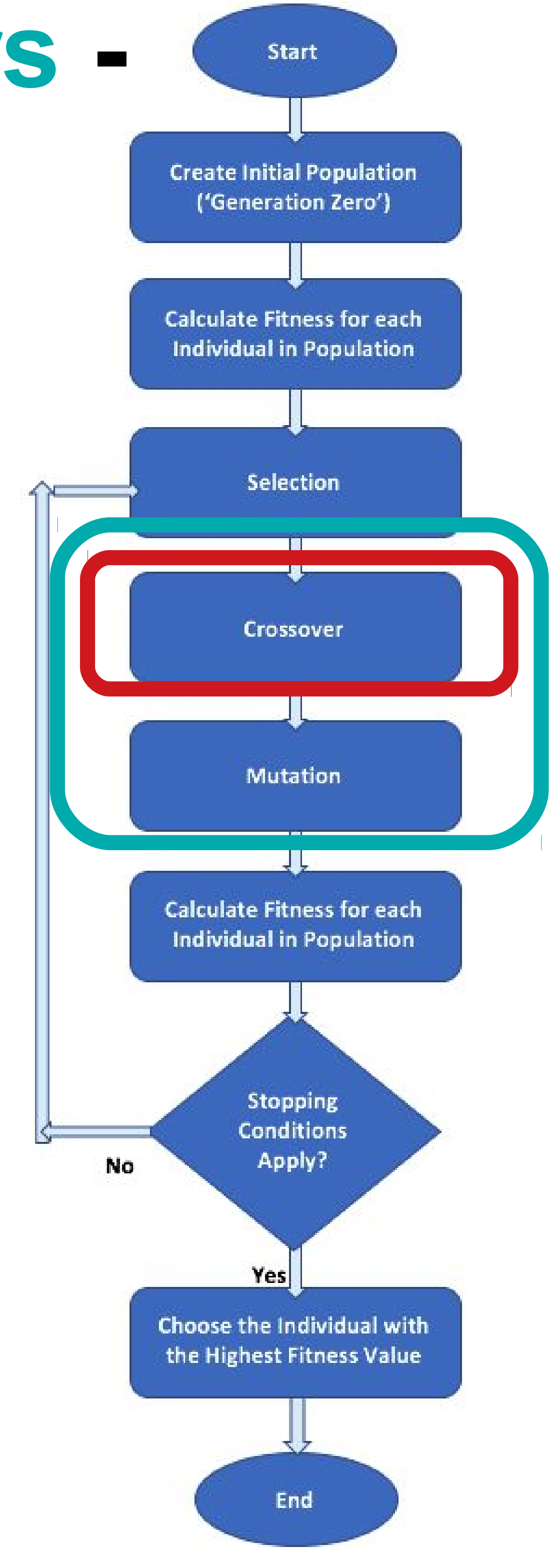


Workflow - Variation Operators - Crossover - *K*-point

For example, in 2-point crossover 2 points on the chromosomes of both parents are selected randomly. The genes residing between these points are swapped between the two parent chromosomes.

A generalization of this method is the **k-point crossover**, where ***k*** represents a positive





Workflow - Variation Operators - Crossover - Uniform

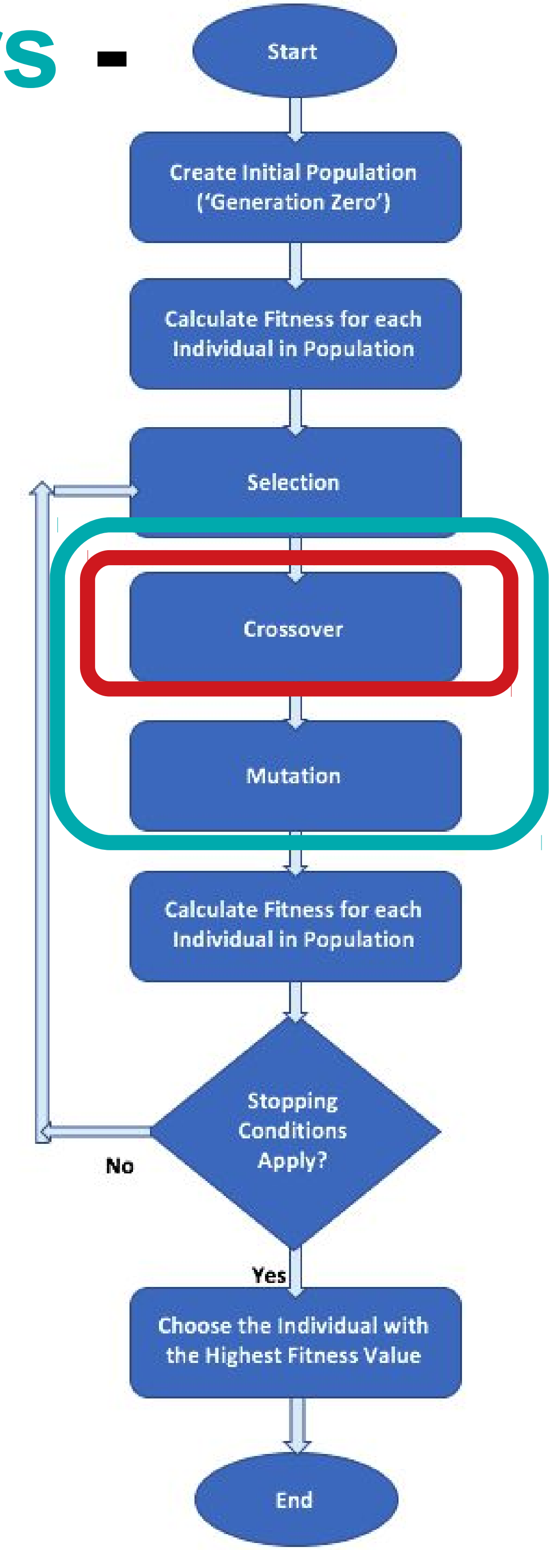
Each gene is **independently determined** by **randomly** choosing one of the parents.
If the random distribution is 50%, each parent has the same chance of influencing the offspring.
NOTE: Below, **integer-based** chromosomes are shown, but it is **the same for binary** ones.

5	7	2	3	1	6	9	8	0

6	8	3	4	2	1	0	9	7

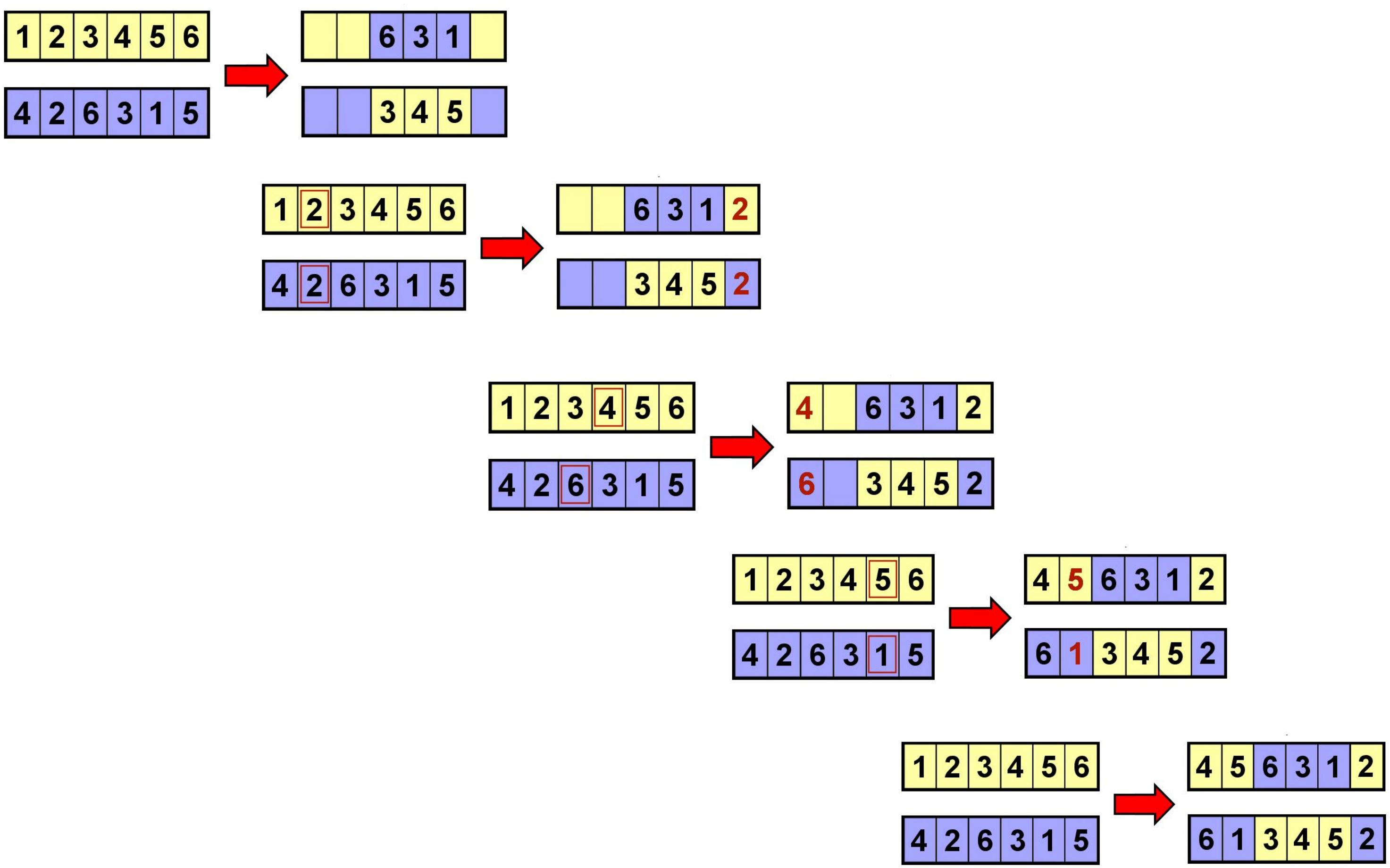
5	8	2	3	2	6	0	9	0

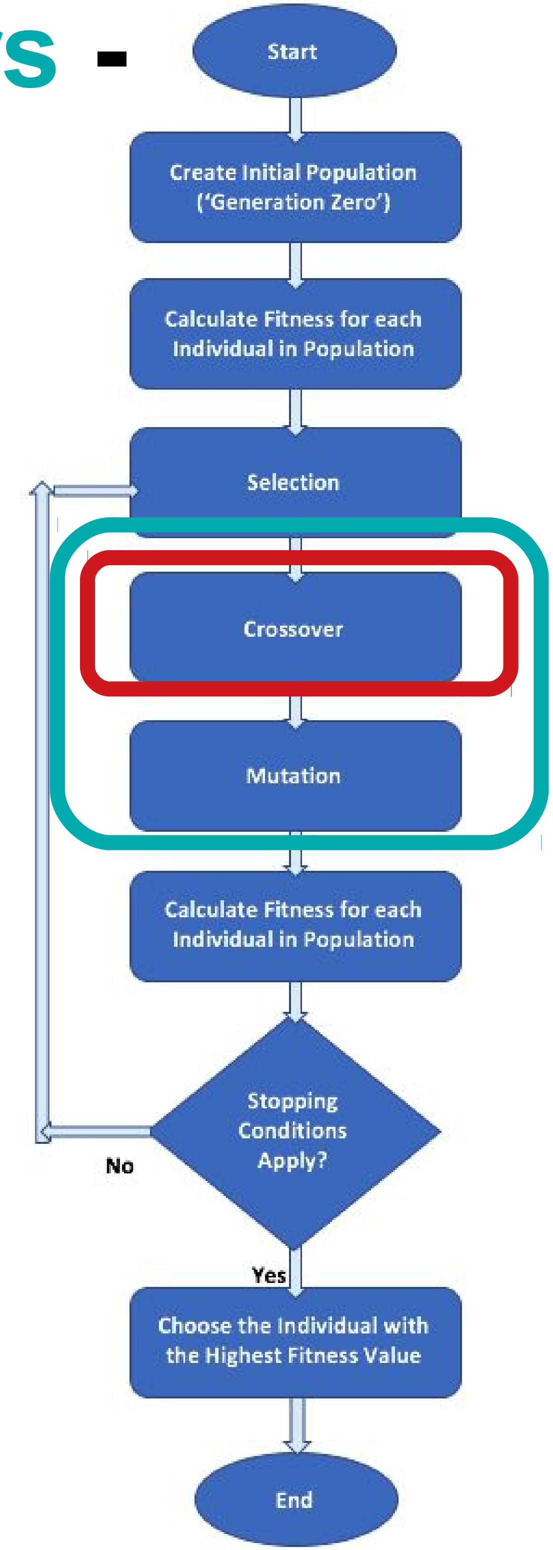
6	7	3	4	1	1	9	8	7



Workflow - Variation Operators - Crossover — Ordered Lists

The **ordered crossover** (OX1) method strives to preserve the relative ordering of the parent's genes as much as possible.





Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- **Mutation — in details now**
- Real-coded EA

Workflow - Variation Operators - Mutation

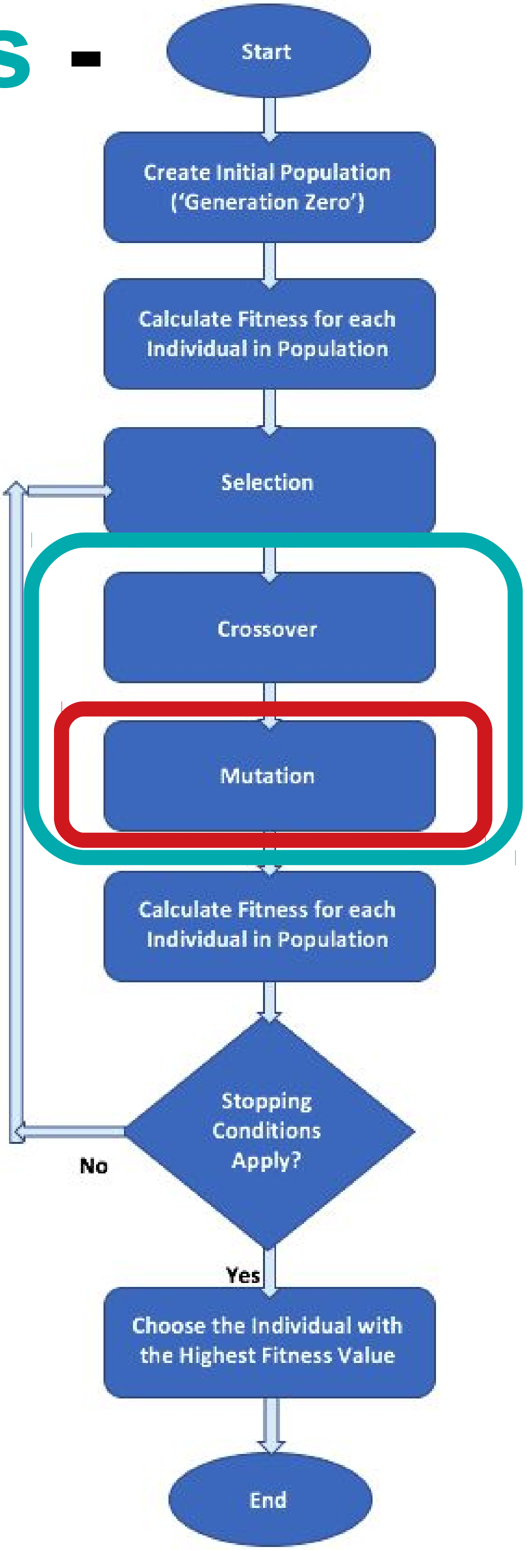
Operates on one genotype and **delivers another**.

Element of **randomness** is essential and differentiates it from other unary heuristic operators.

It depends on representation and dialect:

- **Binary GAs** – background operator responsible for preserving and introducing diversity,
- **EP** for FSM's/ continuous variables – only search operator,
- **GP** – hardly used.

May **guarantee** connectedness of search space and hence **convergence** proofs.

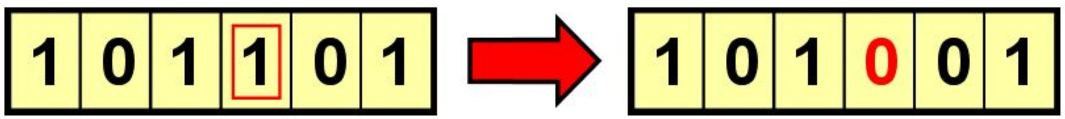


Workflow — Mutation - Flip bit

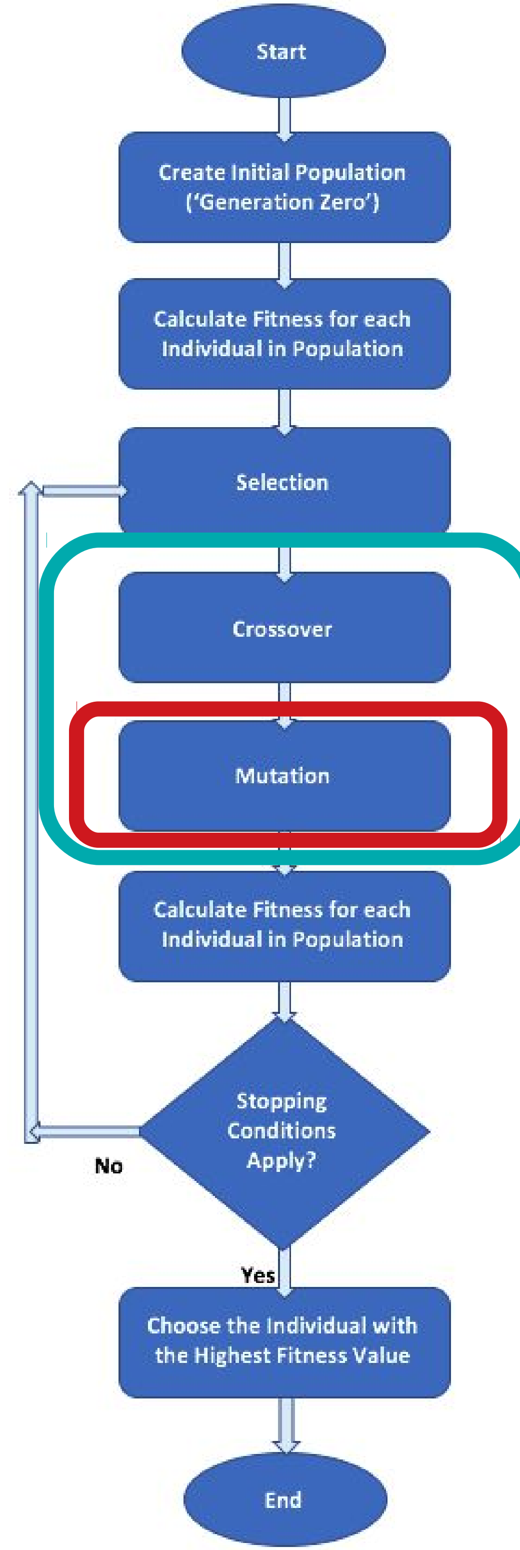
For a binary chromosome,

1 gene is **randomly**

selected and its value is **flipped** (complemented).

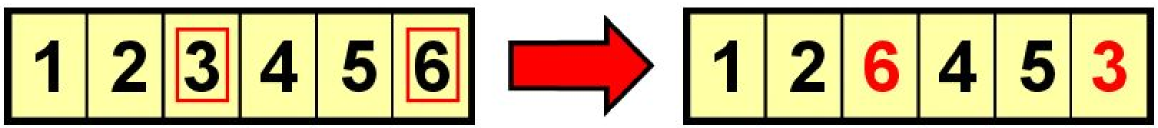


This can be extended to several random genes being flipped instead of just one.

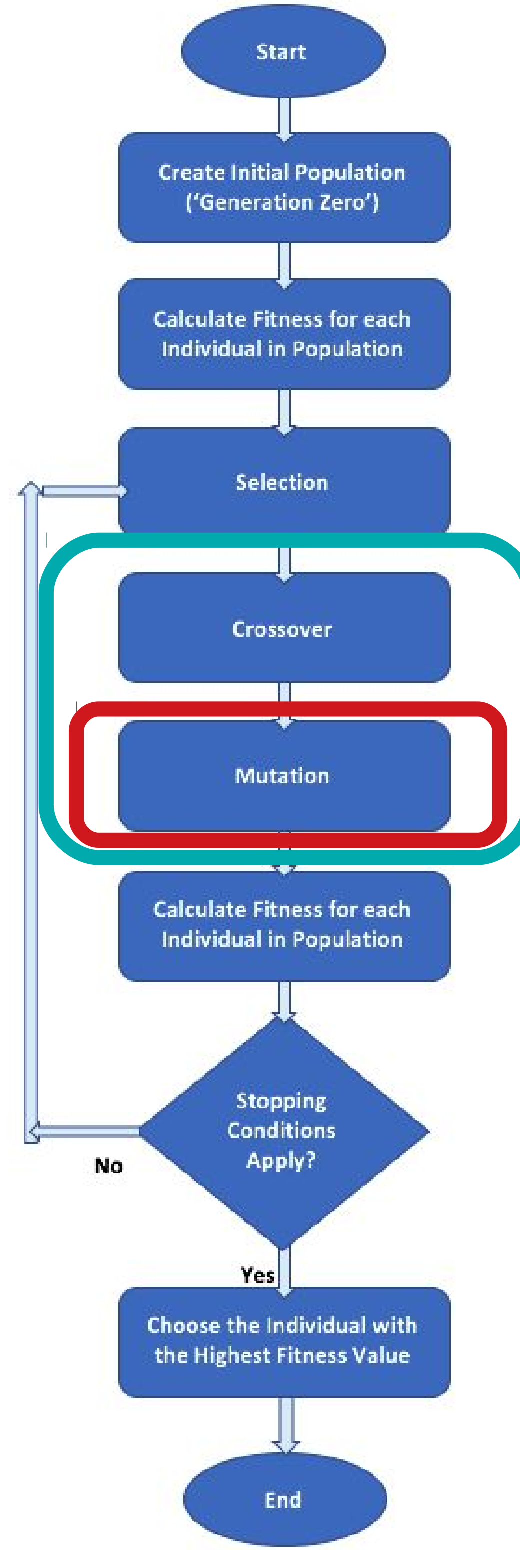


Workflow — Mutation - Swap

For a binary or integer-based chromosomes,
2 genes are **randomly selected**
and their values are **swapped**.

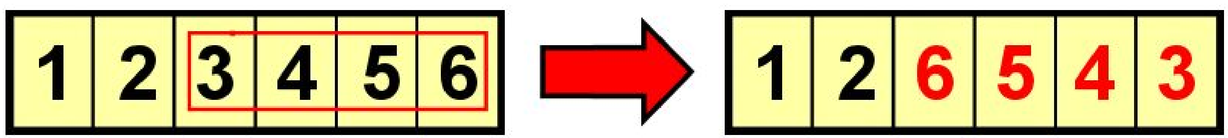


This mutation operation is **suitable**
for the **chromosomes of ordered lists**,
as the **new chromosome**
still **carries** the **same genes** as the original one.

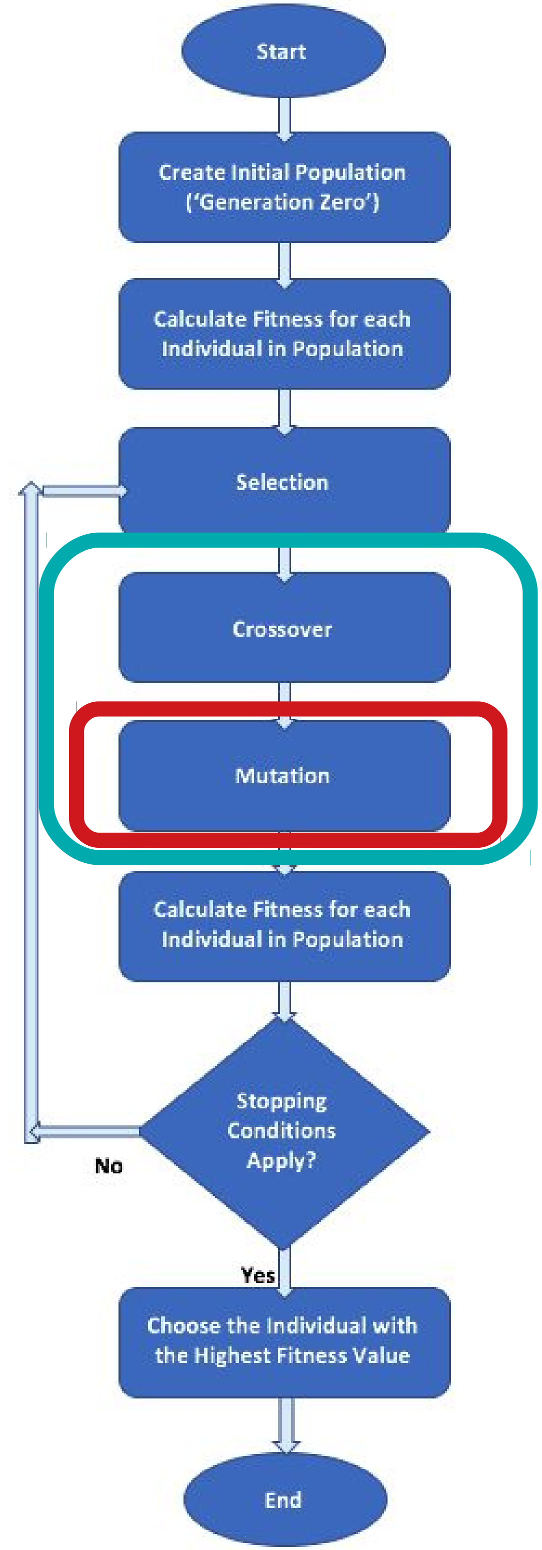


Workflow — Mutation - Inversion

For a **binary** or **integer-based** chromosomes, a **random sequence** of genes is selected and the **order** of the genes in that sequence is **reversed**.

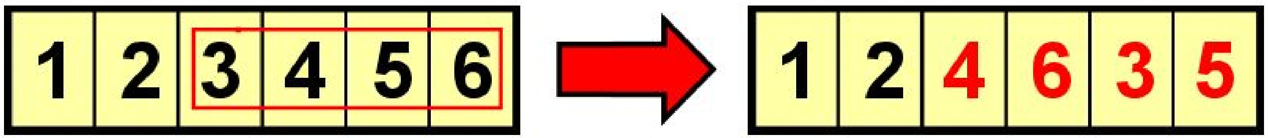


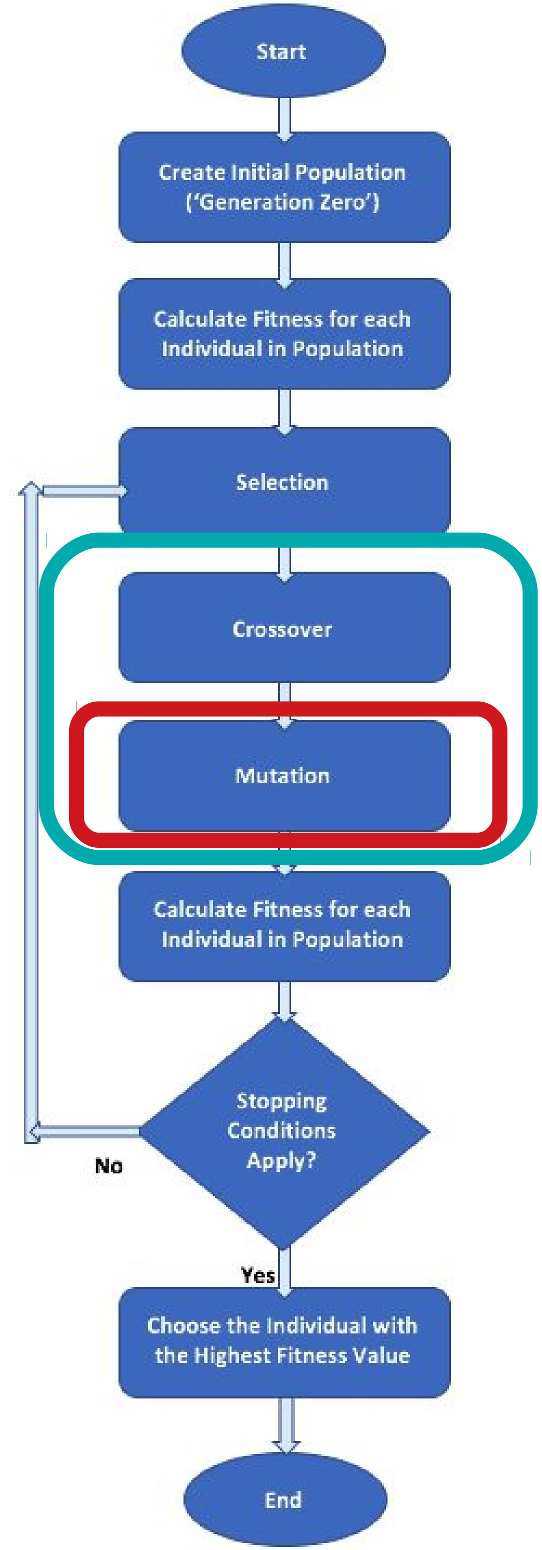
Similar to the **swap mutation**, the **inversion mutation** operation is **suitable** for the chromosomes of **ordered lists**.



Workflow — Mutation - Scramble

For a **binary** or **integer-based** chromosomes, a **random sequence** of genes is selected and and the **order** of the genes in that sequence is **shuffled** (or **scrambled**).

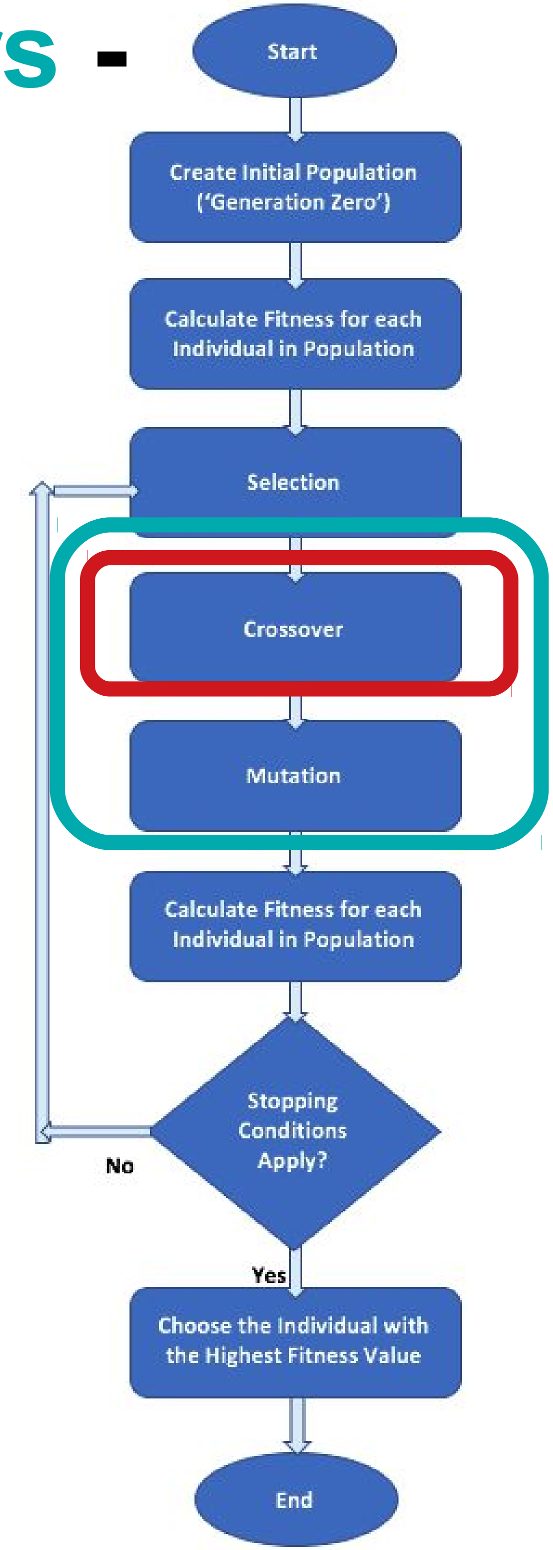




Content

- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- **Real-coded EA**

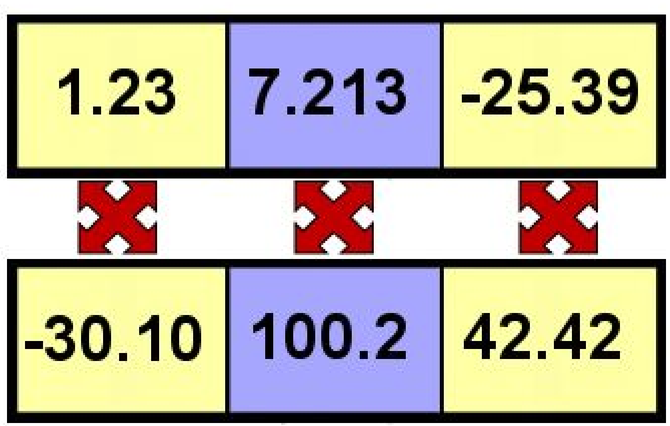
Workflow - Variation Operators - Real-coded



The **selection** methods will work just the same **as they only depend on the fitness** of the individuals and not their representation.

But the **crossover** and **mutation** methods will **not be suitable** and so specialized ones need to be used.

They should be applied **separately for each dimension** of the array that forms the real-coded chromosome.

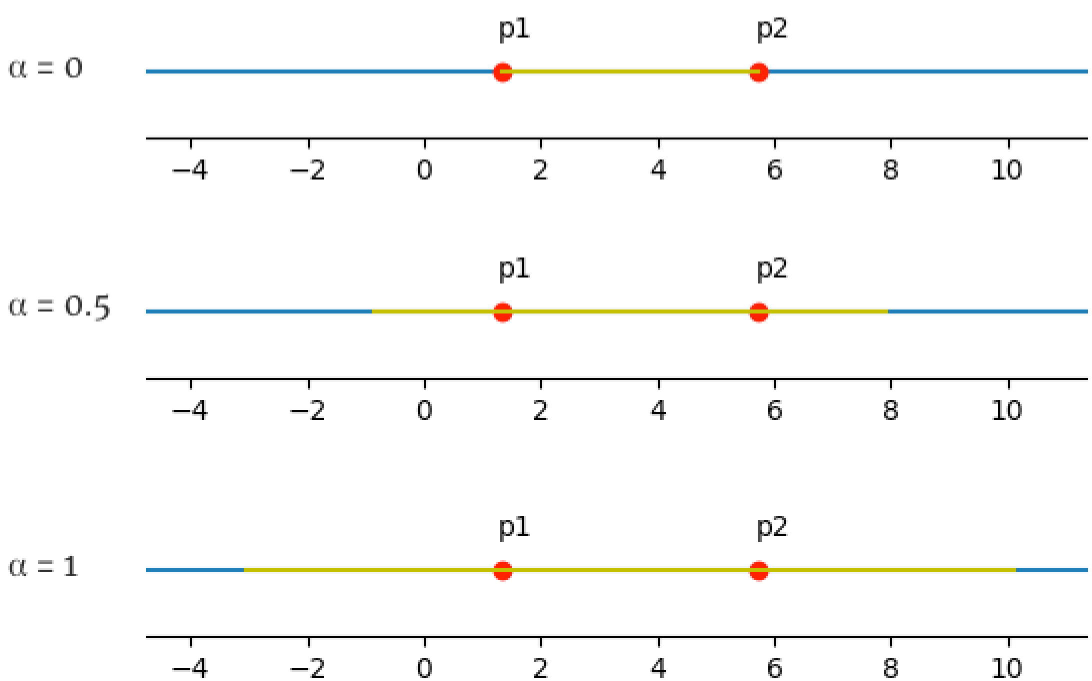


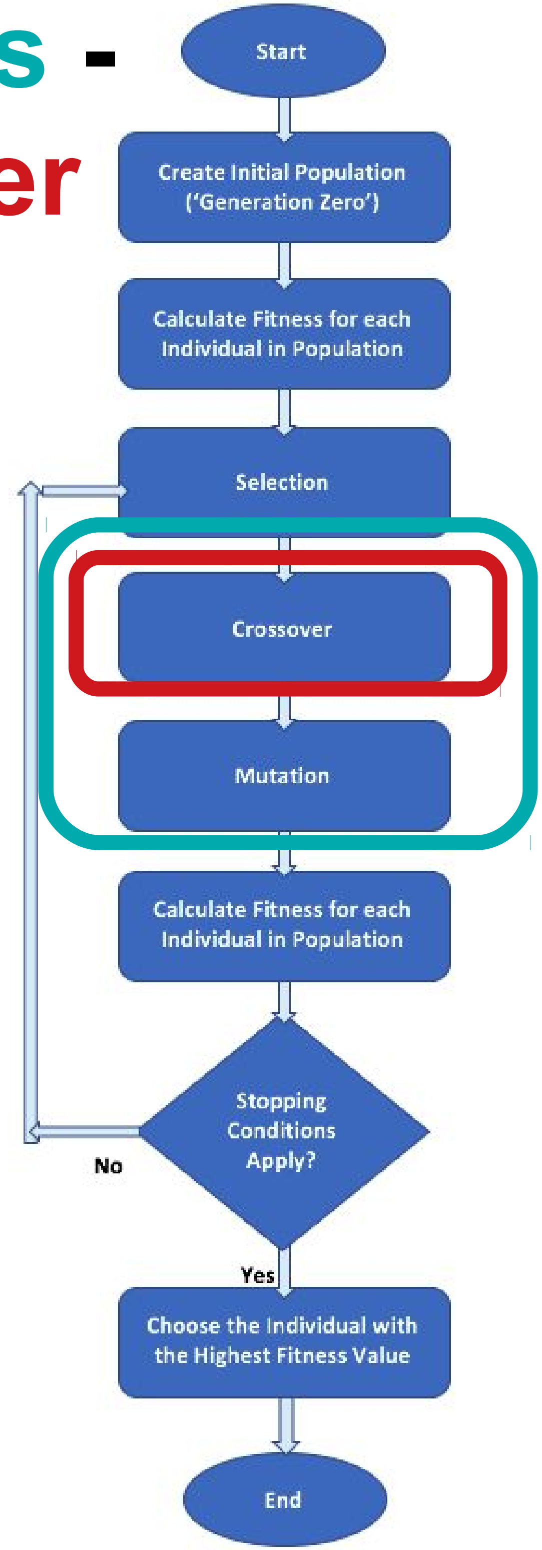
Workflow - Variation Operators - Real-coded — Blend Crossover

Blend crossover (**BLX**) - each **offspring** is randomly **selected** from the interval created by its parent values by some formulae:

$$[parent_1 - \alpha(parent_2 - parent_1),\ \ parent_2 + \alpha(parent_2 - parent_1)]$$

The parameter α is a constant, whose value lies between 0 and 1. With larger values of α, the



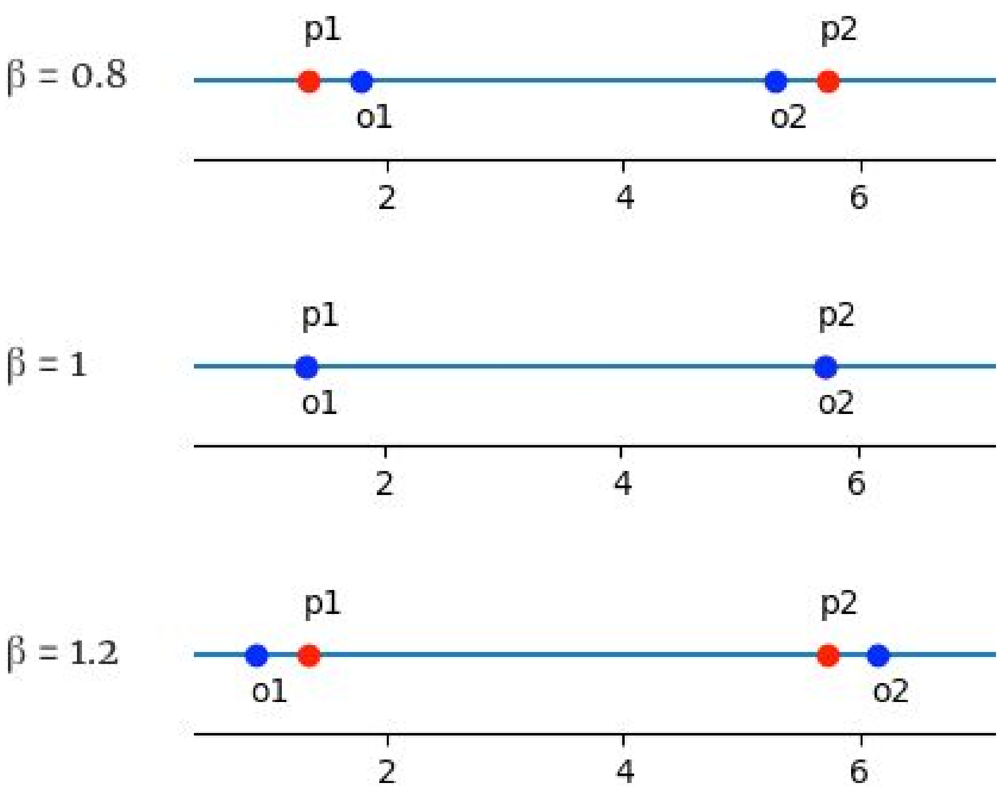


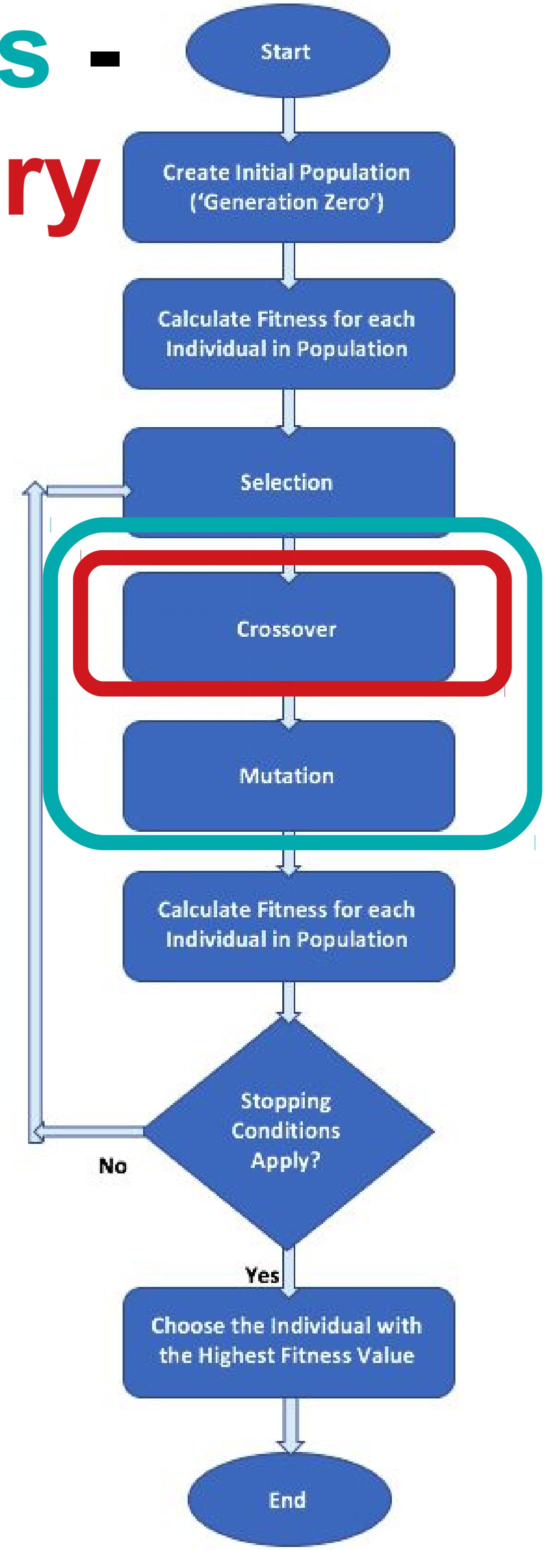
Workflow - Variation Operators - Real-coded — Simulated Binary

Simulated binary crossover (SBX) - each offspring is randomly selected from the interval created by its parent values by formula:

$$offspring_1 = \frac{1}{2}[(1+\beta)parent_1 + (1-\beta)parent_2]$$

$$offspring_2 = \frac{1}{2}[(1-\beta)parent_1 + (1+\beta)parent_2]$$





Workflow - Variation Operators - Real-coded — Simulated Binary

In the preceding cases, the **average** value of the two **offspring** is 3.525, which is **equal to** the **average** value of the two **parents**.

We need to preserve is the similarity between offspring and parents.

For this, the probability of **β** should be **much higher** for values **near 1**, where the offspring are **similar** to the parents.

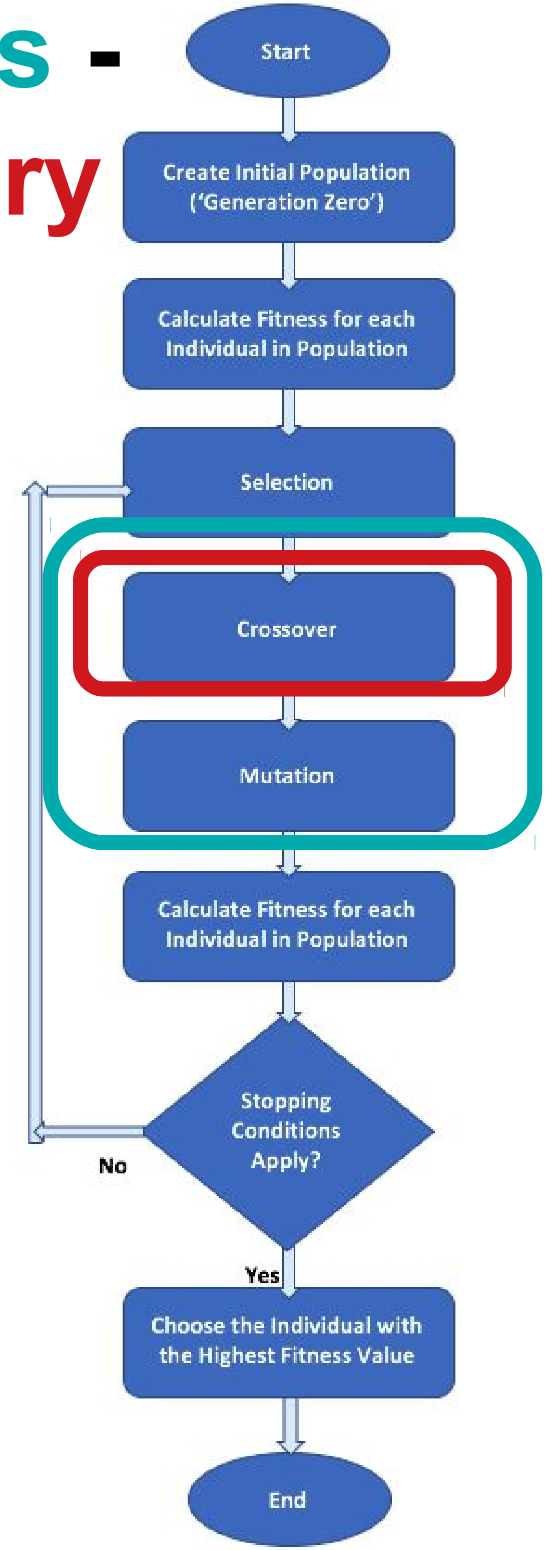
That is why, the β value is calculated using **another random** value, denoted by u, that is uniformly distributed over the interval [0, 1]:

u <=0.5

$$\beta = (2u)^{\frac{1}{\eta+1}}$$

u > 0.5

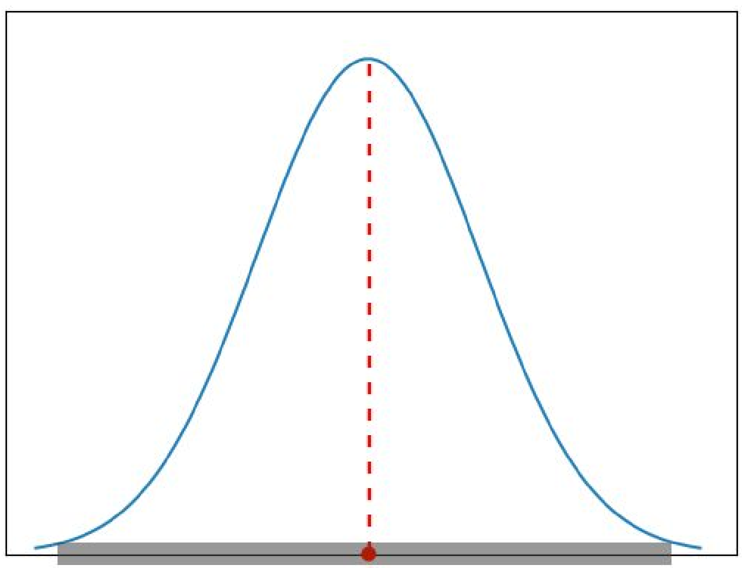
$$\beta = \left[\frac{1}{2(1-u)}\right]^{\frac{1}{\eta+1}}$$

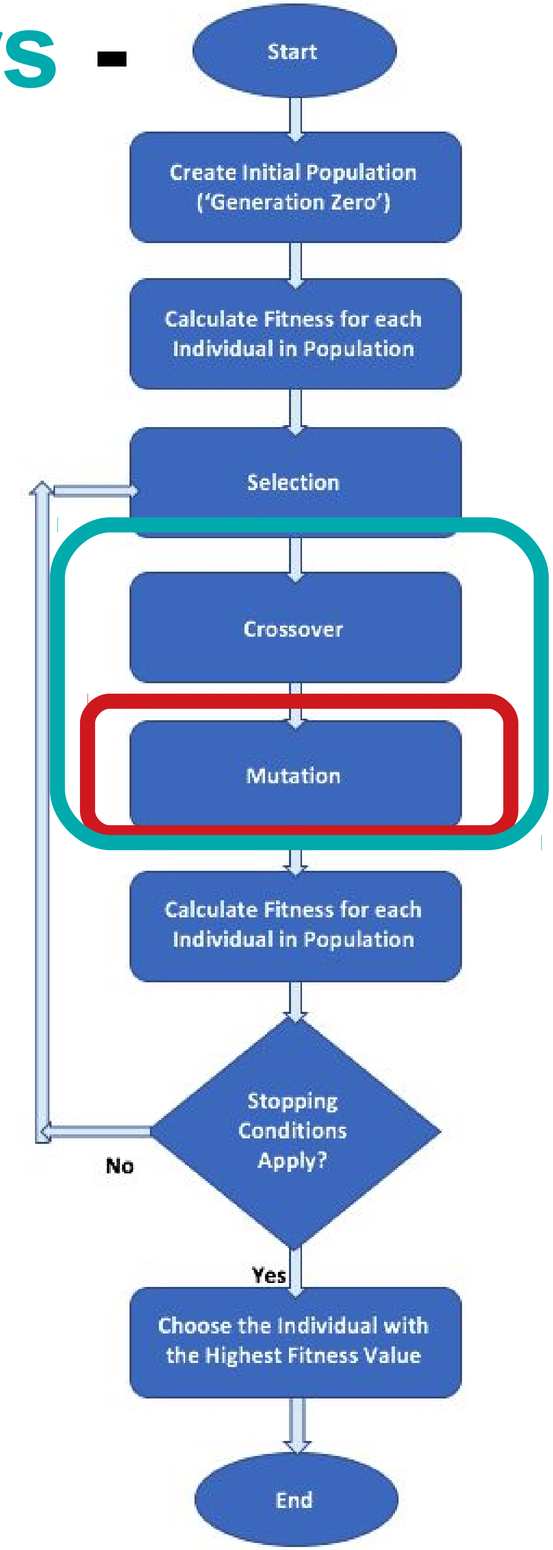


Workflow - Variation Operators - Real-coded — Real Mutation

Another approach is to **generate a random real** number that resides **in the vicinity of the original** individual.

Example: the **normally distributed** (or Gaussian) **mutation** -> a **random number** is generated using a **normal distribution** with a **mean = 0** and some **predetermined standard deviation**.

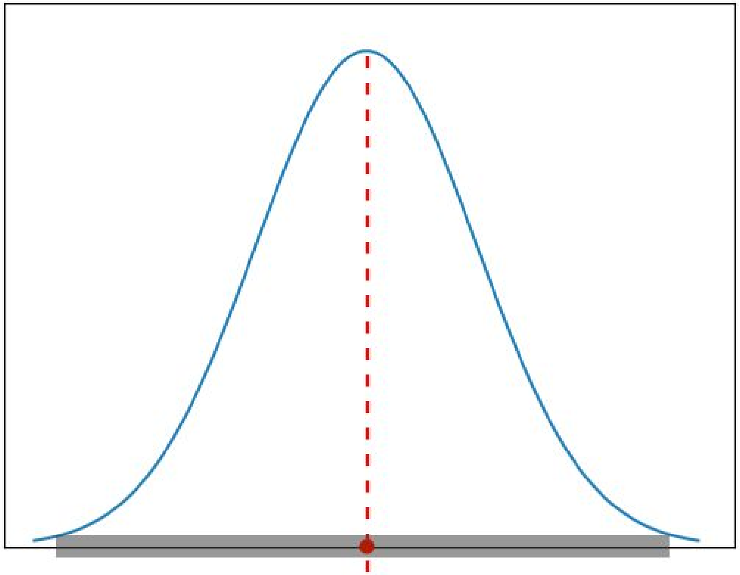


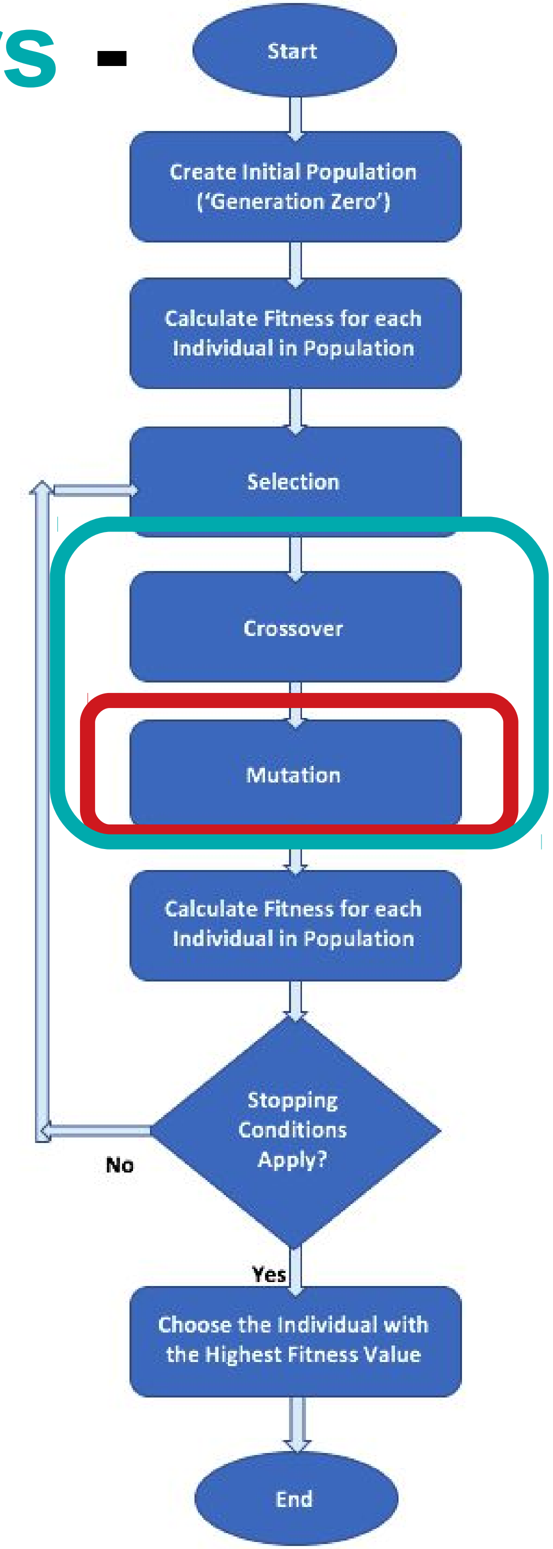


Workflow - Variation Operators - Real-coded — Real Mutation

Another approach is to **generate a random real** number that resides **in the vicinity of the original** individual.

Example: the **normally distributed** (or Gaussian) **mutation** -> a **random number** is generated using a **normal distribution** with a **mean = 0** and some **predetermined standard deviation**.





Content

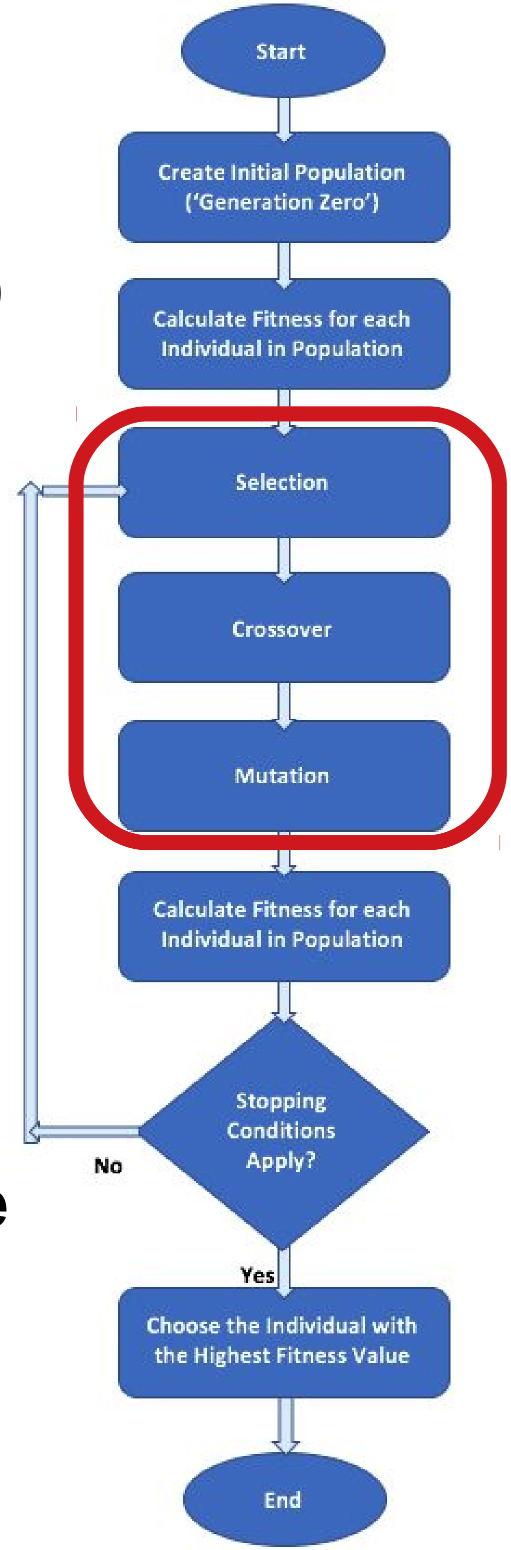
- Recommended Sources
- What is Evolutionary Computing (EC)
- EC History
- Problem Types for EC
- What is Evolutionary Algorithm (EA)
- EA Workflow
- Selection
- Crossover
- Mutation
- Real-coded EA
- **Elitism, Niching, Sharing**

Workflow - Elitism Strategy

We want to guarantee that the **best individual(s)** always make it to the next generation, we can apply the **optional** elitism strategy.

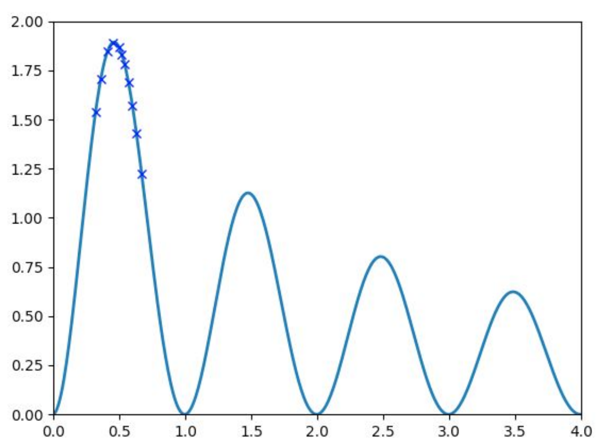
This means that the **top *n* individuals** (*n* is a predefined parameter) **are duplicated into the next generation** before we **fill the rest** of the available spots with **offspring** that are created **using selection, crossover, and mutation**. The **elite** individuals that were duplicated are still **eligible for the selection** process so they can still **be used as the parents** of new individuals.

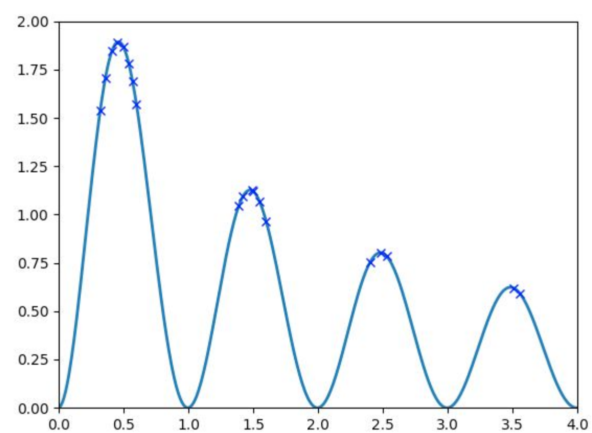
Elitism can sometimes have a significant **positive impact** on the algorithm's performance as it avoids the potential **time waste** needed for **re-discovering good** solutions that were lost.



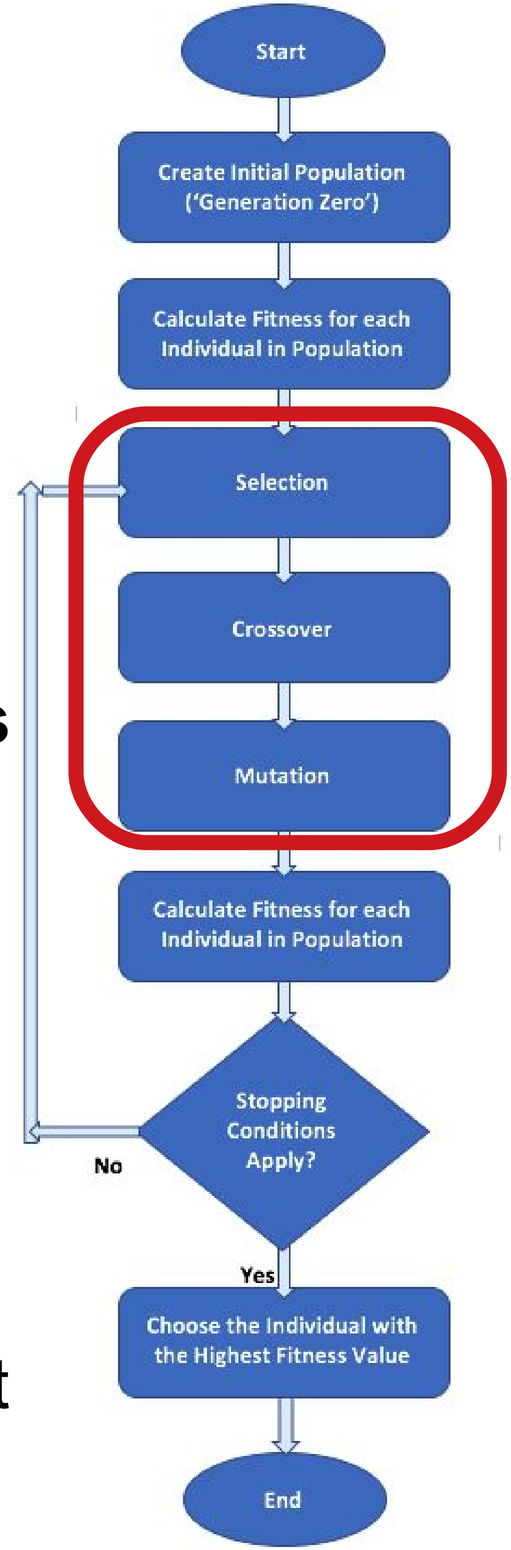
Workflow - Niching and Sharing

When several different species coexist in the same niche, they all compete over the same resources, and a **tendency** is to **search** for new, unpopulated **niches** and populate them.
This can be used to maintain the diversity of the population and to **find several optimal solutions** -> **several niches**.





For this we should **offer resources** in the amount **proportional to a niche height** by **sharing fitness** depended **on distance to others**.



Основи еволюційних обчислень
-
Evolutionary Computing Basics
-
Lecture 03. EC for Machine Learning — Feature Selection

(based on Alan Turing, Holland, Khaled Rasheed, Ben Phillips, Eyal Wirsansky, and others works)

Content

- **Recommended Sources**
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- Resume

Recommended Sources — Books

Books (scientific):

Guyon, I., Gunn, S., Nikravesh, M., & Zadeh, L. A. (Eds.). (2008). *Feature extraction: foundations and applications* (Vol. 207). Springer.

Dong, G., & Liu, H. (Eds.). (2018). *Feature engineering for machine learning and data analytics*. CRC Press.

Books (with codes at github):

Soledad Galli (2020). *Python Feature Engineering Cookbook*. Packt Publishing

Alice Zheng and Amanda Casari (2018). *Feature Engineering for Machine Learning* (O'Reilly)

Recommended Sources - Papers and Datasets

Regression Problem (F1RP):

Breiman, Leo (1996) Bagging predictors. Machine Learning 24, pages 123-140.
Friedman, Jerome H. (1991) Multivariate adaptive regression splines. The Annals of Statistics 19 (1), pages 1-67.

Classification Problem

UCI Zoo dataset (http://archive.ics.uci.edu/ml/datasets/Zoo)
Eibe Frank and Stefan Kramer. Ensembles of nested dichotomies for multi-class problems. ICML. 2004.
Huan Liu and Hiroshi Motoda and Lei Yu. Feature Selection with Selective Sampling. ICML. 2002.

Content

- Recommended Sources
- **EA (GA) for Feature Selection — Why?**
- Problem Types for Feature Selection:
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- Resume

Evolutionary Computing (EC) — for Feature Selection — why?

<u>Supervised learning:</u>

<u>Workflow</u>: the **model** receives a set of **inputs**, called **features**, and maps them to a set of **outputs**.

<u>Assumption</u>: the information described by the **features** is **useful for** determining the value of the corresponding **outputs**.

<u>Common sense</u>: the **more** information we can use as input, the **better** our chances of predicting the output(s) correctly.
<u>Reality</u>: in many cases the **opposite is true** ... if some of the features we use are **irrelevant** or **redundant**, the consequence could be a (sometimes significant) **decrease in the accuracy** of the models.

That is why we need **feature selection**:
the **process** of **selecting** the most **beneficial set of features** out of the entire set of features to **reach the better** solution.

EC for Feature Selection — Benefits

- Decreasing the errors (the lost function) of the model
- Increasing the accuracy of the model
- Training times of the models are shorter.
- Trained models are simpler and easier to interpret.
- Trained resulting models are likely to provide better **generalization**, that is, they perform better with new input data that is dissimilar to the data that was used for training.

Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- **Problem Types for Feature Selection:**
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- Resume

EC for Feature Selection — Problem Types

EC (GA) can be effectively applied to
the classic supervised machine learning problems:

– **regression** (use case of Friedman-1 Regression Problem)
and
– **classification** (use case of UCI-dataset animal classification)

for
– **feature selection**
or
– **dimensionality reduction**

with the purpose of:
– **decrease of MSE**
or
– **increase of mean accuracy**.

Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- **Regression: Friedman-1 Problem**
  - Classic Solution
  - EA (GA) Solution
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- Resume

EC for Feature Selection — Example: Friedman-1 Regression Problem (F1RP)

F1RP was described by Friedman (1991) and Breiman (1996).
Inputs: ***n_features*** independent variables uniformly distributed on the interval [0,1], only 5 out of these ***n_features*** are actually used.
Outputs: are created according to the formula:

$$y(x_0, x_1, x_2, x_3, x_4) = 10 \cdot sin(\pi \cdot x_0 \cdot x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + noise \cdot N(0,1)$$

The last component in the formula is the randomly generated noise. The noise is normally distributed and multiplied by the constant noise, which determines its level.

Various implementations in programming languages:

Python: **make_friedman1()** function in scikit-learn (**sklearn**) library

R: **friedman1()** function in **mlbench** library

Why F1RP is useful for us?

Breiman, Leo (1996) Bagging predictors. Machine Learning 24, pages 123-140.
Friedman, Jerome H. (1991) Multivariate adaptive regression splines. The Annals of Statistics 19 (1), pages 1-67.

EC for Feature Selection — Example: why F1RP is useful for us?

If **n_features = 15**, we will get a dataset with the original **5** input variables (or features) that were used to generate ***y*** values by the formula and **10** features that are completely irrelevant to the output.

Why: **F1RP** is used to **test** various **regression** models as to presence of ***noise*** and irrelevant features in the dataset.

__Example__:

Aim: test EC (GA) as a **feature selection** mechanism.

Workflow: use **make_friedman1()** function to create a dataset with 15 features and use GA to search for the **subset** of features that provides the **best** performance.

Hypothesis: EC (GA) will pick the first 5 features and drop the rest, assuming that the model's accuracy is better when only the relevant features are used as input.

EC (GA) role: The **fitness function** (**FF**) will use a **regression model** that, for each potential **solution** – a **subset of the feature** to use – will be **trained** using the dataset containing **only the selected** features.

EC for Feature Selection — Example:
Individual Representation by EC (GA)

An **individual solution** (genotype) should indicate which **features are selected** and which are dropped:

➔ Each **individual solution** is a **list of binary** values

➔ Every **entry** in the list (0 or 1) is **one of the features** in the dataset:
- ✓ 1 - the corresponding feature **WAS selected**,
- ✓ 0 - the feature has **NOT** been **selected**.

This is very similar to the knapsack 0-1 problem from Lab01.

IMPORTANT:
Each 0 in the individual solution **means**
->
dropping the corresponding **feature's** data **column** from the dataset.

Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- **Regression: Friedman-1 Problem**
  - **Classic Solution**
  - EA (GA) Solution
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- Resume

EC for Feature Selection — Example: F1RP — Classic Solution

1) Create the **dataset** by Friedman formula using ***make_friedman1()*** function in scikit-learn (**sklearn**) library.

2) Divide the data into two subsets – a **training** set and a **validation** set – using ***model_selection.train_test_split()*** function in the scikit-learn.

3) Create the regression **model** ... various can be used ... **Gradient Boosting Regressor (GBR**) in this example.

4) Determine the performance of the used regression model for a set of selected features by ***getMSE()*** **function-metric***.

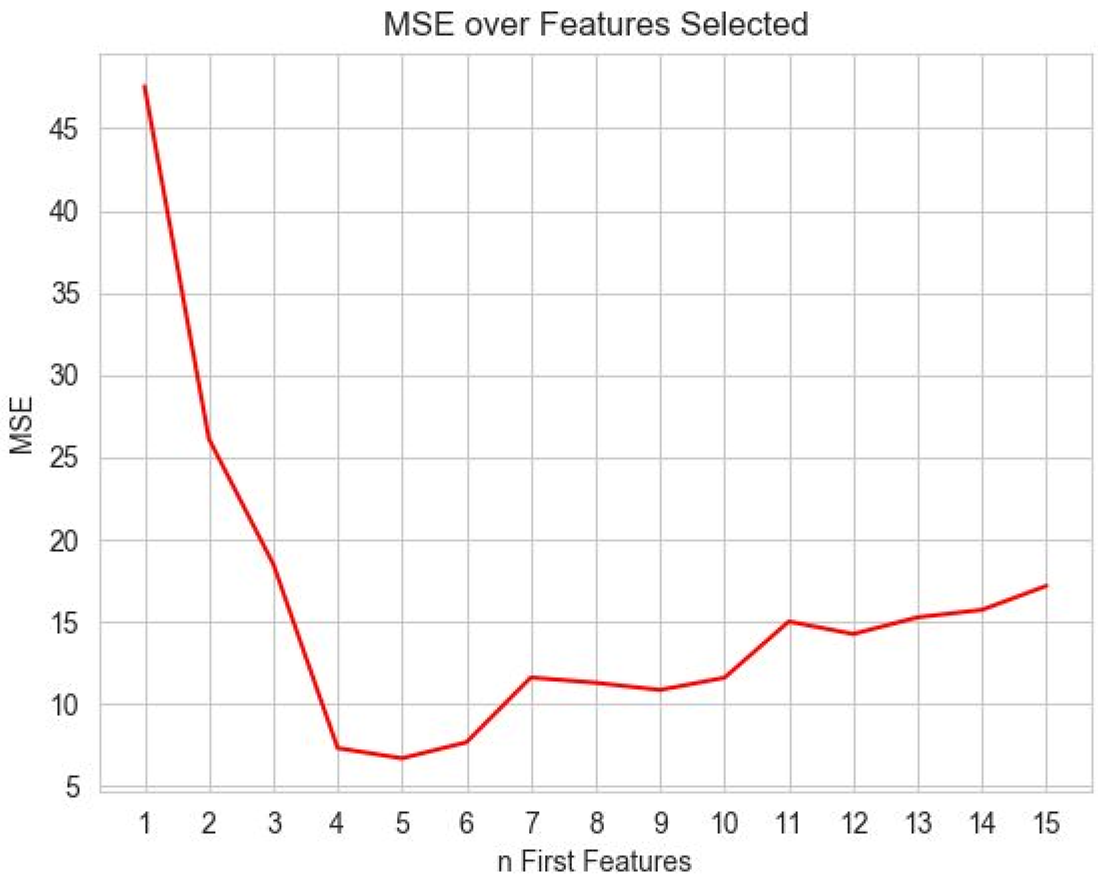
5) Then the **new** training **subset** (with the **selected features only**!) is used to train the model, while the new validation **subset** - to evaluate it.

*) **The mean square error (MSE)** = the average squared difference between the model's predicted values and the actual values. A **lower** value of this

EC for Feature Selection — Example: F1RP — Classic Solution — Results...

As far as we add the first 5 features one by one, the performance improves. However, later each additional feature degrades the performance of the model:

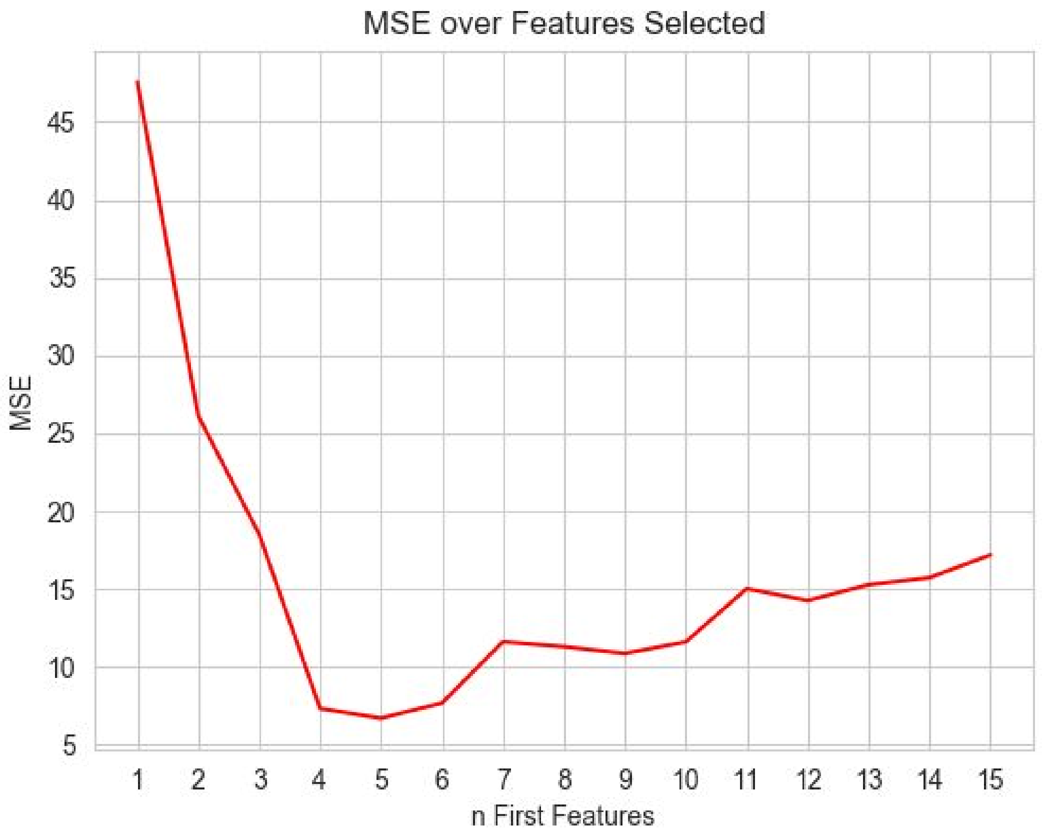
1 first features: score = 47.553993
2 first features: score = 26.121143
3 first features: score = 18.509415
4 first features: score = 7.322589
5 first features: score = 6.702669
6 first features: score = 7.677197
7 first features: score = 11.614536
8 first features: score = 11.294010
9 first features: score = 10.858028
10 first features: score = 11.602919
11 first features: score = 15.017591
12 first features: score = 14.258221
13 first features: score = 15.274851
14 first features: score = 15.726690
15 first features: score = 17.187479



EC for Feature Selection — Example: F1RP — Classic Solution — DEMO...

Try to reproduce these results:

1 first features: score = 47.553993
2 first features: score = 26.121143
3 first features: score = 18.509415
4 first features: score = 7.322589
5 first features: score = 6.702669
6 first features: score = 7.677197
7 first features: score = 11.614536
8 first features: score = 11.294010
9 first features: score = 10.858028
10 first features: score = 11.602919
11 first features: score = 15.017591
12 first features: score = 14.258221
13 first features: score = 15.274851
14 first features: score = 15.726690
15 first features: score = 17.187479



Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- **Regression: Friedman-1 Problem**
  - Classic Solution
  - **EA (GA) Solution**
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- Resume

EC for Feature Selection — Example: F1RP — EC (GA) Solution

The **differences** from **classic** solution:

1) **Chromosomes** - binary lists of selected features

2) **Fitness Function (FF)** - returns the regression model's **MSE**

3) **Selection**
- **tournament** selection with a tournament **size of 2**
- **elitism**, where the **hall of fame (HOF)** members – the current **best** individuals – are **always passed untouched** to the next generation

4) **Evolution (genetic) operators**
- **crossover**
and
- **mutation** operators
that are specialized for binary list chromosomes

EC for Feature Selection — Example: F1RP — EC (GA) Solution — Results

After 30 generations of EC (GA):

Best Ever Individual = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Best Ever Fitness = 6.702668910463287

What does it mean?

The best MSE (about 6.7) is provided by the first five features.

IMPORTANT:
EA (GA) makes **no assumptions** about the set of features.
EA (GA) does **not know** about the first or last *n* features.
EA (GA) simply searched for the **best possible subset** of features.

EC for Feature Selection — Example: F1RP — EC (GA) Solution — DEMO

Try to reproduce these results:

After 30 generations of EC (GA):

Best Ever Individual = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Best Ever Fitness = 6.702668910463287

What does it mean?

The best MSE (about 6.7) is provided by the first five features.

Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- **Classification: Animals Problem**
  - Classic Solution
  - EA (GA) Solution
- Resume

EC for Feature Selection — Example: Animals Classification Problem

It is the classic example of classification problem.

archive.ics.uci.edu/ml/datasets/Zoo

UCI

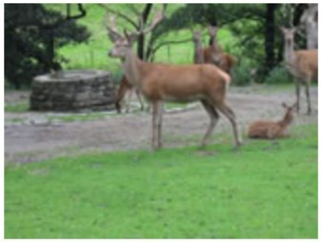
Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Zoo Data Set

Download: Data Folder, Data Set Description

Abstract: Artificial, 7 classes of animals



Data Set Characteristics:	Multivariate	**Number of Instances:**	101	**Area:**	Life
Attribute Characteristics:	Categorical, Integer	**Number of Attributes:**	17	**Date Donated**	1990-05-15
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	346207

UCI Zoo dataset (http://archive.ics.uci.edu/ml/datasets/Zoo).

EC for Feature Selection — Example: Animals Classification — Dataset

Dataset General Information:

A simple database containing **17 Boolean-valued attributes**.

The "**type**" attribute appears to be the **class** attribute. Here is a breakdown of which animals are in which type: (I find it unusual that there are **2 instances of "frog"** and **one of "girl"**!)

Class# -- Set of animals:

1 -- (41) aardvark, antelope, bear, boar, buffalo, calf, cavy, cheetah, deer, dolphin, elephant, fruitbat, giraffe, girl, goat, gorilla, hamster, hare, leopard, lion, lynx, mink, mole, mongoose, opossum, oryx, platypus, polecat, pony, porpoise, puma, pussycat, raccoon, reindeer, seal, sealion, squirrel, vampire, vole, wallaby,wolf

2 -- (20) chicken, crow, dove, duck, flamingo, gull, hawk, kiwi, lark, ostrich, parakeet, penguin, pheasant, rhea, skimmer, skua, sparrow, swan, vulture, wren

3 -- (5) pitviper, seasnake, slowworm, tortoise, tuatara

4 -- (13) bass, carp, catfish, chub, dogfish, haddock, herring, pike, piranha, seahorse, sole, stingray, tuna

5 -- (4) frog, frog, newt, toad

6 -- (8) flea, gnat, honeybee, housefly, ladybird, moth, termite, wasp

7 -- (10) clam, crab, crayfish, lobster, octopus, scorpion, seawasp, slug, starfish, worm

EC for Feature Selection — Example: Animals Classification — Dataset

Attribute (Feature) Information:

1. animal name: Unique for each instance
2. hair: Boolean
3. feathers: Boolean
4. eggs: Boolean
5. milk: Boolean
6. airborne: Boolean
7. aquatic: Boolean
8. predator: Boolean
9. toothed: Boolean
10. backbone: Boolean
11. breathes: Boolean
12. venomous: Boolean
13. fins: Boolean
14. legs: Numeric (set of values: {0,2,4,5,6,8})
15. tail: Boolean
16. domestic: Boolean
17. catsize: Boolean

EC for Feature Selection — Example: Animals Classification — Problem

Origin: it is the classic example of classification problem, where the input features need to be mapped into two or more categories/labels.

Inputs: features 2-17 (hair, feathers, fins, and so on), mostly features are Boolean (value of 1 or 0) meaning the presence or absence of a certain attribute, such as hair, fins, and so on.
Note: The 1st feature - **animal name** - is just to provide us with some information and **does not participate** in the learning.

Outputs: the last feature – **type** – represents 7 categories.
For instance, type 5 represents a category with: frog, newt, and toad.

Aim: **train** a **classification model** on this dataset with **features 2-17** (hair, feathers, fins, and so on) to **predict** the value of **feature 18** (animal **type**).

Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- **Classification: Animals Problem**
  - **Classic Solution**
  - EA (GA) Solution
- Resume

EC for Feature Selection — Example: Animals Classification — Classic Way

1) Load the UCI-Zoo **dataset** by the standard **read_csv** function.

2) Divide the data into **input** features (first remaining 16 columns) and the resulting **output** category (last column). Then instead of separating the data into **1 training** set and **1 test** set, like we did in the previous section, we're using ***k-fold cross-validation ->*** The data is split into ***k*** equal parts and the model is evaluated ***k*** times:
(k-1) parts for **training** and **1** remaining part for testing (or **validation**).

3) Create the classification **model** ... various models can be used ...
Decision Tree Classifier (DCT) in this example.

4) Determine the performance of the used regression model for a set of selected features by ***getMeanAccuracy()*** **function-metric***.

*) **Accuracy** – the portion of the cases that were classified correctly. A **higher** value of this measurement indicates **better performance** of the model.

EC for Feature Selection — Example: Classification — Classic Way — DEMO...

After training/testing:

the model - DTC-classifier
5-fold cross-validation
all 16 features

the classification accuracy was about **91%**.

Try to reproduce these results:

All features selected:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Accuracy = 0.9099999999999999

Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- **Classification: Animals Problem**
  - Classic Solution
  - **EA (GA) Solution**
- Resume

EC for Feature Selection — Example: Classification — EC (GA) Solution

The **differences** from **classic** solution:

1) **Chromosomes** - binary lists of selected features

2) **Fitness Function (FF)** - returns the model's **mean accuracy**

3) **Selection**
- **tournament** selection with a tournament **size of 2**
- elitism, where the **hall of fame (HOF)** members – the current **best** individuals – are **always passed untouched** to the next generation

4) **Evolution (genetic) operators**
- **crossover**
and
- mutation operators
that are specialized for binary list chromosomes

EC for Feature Selection — Example: Classification — EC (GA) — Results

After 50 generations of EC (GA) and HOF size of 5:

Best solutions are:
0 : [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0] fitness = 0.964 accuracy = 0.97 features = 6
1 : [0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0] fitness = 0.963 accuracy = 0.97 features = 7
2 : [1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0] fitness = 0.963 accuracy = 0.97 features = 7
3 : [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0] fitness = 0.963 accuracy = 0.97 features = 7
4 : [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1] fitness = 0.963 accuracy = 0.97 features = 7

The **top solution** is the set of **6 features**, which are as follows:
feathers, milk, airborne, backbone, fins, tail

By selecting these particular features out of the 16 given in the dataset:
1 - we **reduced the dimensionality** of the problem,
2 - we also **improved** our model **accuracy** from 91% to 97%.

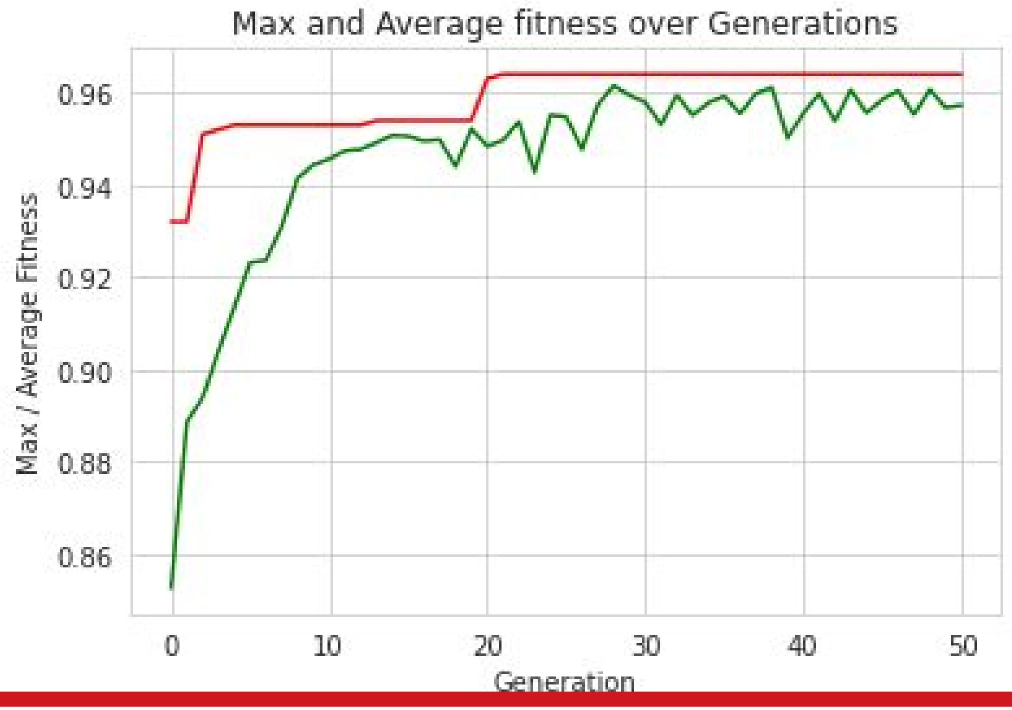
IMPORTANT: It is **not very large** increase of an absolute **accuracy**, **BUT** a **great (TRIPLE!) reduction** of the **error rate** from 9% to 3% – a very significant improvement in terms of classification performance

EC for Feature Selection — Example: Classification — EC (GA) — DEMO

Try to reproduce these results:

Best solutions are:

0 : [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0] fitness = 0.964 accuracy = 0.97 features = 6
1 : [0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0] fitness = 0.963 accuracy = 0.97 features = 7
2 : [1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0] fitness = 0.963 accuracy = 0.97 features = 7
3 : [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0] fitness = 0.963 accuracy = 0.97 features = 7
4 : [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1] fitness = 0.963 accuracy = 0.97 features = 7



Content

- Recommended Sources
- EA (GA) for Feature Selection — Why?
- Problem Types for Feature Selection:
- Regression: Friedman-1 Problem
  - Classic Solution
  - EA (GA) Solution
- Classification: Animals Problem
  - Classic Solution
  - EA (GA) Solution
- **Resume**

EC for Feature Selection — Resume

EC (GA) can be effectively applied to
the classic supervised machine learning problems:

– **regression** (use case of Friedman-1 Regression Problem)
and
– **classification** (use case of UCI-dataset animal classification)

for
– **feature selection**
or
– **dimensionality reduction**

with the purpose of:
– **decrease of MSE**
or
– **increase of mean accuracy**.

Основи еволюційних обчислень
-
Evolutionary Computing Basics
-
Lecture 04. EC for Machine Learning — Hyperparameter Tuning

(based on Holland, Khaled Rasheed, Ben Phillips, Eyal Wirsansky, and others works)

Content

- **Recommended Sources**
- EA (GA) for Hyperparameter Tuning — Why?
- Problem Types for Feature Selection
- Classification Problem Example
    - UCI Wine Dataset
    - Hyperparameter Tuning
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- Resume

Recommended Sources — Books

Books (scientific):
Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning*. Cambridge: MIT press
Цитовано в 23692 джерелах.

Books (with codes at github):
Alan Fontaine (2018) *Mastering Predictive Analytics with scikit-learn and TensorFlow*. Packt Publishing.

Tanay Agrawal (2021). *Hyperparameter Optimization in Machine Learning: Make Your Machine Learning and Deep Learning Models More Efficient*, Apress

Recommended Sources - Papers and Datasets

Example Problem and Dataset

UCI Wine dataset (https://archive.ics.uci.edu/ml/datasets/wine)

S. Aeberhard, D. Coomans and O. de Vel,
Comparison of Classifiers in High Dimensional Settings,
Tech. Rep. no. 92-02, (1992), Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland.
The data was used for comparing various classifiers.
(RDA : 100%, QDA 99.4%, LDA 98.9%, 1NN 96.1% (z-transformed data))
(All results using the leave-one-out technique)

Mikhail Bilenko (**Head of AI and Research, Yandex**) and Sugato Basu and Raymond J. Mooney. *Integrating constraints and metric learning in semi-supervised clustering*. ICML. 2004.

Kamal Ali and Michael J. Pazzani. *Error Reduction through Learning Multiple Descriptions*. Machine Learning, 24. 1996

Content

- Recommended Sources
- **EA (GA) for Hyperparameter Tuning — Why?**
- Problem Types for Feature Selection
- Classification Problem Example
    - UCI Wine Dataset
    - Hyperparameter Tuning
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- Resume

Evolutionary Computing (EC) —
for Hyperparameter Tuning — why?

Supervised learning:

Workflow: the **model** receives a set of **inputs**, called **features**, and maps them to a set of **outputs**.
Assumption: the information described by the **features** is **useful for** determining the value of the corresponding **outputs**.
Model: learning is **adjusting** (or tuning) the **internal parameters** of a model to produce the **desired outputs** in response to **given inputs**. For this, each type of supervised learning model is accompanied by a **learning algorithm** that **iteratively adjusts** its internal **parameters** during the learning (or training) phase.
Reality: **BUT** ... most models have another set of **hyperparameters** that are set **before** the learning and they affect the way the learning is done!
Usually: hyperparameters have some **default values** that will take effect if we don't specifically set them and **they are not optimal!**
That is why we need **hyperparameter tuning!**

EC for Hyperparameter Tuning — Benefits and Overheads

Benefits:

- Decreasing the errors (the lost function) of the model
- Increasing the accuracy of the model
- Training times of the models are shorter.

Overheads:

- The possible number of hyperparameter combinations can be very-very huge.
- Search for the best hyperparameter combinations (hyperparameter tuning) takes significant amounts of time.

Content

- Recommended Sources
- EA (GA) for Hyperparameter Tuning — Why?
- **Problem Types for Feature Selection**
- Classification Problem Example
    - UCI Wine Dataset
    - Hyperparameter Tuning
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- Resume

EC for Hyperparameter Tuning — Problem Type - Classification

EC (GA) can be effectively applied to
the classic supervised machine learning probles:

– **classification** (use case of UCI-dataset Wine classification)

for
– **hyperparameter tuning**

with the purpose of:
– **decrease of MSE**
or
– **increase of mean accuracy**.

Content

- Recommended Sources
- EA (GA) for Hyperparameter Tuning — Why?
- Problem Types for Feature Selection
- **Classification Problem Example**
    - **UCI Wine Dataset**
    - Hyperparameter Tuning
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- Resume

EC for Hyperparameter Tuning —
Example: Wine Classification Problem

It is the classic example of classification problem.

UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Wine Data Set

Download: Data Folder, Data Set Description

Abstract: Using chemical analysis determine the origin of wines

Data Set Characteristics:	Multivariate	Number of Instances:	178	Area:	Physical
Attribute Characteristics:	Integer, Real	Number of Attributes:	13	Date Donated	1991-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	1602802

Source:

Original Owners:

Forina, M. et al, PARVUS -
An Extendible Package for Data Exploration, Classification and Correlation.
Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno,
16147 Genoa, Italy.

UCI Wine dataset (https://archive.ics.uci.edu/ml/datasets/wine)

EC for Hyperparameter Tuning — Wine Classification — Dataset

Dataset General Information:

These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from **3 different cultivars**.

The analysis determined the quantities of **13 constituents** found in each of the 3 types of wines.

- In a classification context, this is a well posed problem with "well behaved" class structures.
- A good data set for first testing of a new classifier, but not very challenging.

EC for Hyperparameter Tuning — Wine Classification — Dataset

Attribute (Feature) Information:

1) Alcohol
2) Malic acid
3) Ash
4) Alcalinity of ash
5) Magnesium
6) Total phenols
7) Flavanoids
8) Nonflavanoid phenols
9) Proanthocyanins
10) Color intensity
11) Hue
12) OD280/OD315 of diluted wines
13) Proline

Class identifier: One (0th) attribute is class identifier (1,2,3)

Content

- Recommended Sources
- EA (GA) for Hyperparameter Tuning — Why?
- Problem Types for Feature Selection
- Classification Problem Example
    - UCI Wine Dataset
    - **Workflow and Hyperparameter Tuning**
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- Resume

EC for Hyperparameter Tuning — Wine Classification — Problem

Origin: it is the **classic** example of **classification problem**, where the input features need to be mapped into **3 categories**/labels.

Inputs: all features (wine properties) are **continuous**.

Outputs: the one feature – **class** – represents 3 categories (**cultivars**).

Aim: **train** a **classification model** on this dataset with **13 features** to **predict** the value of **feature 0** (**cultivar**).

Wine Classification — Workflow

1) Load the UCI Wine **dataset** by the standard **read_csv** function (with url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data').

2) Divide the data into **input** features (first remaining 13 columns) and the resulting **output** category (the first column). Then instead of separating the data into **1 training** set and **1 test** set, like we did in the previous example, we're using ***k-fold cross-validation* ->** The data is split into ***k*** equal parts and the model is evaluated ***k*** times:
(k-1) parts for **training** and **1** remaining part for testing (or **validation**).

3) Create the classification **model** ... various models can be used ... **AdaBoostClassifier** in this example.

4) Determine the performance of the used regression model for a set of selected **hyperparameters** by ***accuracy*** **metric***.

*) **Accuracy** – the portion of the cases that were classified correctly. A **higher** value of this measurement indicates **better performance** of the model.

Wine Classification — Hyperparameter Tuning

Let’s consider in details this stage:
3) Create the classification **model** ... various models can be used ... **AdaBoostClassifier** in this example.

The ***adaptive boosting algorithm*** (***AdaBoost***) is a powerful ML model that **combines** the **outputs** of multiple **instances** of a simple ML algorithm (weak learner) using a weighted sum. AdaBoost adds instances of the weak learner during the learning process, each of which is adjusted to improve previously misclassified inputs.
We’ll use *sklearn* library's implementation of AdaboostClassifier with some hyperparameters:

Name	**Type**	**Description**	**Default Value**
`n_estimators`	int	The maximum number of estimators	50
`learning_rate`	float	Can be used to shrink the contribution of each classifier	1
`algorithm`	`{'SAMME', 'SAMME.R'}`	`'SAMME.R'` – uses a real boosting algorithm, `'SAMME'` – uses a discrete boosting algorithm	2

Content

- Recommended Sources
- EA (GA) for Hyperparameter Tuning — Why?
- Problem Types for Feature Selection
- Classification Problem Example
    - UCI Wine Dataset
    - Hyperparameter Tuning
- **Classic Solutions**
    - **DEMO 1 - Default Values**
    - **DEMO 2 - Extensive Grid Search**
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- Resume

Wine Classification — Classic Way

Let's start from 2 classic approaches:

◆**default values** of model hyperparameters:
{'**algorithm**': 'SAMME.R', '**learning_rate**': **1.0**, '**n_estimators**': **50**, 'random_state': 42},

◆**grid search** of the **best values** of model hyperparameters:

Algorithm — 2 possible values 'SAMME' and 'SAMME.R',
learning_rate - 10 values logarithmically spaced between 0.01 (10^{-2}) and 1 (10^{0}),
n_estimators -> 10 values linearly spaced between 10 and 100,

Total: 200 = (10×10×2) different combinations of the grid parameters.

Wine Classification —
Classic Way — DEMO 1 and 2

Results:

DEMO 1 - Default values:

Default Classifier Hyperparameter values:
{'algorithm': 'SAMME.R', 'base_estimator': None, 'learning_rate': 1.0, 'n_estimators': 50, 'random_state': 42}
Score (with default values) = 0.6457142857142857
Time Elapsed = 0.4167492389678955

DEMO 2 - After gridSearch:

Best parameters: {'algorithm': 'SAMME.R', 'learning_rate': 0.3593813663804626, 'n_estimators': 70}
Score (after gridSearch): 0.9325842696629213
Time Elapsed = 74.51628732681274

Try to reproduce these results!

Wine Classification —
Classic Way — DEMO 1 and 2

After training/testing - see **test.gridTest()** function in the DEMO code:

◆the model is **AdaBoostClassifier**-classifier

◆5-fold cross-validation

Results:

DEMO 1 - Default values:

Model hyperparameter values:

{'algorithm': 'SAMME.R', 'learning_rate': **1.0**, 'n_estimators': **50**, 'random_state': 42}

Accuracy: 0.65%

Time Elapsed = 0.42 seconds

DEMO 2 - After gridSearch:

Best hyperparameter values:

{'algorithm': 'SAMME.R', 'learning_rate': **0.359**, 'n_estimators': **70**}

Accuracy: 0.93%

Time Elapsed = 74 seconds

Try to reproduce these results!

Content

- Recommended Sources
- EA (GA) for Hyperparameter Tuning — Why?
- Problem Types for Feature Selection
- Classification Problem Example
    - UCI Wine Dataset
    - Hyperparameter Tuning
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- **EA (GA) Solutions**
    - **DEMO 3 — GA-driven Grrid Search**
    - **DEMO 4 — Direct GA**
- Resume

Wine Classification — EC (GA) Ways
Difference from Classic Ways

The **differences** from **classic** solution:

1) **Chromosomes** — **heterogeneous** sets of selected **values of hyperparameters**:

- **n_estimators values** - a list of **10 integers**
- **learning_rate** - an ndarray of 10 **floats,**
- **algorithm** - a list of **2 strings**

2) **Fitness Function (FF)** - returns the model's **mean accuracy**

3) **Selection**

- **tournament** selection with a tournament **size of 2**
- **elitism**, where the **hall of fame (HOF)** members – the current **best** individuals – are **always passed untouched** to the next generation

4) **Evolution (genetic) operators**

- **crossover** and
- **mutation** operators

that are specialized for chromosomes

Wine Classification — EC (GA) Ways — Grid and Direct

The possible EC-GA-based approaches:

- **GA-based grid search:**
  to search **among** the **initially selected** 200 grid combinations **only**,

- **direct** GA**:**
  to search directly the **entire parameter space,**
  where each hyperparameter can be represented
  as a variable participating in the search,
  and the **chromosome** can be a **combination** of **all** these variables.

Wine Classification — EC (GA) Ways

3.GA-based Grid Search — DEMO 3

3) **GA-based grid search:**

****

--- Evolve in 200 possible combinations ---

gen	nevals	avg	min	max	std
0	20	0.708427	0.117978	0.910112	0.265992
1	13	0.865169	0.662921	0.926966	0.0717915
2	15	0.887921	0.646067	0.926966	0.0571676
3	12	0.896348	0.679775	0.926966	0.0526256
4	16	0.918539	0.88764	0.926966	0.0110233
5	9	0.911517	0.730337	0.926966	0.0425958

Best individual is: {'n_estimators': 60, 'learning_rate': 0.5994842503189409, 'algorithm': 'SAMME.R'}
with fitness: 0.9269662921348315
Time Elapsed = 24.287983655929565

Try to reproduce these results!

Wine Classification — EC (GA) Ways

3.GA-based Grid Search — DEMO 3

3) **GA-based grid search:**
to search **among** the **initially selected** 200 grid combinations **only**
GA-parameters:
population_size=20,
gene_mutation_prob=0.30,
tournament_size=2,
generations_number=5

Results:
Model hyperparameter values:
{'algorithm': 'SAMME.R', 'learning_rate': **0.5995**, 'n_estimators': 6**0**, 'random_state': 42}
Accuracy: **0.93%**
Time Elapsed = 24 secs for 6 generations (compare with gridSearch: 74 sec, but it takes only 2 generations - 8 secs! - to reach Max Accuracy)
Try to reproduce these results!

Wine Classification — EC (GA) Ways
3.GA-based Grid Search — DEMO 3

Conclusions:

GA-driven grid search can find the same best result
(found by the classic search),
but in a **6 times(!) faster** – about 12 seconds (2 generations).

BUT ... in real-life situations:

- **datasets** are much **larger**,
- **models** are more **complex**, and
- **hyperparameter grids** are **larger**!

- That is why **exhaustive classic grid search** can be **prohibitively lengthy**, while the **GA-driven grid search** can reach **good results** within a **reasonable time**.

BUT here ... GAs are limited to the subset of hyperparameter values that are defined by the grid.
Let's search outside the grid of a subset of predefined values?

Wine Classification — EC (GA) Ways
4.Direct GA — DEMO 4

4) **Direct** GA**:**
to search directly the **entire parameter space,**
where each hyperparameter can be represented
as a variable participating in the search,
and the **chromosome** can be a **combination** of **all** these variables.

We need to represent each hyperparameter as a floating-point number,
regardless of its actual type:

- **n_estimators** - originally an **integer** — it will be represented by a **float** value in the **range of [1, 100]**,
- **learning_rate** - already a **float**, so no conversion is needed — it will be bound to the **range of [0.01, 1.0]**,
- **algorithm** - have one of two **string** values, 'SAMME' or 'SAMME.R' — it and will be represented by a **float** number in the range of **[0, 1]**.

Wine Classification — EC (GA) Ways

4.Direct GA — DEMO 4

4) **Direct GA - Results:**

****

gen	nevals	max	avg
0	20	0.92127	0.841024
1	14	0.943651	0.900603
2	13	0.943651	0.912841
3	14	0.943651	0.922476
4	15	0.949206	0.929468
5	13	0.949206	0.938563

Time Elapsed = 46.62226867675781

- Best solution is:

params = 'n_estimators'= 69, 'learning_rate'=0.628, 'algorithm'=SAMME.R

Accuracy = 0.94921

Try to reproduce these results!

Wine Classification — EC (GA) Ways

4.Direct GA — DEMO 4

4) **Direct GA with GA-parameters:**
population_size=20,
gene_mutation_prob=0.50,
probability for crossover = 0.90,
tournament_size=2,
generations_number=5
hall_of_fame_size=5

Results:

Model hyperparameter values:
{'algorithm': '**SAMME.R**', 'learning_rate': **0.628**, 'n_estimators': **69**, 'random_state': 42}

Accuracy: 0.95%

Time Elapsed = 46 secs for 6 generations (compare with gridSearch: 74 sec, but it takes only 2 generations - 16 secs! - to > GA-grid Accuracy)

Try to reproduce these results!

Wine Classification — EC (GA) Ways
4.Direct GA — DEMO 4

Conclusions:

◆ Direct GA can find the **better accuracy 95%** than classic (65-93%) and GA-driven grid search (93%),

◆ and in **4-5 times(!) faster** (8 secs for 2 generations) than classic and the same time for GA-driven grid search.

NOTE: the **best** hyperparameter values (for n_estimators and learning_rate) were found **outside the grid** values!

BUT ... again! ... in real-life situations:

◆ **datasets** are much **larger**,

◆ **models** are more **complex**, and

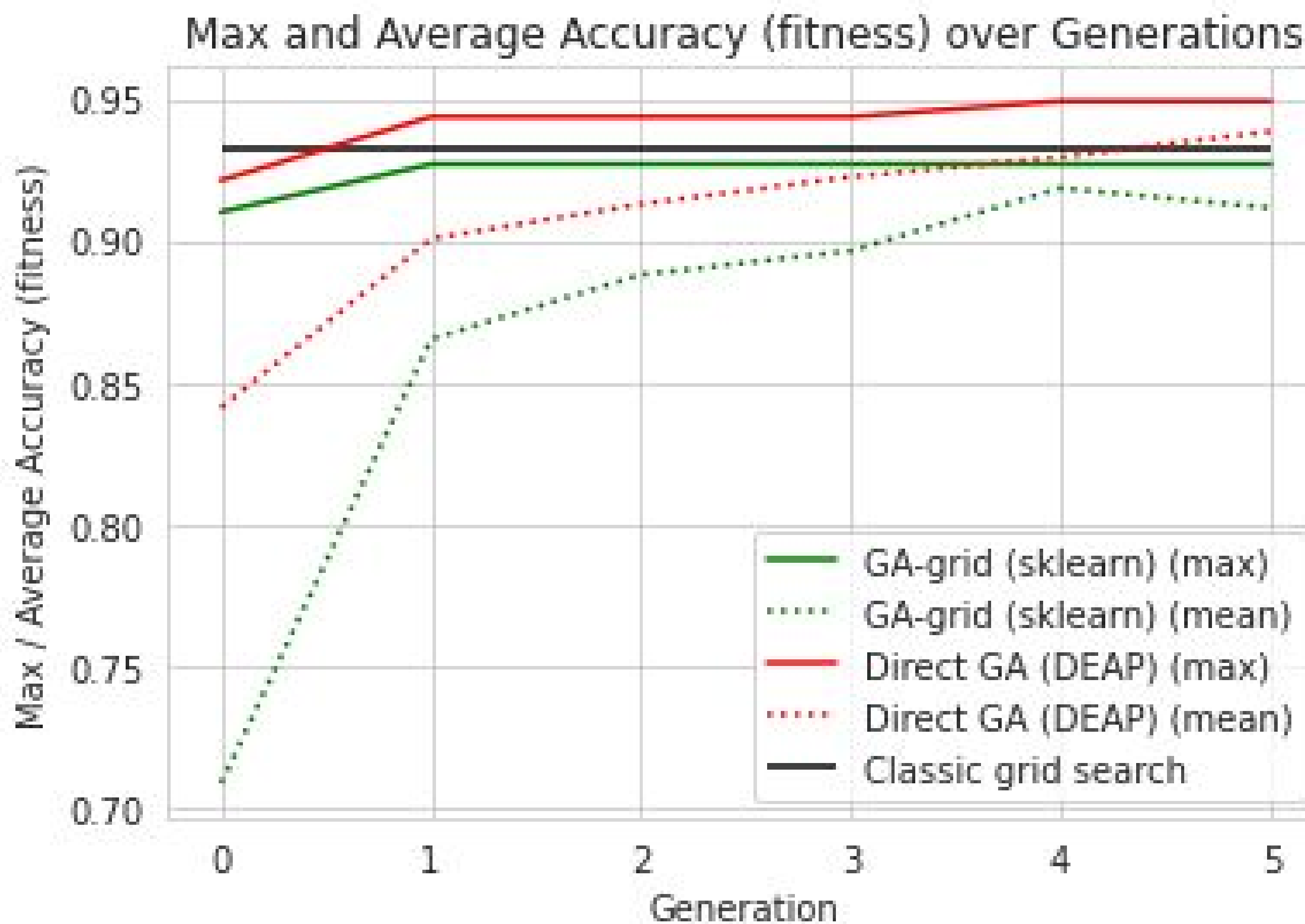
◆ **hyperparameter grids** are **larger**!

● That is why **exhaustive classic grid search** can be **prohibitively lengthy**, while the **GA-driven grid search** can reach **good results** within a **reasonable time**.

Content

- Recommended Sources
- EA (GA) for Hyperparameter Tuning — Why?
- Problem Types for Feature Selection
- Classification Problem Example
    - UCI Wine Dataset
    - Hyperparameter Tuning
- Classic Solutions
    - DEMO 1 - Default Values
    - DEMO 2 - Extensive Grid Search
- EA (GA) Solutions
    - DEMO 3 — GA-driven Grrid Search
    - DEMO 4 — Direct GA
- **Resume**

EC for Feature Selection — Classification — Comparative Plot



Try to reproduce these results!

EC for Feature Selection — Resume

EC (GA) can be effectively applied to
the classic supervised machine learning problems:

– **regression** (use case of Friedman-1 Regression Problem)
and
– **classification** (use case of UCI-dataset animal classification)

for
– **feature selection**
or
– **dimensionality reduction**

with the purpose of:
– **decrease of MSE**
or
– **increase of mean accuracy**.

Основи еволюційних обчислень
-
Evolutionary Computing Basics
-
Lecture 05. EC for Neural Networks — Architecture and Hyperparameter Tuning

(based on Varoquaux, Grobler, Rasheed, Phillips, Wirsansky, and others works)

Content

- **Recommended Sources**
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

Recommended Sources — Books

Books (scientific):
Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning*. Cambridge: MIT press
Цитовано в 23692 джерелах.

Books (with codes at github):
Alan Fontaine (2018) *Mastering Predictive Analytics with scikit-learn and TensorFlow*. Packt Publishing.

Tanay Agrawal (2021). *Hyperparameter Optimization in Machine Learning: Make Your Machine Learning and Deep Learning Models More Efficient*, Apress

Recommended Sources - Papers and Datasets

Example Problem and Dataset

UCI Wine dataset (https://archive.ics.uci.edu/ml/datasets/wine)
S. Aeberhard, D. Coomans and O. de Vel,
Comparison of Classifiers in High Dimensional Settings,
Tech. Rep. no. 92-02, (**1992**), Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland.

UCI Iris dataset (https://archive.ics.uci.edu/ml/datasets/iris)
Fisher,R.A. *The use of multiple measurements in taxonomic problems,* Annual Eugenics, 7, Part II, 179-188 (**1936**); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).

UCI Breast Cancer dataset
(https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic))
W.N. Street, W.H. Wolberg and O.L. Mangasarian. *Nuclear feature extraction for breast tumor diagnosis*. IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology, volume 1905, 861-870, San Jose, CA, **1993**.

Content

- Recommended Sources
- **EA (GA) for Neural Network (NN) Tuning**
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

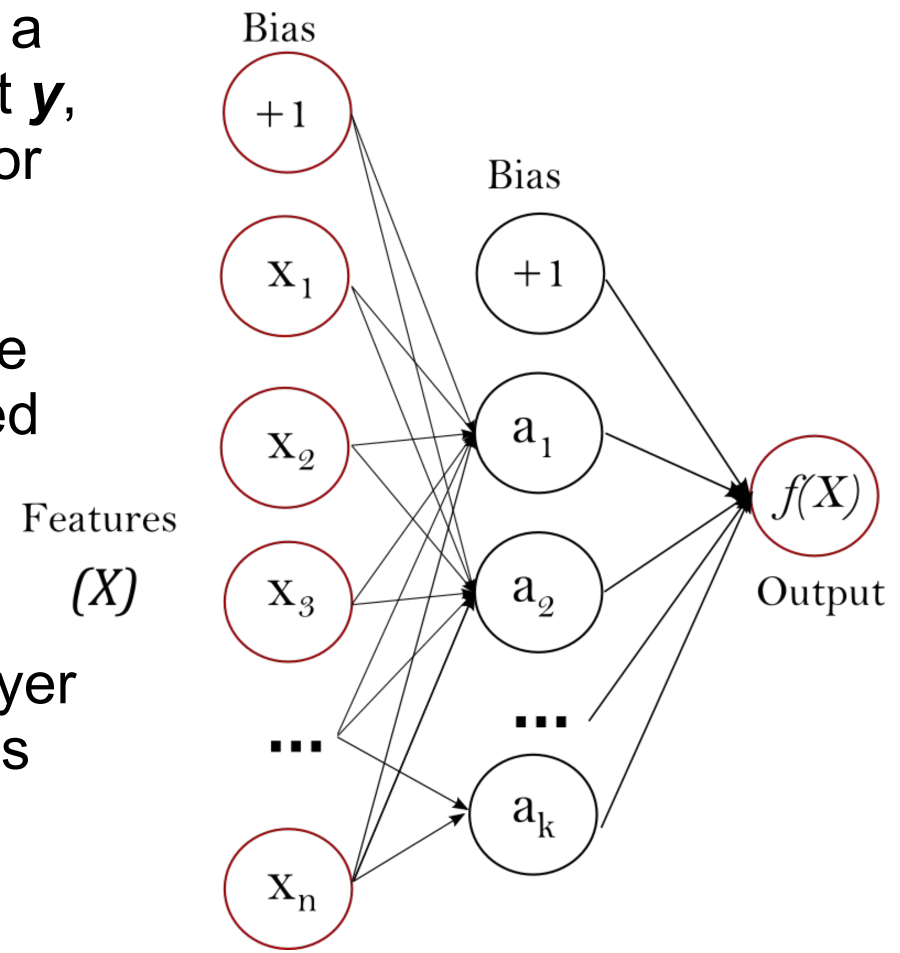
Evolutionary Computing (EC) — MLP/Neural Network (NN) — Intro

Multi-layer Perceptron (MLP) is a supervised learning algorithm (**artifical neural netwrork - NN**) that learns a function ***f*(*X*)** by training on a dataset. Given a set of features ***X*** and a target ***y***, it can learn a non-linear function approximator for classification or regression.

Between the input and the output layer, there can be one or **more non-linear layers**, called **hidden layers**.

The weight matrix $\boldsymbol{W_i}$ at some index $\boldsymbol{i}$ represents the weights between layer $\boldsymbol{i}$ and layer $\boldsymbol{i+1}$. The bias $\boldsymbol{b_i}$ at index $\boldsymbol{i}$ represents the bias values added to layer $\boldsymbol{i+1}$.

MLP uses **backpropagation** for training.
It can **distinguish not linearly** separable data.



NN Tuning —
What are Tuning Objects?

Supervised learning:

Workflow: the **model (NN here)** receives a set of **inputs**, called **features**, and maps them to a set of **outputs**.

Assumption: the information described by the **features** is **useful for** determining the value of the corresponding **outputs**.

Model: learning is **adjusting** (or tuning) the **internal parameters (weights in NN layers here)** of a model to produce the **desired outputs** in response to **given inputs**. Each type of supervised learning model is accompanied by a **learning algorithm** that **iteratively adjusts** its internal **parameters (weights in NN layers here)** during the learning.

AND ... most models **(NN here)** have structure (**NN architecture here: layers, blocks, and connections between them**) + **hyperparameters** (learning rate, ...) that are set **before** the learning and they affect it!

Usually: EC can be applied for search of optimal: a) **weights**, b) **hyperparameters** (like in the previous lecture for ML), c) **architecture**.

IMPORTANT: Weights tuning by EC is NOT considered here, because it is performed by gradient-based methods.

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- **Types of NN Tuning**
- Classification Problem Example: Dataset + Workflow
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

NN Tuning —
What Types of Tuning?

Tuning Types ... for various parts of NN:

◆ **internal parameters:**

weights in NN - is **NOT considered** here, because it is performed by gradient-based methods;

◆ **external parameters:**

1) NN **architecture** (layers and nodes in layers here)

+ influence of various ...

- RANDOM_SEEDs,

- datasets,

- **MAX number of layers**.

2) NN **hyperparameters** (learning rate, activation function, optimization solver, and regularization, here),

3) NN **architecture +** NN **hyperparameters**.

EC for Hyperparameter Tuning — Benefits and Overheads

Benefits:

- Decreasing the errors (the lost function) of the model
- Increasing the accuracy of the model
- Training times of the models are shorter.

Overheads:

- The possible **number** of **NN architectures** and **NN hyperparameter** combinations can be very-very **huge**.
- Search for the best **NN architectures** and **NN hyperparameter** combinations (hyperparameter tuning) takes **significant** amounts of **time**.

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- **Classification Problem Example**
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

EC for Hyperparameter Tuning — Problem Type - Classification

EC (GA) can be effectively applied to
the classic supervised machine learning probles:

– **classification** (use case of UCI-dataset Wine classification)

for
– **NN tuning**

with the purpose of:
– **decrease of MSE**
or
– **increase of mean accuracy**.

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- **Classification Problem: Dataset + Workflow**
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

EC for Hyperparameter Tuning — Example: Wine Classification Problem

It is the classic example of classification problem.

UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Wine Data Set

Download: Data Folder, Data Set Description

Abstract: Using chemical analysis determine the origin of wines

Data Set Characteristics:	Multivariate	Number of Instances:	178	Area:	Physical
Attribute Characteristics:	Integer, Real	Number of Attributes:	13	Date Donated	1991-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	1602802

Source:

Original Owners:

Forina, M. et al, PARVUS -
An Extendible Package for Data Exploration, Classification and Correlation.
Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno,
16147 Genoa, Italy.

UCI Wine dataset (https://archive.ics.uci.edu/ml/datasets/wine)

EC for Hyperparameter Tuning — Wine Classification — Dataset

Dataset General Information:

These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from **3 different cultivars**.

The analysis determined the quantities of **13 constituents** found in each of the 3 types of wines.

- In a classification context, this is a well posed problem with "well behaved" class structures.
- A good data set for first testing of a new classifier, but not very challenging.
-

EC for Hyperparameter Tuning — Wine Classification — Dataset

Attribute (Feature) Information:

1) Alcohol
2) Malic acid
3) Ash
4) Alcalinity of ash
5) Magnesium
6) Total phenols
7) Flavanoids
8) Nonflavanoid phenols
9) Proanthocyanins
10) Color intensity
11) Hue
12) OD280/OD315 of diluted wines
13) Proline

Class identifier: One (0th) attribute is class identifier (1,2,3)

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- **Classification Problem: Dataset + Workflow**
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

EC for Hyperparameter Tuning — Wine Classification — Workflow

Origin: it is the **classic** example of **classification problem**, where the input features need to be mapped into **3 categories**/labels.

Inputs: all features (wine properties) are **continuous**.

Outputs: the one feature – **class** – represents 3 categories (**cultivars**).

Aim: **train** a **classification model** on this dataset with **13 features** to **predict** the value of **feature 0** (**cultivar**).

NN Tuning —
Wine Classification — Workflow

1) Load the UCI Wine **dataset** by the standard **read_csv** function (with url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data').

2) Divide the data into **input** features (first remaining 13 columns) and the resulting **output** category (the first column). Then instead of separating the data into **1 training** set and **1 test** set, like we did in the previous example, we're using ***k-fold cross-validation* ->** The data is split into ***k*** equal parts and the model is evaluated ***k*** times:
(k-1) parts for **training** and **1** remaining part for testing (or **validation**).

3) Create the classification **model** ... various models can be used ...
Multi-layer Perceptron (MLP) in this example.

4) Determine the performance of the used regression model for a set of selected **hyperparameters** by ***accuracy*** **metric***.

*) **Accuracy** – the portion of the cases that were classified correctly. A **higher** value of this measurement indicates **better performance** of the model.

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- **DEMO - Part 1: NN Architecture Tuning Solution**
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

DEMO — Part1: NN Architecture Tuning — Layers and Nodes

We limit NN to **4** hidden layers, the chromosome will be:

[n_1, n_2, n_3, n_4]

Here, ***n*** i denotes the number of nodes in the layer ***i*** from 1 to 4.
To control the number of hidden layers in NN, some of n_i may be 0 or <0 ... it means -> **no more layers** will be added to NN.

<u>Example of some chromosomes:</u>

[10, 20, **-5**, 15] -> tuple (10, 20) since **-5** ends the layer count.
[10, **0**, -5, 15] -> tuple (10,) since **0** ends the layer count.
[10, 20, 5, **-15**] -> tuple (10, 20, 5) since **-15** ends the count.
[10, 20, 5, 15] > tuple (10, 20, 5, 15).

DEMO — Part1: NN Architecture Tuning — Layers and Nodes

We limit NN to **4** hidden layers, the chromosome will be:

[n_1, n_2, n_3, n_4]

Here, ***n*** i denotes the number of nodes in the layer ***i*** from 1 to 4.
To control the number of hidden layers in NN, some of n_i may be 0 or <0 ... it means -> **no more layers** will be added to NN.

<u>Example of some chromosomes:</u>

[**10**, 20, -5, 15] -> tuple (10, 20) since -5 ends the layer count.
[**10**, 0, -5, 15] -> tuple (10,) since 0 ends the layer count.
[**10**, 20, 5, -15] -> tuple (10, 20, 5) since -15 ends the count.
[**10**, 20, 5, 15] > tuple (10, 20, 5, 15).

To **guarantee** that there is **at least 1 hidden layer**, the **1st** parameter (**10** here) is **always >0**.
The **other layer parameters** can have **varying distributions** around 0 ... why ... to control their chances of being the terminating parameters.

DEMO - Part 1:
NN Architecture Tuning Solution

Results:

DEMO 1 - Default MLP Hyperparameter values.

```
***************************
gen     nevals  max             avg
0       20      0.769841        0.284063
1       17      0.769841        0.473413
2       15      0.769841        0.606905
3       16      0.769841        0.659238
4       17      0.769841        0.673444
5       14      0.769841        0.703746
6       17      0.769841        0.739619
7       14      0.769841        0.70954
8       16      0.769841        0.686921
9       17      0.769841        0.689833
10      15      0.769841        0.680286
Time Elapsed =  82.1906521320343
Best solution is:  'hidden_layer_sizes'=(13, 4, 7)
Accuracy = 0.76984
```

Try to reproduce these results!

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- **DEMO - Part 1: NN Architecture Tuning Solution**
    - **Random Seed**
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

DEMO - Part 1: NN Architecture Tuning Solution

What is Influence of ...

- **Random Seed**
- Dataset (Wine, Iris, Breast Cancer)
- Max NN Layer Number

DEMO - Part 1: NN Architecture Tuning Solution — Various Random Seeds?

Results for various RANDOM_SEEDs:

```
# dataset = 'wine'
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.769841  0.284063
1   17      0.769841  0.473413
2   15      0.769841  0.606905
3   16      0.769841  0.659238
4   17      0.769841  0.673444
5   14      0.769841  0.703746
6   17      0.769841  0.739619
7   14      0.769841  0.70954
8   16      0.769841  0.686921
9   17      0.769841  0.689833
10  15      0.769841  0.680286
Best solution is:  'hidden_layer_sizes'=(13, 4, 7)
Accuracy =  0.76984
```

Try to reproduce these results!

DEMO - Part 1: NN Architecture Tuning Solution — Various Random Seeds?

Results for various RANDOM_SEEDs:

```
# dataset = 'wine'
# RANDOM_SEED = 666

**************************
gen nevals  max       avg
0   20      0.647937  0.31354
1   17      0.647937  0.41869
2   15      0.647937  0.478095
3   16      0.647937  0.418651
4   17      0.647937  0.503325
5   12      0.647937  0.492421
6   17      0.647937  0.435524
7   16      0.647937  0.503032
8   16      0.647937  0.466016
9   16      0.647937  0.51246
10  17      0.647937  0.572524
Time Elapsed =  93.69340062141418
Best solution is:  'hidden_layer_sizes'=(14, 3, 4, 4)
Accuracy = 0.64794
```

```
# dataset = 'wine'
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.769841  0.284063
1   17      0.769841  0.473413
2   15      0.769841  0.606905
3   16      0.769841  0.659238
4   17      0.769841  0.673444
5   14      0.769841  0.703746
6   17      0.769841  0.739619
7   14      0.769841  0.70954
8   16      0.769841  0.686921
9   17      0.769841  0.689833
10  15      0.769841  0.680286
Best solution is:  'hidden_layer_sizes'=(13, 4, 7)
Accuracy =  0.76984
```

Try to reproduce these results!

DEMO - Part 1: NN Architecture Tuning Solution — Various Random Seeds?

Results for various RANDOM_SEEDs:

```
# dataset = 'wine'
# RANDOM_SEED = 1042
***************************
gen nevals   max        avg
0   20       0.520159   0.289151
1   15       0.520159   0.322095
2   12       0.520159   0.42246
3   17       0.541587   0.419079
4   14       0.541587   0.41527
5   17       0.541587   0.471929
6   15       0.541587   0.457198
7   16       0.541587   0.472865
8   17       0.541587   0.493143
9   16       0.541587   0.45669
10  13       0.541587   0.488968
Time Elapsed =  84.34443235397339
Best solution is:  'hidden_layer_sizes'=(9, 9, 5)
Accuracy = 0.54159
```

```
# dataset = 'wine'
# RANDOM_SEED = 666

***************************
gen nevals   max        avg
0   20       0.647937   0.31354
1   17       0.647937   0.41869
2   15       0.647937   0.478095
3   16       0.647937   0.418651
4   17       0.647937   0.503325
5   12       0.647937   0.492421
6   17       0.647937   0.435524
7   16       0.647937   0.503032
8   16       0.647937   0.466016
9   16       0.647937   0.51246
10  17       0.647937   0.572524
Time Elapsed =  93.69340062141418
Best solution is:  'hidden_layer_sizes'=(14, 3, 4, 4)
Accuracy = 0.64794
```

```
# dataset = 'wine'
# RANDOM_SEED = 42

gen nevals   max        avg
0   20       0.769841   0.284063
1   17       0.769841   0.473413
2   15       0.769841   0.606905
3   16       0.769841   0.659238
4   17       0.769841   0.673444
5   14       0.769841   0.703746
6   17       0.769841   0.739619
7   14       0.769841   0.70954
8   16       0.769841   0.686921
9   17       0.769841   0.689833
10  15       0.769841   0.680286
Best solution is:  'hidden_layer_sizes'=(13, 4, 7)
Accuracy =  0.76984
```

Try to reproduce these results!

DEMO - Part 1: NN Architecture Tuning Solution — Various Random Seeds - Resume

Resume for various RANDOM_SEEDs:

For various RANDOM_SEED we can obtain NNs with the very different:

- **performance (accuracy),**
- **the number of nodes in layers,**
- **the number of layers.**

The **possible reason** is
the **stochastic** (so-called **non-gradient**) manner of **parameter change** during evolution.
There is some possibility that
all these **models** for different RANDOM_SEEDs can **reach** the different **local** (**NOT** global) the **maximum** value of **fitness function** (**accuracy** here).

Try to reproduce these results!

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- **DEMO - Part 1: NN Architecture Tuning Solution**
  - Random Seed
  - **Dataset (Wine, Iris, Breast Cancer)**
  - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

DEMO - Part 1: NN Architecture Tuning Solution

What is Influence of ...

- Random Seed
- **Dataset (Wine, Iris, Breast Cancer)**
- Max NN Layer Number

NN Tuning Example:
Wine Dataset

It is the classic example of classification problem.

UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Wine Data Set

Download: Data Folder, Data Set Description

Abstract: Using chemical analysis determine the origin of wines

Data Set Characteristics:	Multivariate	**Number of Instances:**	178	**Area:**	Physical
Attribute Characteristics:	Integer, Real	**Number of Attributes:**	13	**Date Donated**	1991-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	1602802

Source:

Original Owners:

Forina, M. et al, PARVUS -
An Extendible Package for Data Exploration, Classification and Correlation.
Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno,
16147 Genoa, Italy.

UCI Wine dataset (https://archive.ics.uci.edu/ml/datasets/wine)

NN Tuning Example:
Iris Dataset

It is the classic example of classification problem.

UCI

Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Iris Data Set

Download: Data Folder, Data Set Description

Abstract: Famous database; from Fisher, 1936

Data Set Characteristics:	Multivariate	**Number of Instances:**	150	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	4	**Date Donated**	1988-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	3847949

UCI Iris dataset (https://archive.ics.uci.edu/ml/datasets/iris)

NN Tuning Example:
Breast Cancer Dataset

It is the classic example of classification problem.

UCI

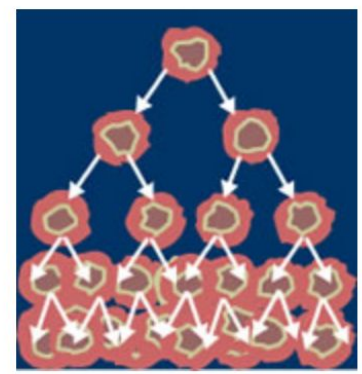
Machine Learning Repository

Center for Machine Learning and Intelligent Systems

Breast Cancer Wisconsin (Diagnostic) Data Set

Download: Data Folder, Data Set Description

Abstract: Diagnostic Wisconsin Breast Cancer Database



Data Set Characteristics:	Multivariate	**Number of Instances:**	569	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	32	**Date Donated**	1995-11-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	1444676

UCI Breast Cancer dataset
(https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic))

DEMO - Part 1: NN Architecture Tuning Solution — Various Datasets?

Results for various RANDOM_SEEDs:

```
### wine
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.769841  0.284063
1   17      0.769841  0.473413
2   15      0.769841  0.606905
3   16      0.769841  0.659238
4   17      0.769841  0.673444
5   14      0.769841  0.703746
6   17      0.769841  0.739619
7   14      0.769841  0.70954
8   16      0.769841  0.686921
9   17      0.769841  0.689833
10  15      0.769841  0.680286
- Best solution is:  'hidden_layer_sizes'=(13, 4, 7) , accuracy =  0.7698412698412699
```

Try to reproduce these results!

DEMO - Part 1: NN Architecture Tuning Solution — Various Datasets?

Results for various RANDOM_SEEDs:

```
### iris
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.666667  0.416333
1   17      0.693333  0.487
2   15      0.76      0.537333
3   14      0.76      0.550667
4   17      0.76      0.568333
5   17      0.76      0.653667
6   14      0.76      0.589333
7   15      0.76      0.618
8   16      0.866667  0.616667
9   16      0.866667  0.666333
10  16      0.866667  0.722667
- Best solution is:  'hidden_layer_sizes'=(15, 5, 8) , accuracy =  0.8666
```

```
### wine
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.769841  0.284063
1   17      0.769841  0.473413
2   15      0.769841  0.606905
3   16      0.769841  0.659238
4   17      0.769841  0.673444
5   14      0.769841  0.703746
6   17      0.769841  0.739619
7   14      0.769841  0.70954
8   16      0.769841  0.686921
9   17      0.769841  0.689833
10  15      0.769841  0.680286
- Best solution is:  'hidden_layer_sizes'=(13, 4, 7) , accuracy =  0.7698412698412699
```

Try to reproduce these results!

DEMO - Part 1: NN Architecture Tuning Solution — Various Datasets?

Results for various RANDOM_SEEDs:

```
### breast_cancer
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.927946  0.808865
1   15      0.929669  0.889953
2   15      0.929669  0.893562
3   15      0.929669  0.893683
4   16      0.934932  0.839395
5   17      0.934932  0.912204
6   14      0.934932  0.895351
7   16      0.934932  0.908839
8   17      0.934932  0.900869
9   16      0.934932  0.845574
10  15      0.934932  0.900429
- Best solution is:  'hidden_layer_sizes'=(15, 8, 10, 4) , accuracy =  0.934
```

```
### iris
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.666667  0.416333
1   17      0.693333  0.487
2   15      0.76      0.537333
3   14      0.76      0.550667
4   17      0.76      0.568333
5   17      0.76      0.653667
6   14      0.76      0.589333
7   15      0.76      0.618
8   16      0.866667  0.616667
9   16      0.866667  0.666333
10  16      0.866667  0.722667
- Best solution is:  'hidden_layer_sizes'=(15, 5, 8) , accuracy =  0.8666
```

```
### wine
# RANDOM_SEED = 42

gen nevals  max       avg
0   20      0.769841  0.284063
1   17      0.769841  0.473413
2   15      0.769841  0.606905
3   16      0.769841  0.659238
4   17      0.769841  0.673444
5   14      0.769841  0.703746
6   17      0.769841  0.739619
7   14      0.769841  0.70954
8   16      0.769841  0.686921
9   17      0.769841  0.689833
10  15      0.769841  0.680286
- Best solution is:  'hidden_layer_sizes'=(13, 4, 7) , accuracy =  0.7698412698412699
```

Try to reproduce these results!

DEMO - Part 1: NN Architecture Tuning Solution — Various Datasets - Resume

Resume for various datasets:

Again ... for various datasets we can obtain NNs with the very different:

performance (accuracy),

- **the number of nodes in layers,**
- **the number of layers.**

The **possible reason** is

more evident here:

- **different features,**

- **different number** of features,

- their **different contribution**

to fitness function (accuracy).

Try to reproduce these results!

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- **DEMO - Part 1: NN Architecture Tuning Solution**
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - **Max NN Layer Number**
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

DEMO - Part 1: NN Architecture Tuning Solution

What is Influence of ...

- Random Seed
- Dataset (Wine, Iris, Breast Cancer)
- **Max NN Layer Number**

DEMO - Part 1: NN Architecture Tuning Solution — Various MAX Layer Number?

Results for various MAX Layer Number:

BUT

...

try it as a self-guided learning

...

if you want! :)

Try to reproduce these results!

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- **DEMO - Part 2: NN Hyperparameter Tuning**
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- Resume

DEMO - Part 2: NN Hyperparameter Tuning

For the previous 1) NN Architecture Tuning
we used the default hyperparameters.
BUT ... from the previous lecture ...
tuning the various hyperparameters can
increase the classifier's performance.

Q: Can we use hyperparameter tuning here? **A:** Yes.

From *sklearn* implementation of MLP we can use numerous tunable hyperparameters:

Name	Type	Description	Default value
`activation`	`{'tanh', 'relu', 'logistic'}`	Activation function for the hidden layers	`'relu'`
`solver`	`{'sgd', 'adam', 'lbfgs'}`	The solver for weight optimization	`'adam'`
`alpha`	`float`	Regularization term parameter	`0.0001`
`learning_rate`	`{'constant', 'invscaling', 'adaptive'}`	Learning rate schedule for weight updates	`'constant'`

DEMO - Part 2: NN Hyperparameter Tuning

Like in the previous lecture demos,
a floating point-based chromosome representation allows us to combine various types of hyperparameters into GA-based optimization process.

activation - one of three values: *tanh*, *relu*, or *logistic*.
This can be achieved by representing it as a float in the range of [0, 2.99] .
To transform the float into one of the aforementioned string values, we need to apply the *floor()* function to it, which will yield either 0, 1, or 2.
Then we replace 0 -> *tanh*, 1 -> *relu*, and 2 -> *logistic*.

solver - one of 3 values: *sgd*, *adam*, or *lbfgs*.
Like for **activation**: it can be represented using a float in [0, 2.99] range.

alpha - already a float, no conversion is needed.
It will be bound to the range of [0.0001, 2.0].

learning_rate - one of 3 values: *constant*, *invscaling*, *adaptive*.
Like for **activation**: it can be represented using a float in [0, 2.99] range.

DEMO - Part 2: NN Hyperparameter Tuning

Results:

DEMO 2 - The best NN Architecture from DEMO 1.

HIDDEN_LAYER_SIZES = [13, 4, 7]

```
***************************
gen     nevals  max             avg
0       20      0.946667        0.362
1       15      0.946667        0.599667
2       16      0.946667        0.864333
3       16      0.946667        0.927333
4       17      0.946667        0.944667
5       14      0.946667        0.887
6       15      0.946667        0.944667
7       14      0.946667        0.946
8       16      0.946667        0.907667
9       15      0.946667        0.945
10      17      0.946667        0.946
Time Elapsed =  92.3740484714508
Best solution is:  'activation'='logistic'
 'solver'='lbfgs'
 'alpha'=0.1713983879055075
 'learning_rate'='invscaling'
Accuracy = 0.94667
```

Try to reproduce these results!

Content

- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- **DEMO - Part 3: NN Architecture + NN Hyperparameter Tuning Solution**
- Resume

DEMO - Part 3: Hybrid: NN Architecture + NN Hyperparameter Tuning

The first 4 ranges for NN Arhictecture tuning
–> one for each hidden layer.

The next 4 ranges
-> represent the additional 4 hyperparameters.

```
# 'hidden_layer_sizes': first four values
# 'activation':    0..2.99
# 'solver':        0..2.99
# 'alpha':    0.0001..2.0
# 'learning_rate': 0..2.99
BOUNDS_LOW =  [ 5,  -5, -10, -20, 0,     0,     0.0001, 0    ]
BOUNDS_HIGH = [15,  10,  10,  10, 2.999, 2.999, 2.0,    2.999]
```

DEMO - Part 3: Hybrid: NN Architecture + NN Hyperparameter Tuning

Input Boundaries:

```
# boundaries for all parameters:
# 'hidden_layer_sizes': first four values
# 'activation': ['tanh', 'relu', 'logistic'] -> 0, 1, 2
# 'solver': ['sgd', 'adam', 'lbfgs'] -> 0, 1, 2
# 'alpha': float in the range of [0.0001, 2.0],
# 'learning_rate': ['constant', 'invscaling', 'adaptive'] -> 0, 1, 2
BOUNDS_LOW =  [ 5,  -5, -10, -20, 0,     0,     0.0001, 0    ]
BOUNDS_HIGH = [15,  10,  10,  10, 2.999, 2.999, 2.0,    2.999]
```

DEMO - Part 3: Hybrid: NN Architecture + NN Hyperparameter Tuning

Results:

```
***************************
gen     nevals  max     avg
0       20      0.94    0.448
1       16      0.94    0.633
2       15      0.94    0.737667
3       16      0.946667        0.842
4       17      0.946667        0.889667
5       15      0.946667        0.937667
6       16      0.946667        0.939
7       16      0.946667        0.875
8       16      0.946667        0.876333
9       14      0.946667        0.942333
10      16      0.946667        0.902667
Time Elapsed =  83.84976434707642
Best solution is:  'hidden_layer_sizes'=(8, 7)
 'activation'='relu'
 'solver'='lbfgs'
 'alpha'=0.563775972907702
 'learning_rate'='adaptive'
Accuracy = 0.94667
```

Try to reproduce these results!

DEMO - Part 3: Hybrid: NN Architecture + NN Hyperparameter Tuning

Results -> Compare with NN Hyper ONLY

```
*************************
gen     nevals  max     avg
0       20      0.94    0.448
1       16      0.94    0.633
2       15      0.94    0.737667
3       16      0.946667        0.842
4       17      0.946667        0.889667
5       15      0.946667        0.937667
6       16      0.946667        0.939
7       16      0.946667        0.875
8       16      0.946667        0.876333
9       14      0.946667        0.942333
10      16      0.946667        0.902667
Time Elapsed =  83.84976434707642
Best solution is:  'hidden_layer_sizes'=(8, 7)
 'activation'='relu'
 'solver'='lbfgs'
 'alpha'=0.563775972907702
 'learning_rate'='adaptive'
Accuracy = 0.94667
```

```
*************************
gen     nevals  max             avg
0       20      0.946667        0.362
1       15      0.946667        0.599667
2       16      0.946667        0.864333
3       16      0.946667        0.927333
4       17      0.946667        0.944667
5       14      0.946667        0.887
6       15      0.946667        0.944667
7       14      0.946667        0.946
8       16      0.946667        0.907667
9       15      0.946667        0.945
10      17      0.946667        0.946
Time Elapsed =  92.3740484714508
Best solution is:  'activation'='logistic'
 'solver'='lbfgs'
 'alpha'=0.17139833879055075
 'learning_rate'='invscaling'
Accuracy = 0.94667
```

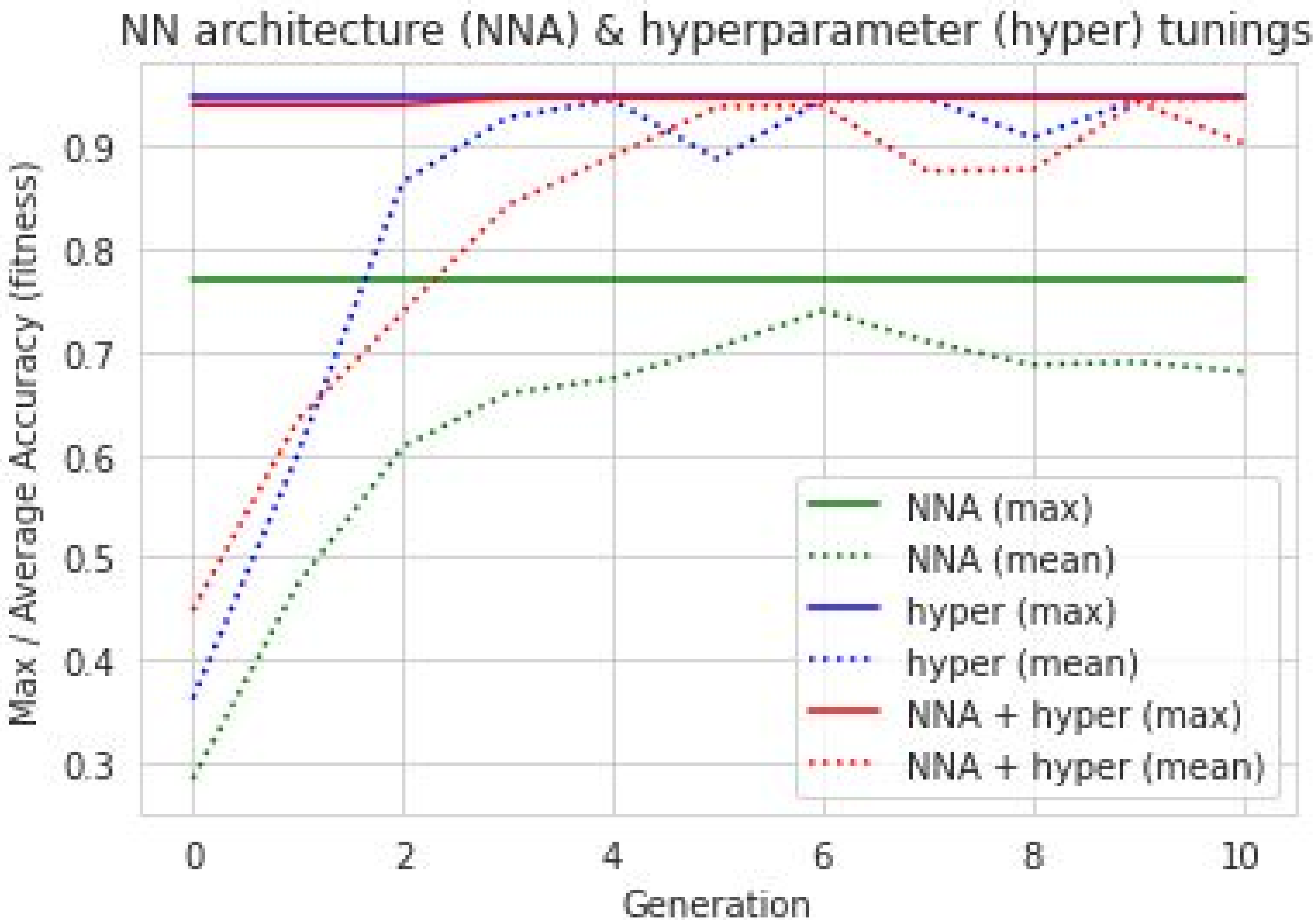
Funny Differences in Best Solution Parameters ... :)

Try to reproduce these results!

Content

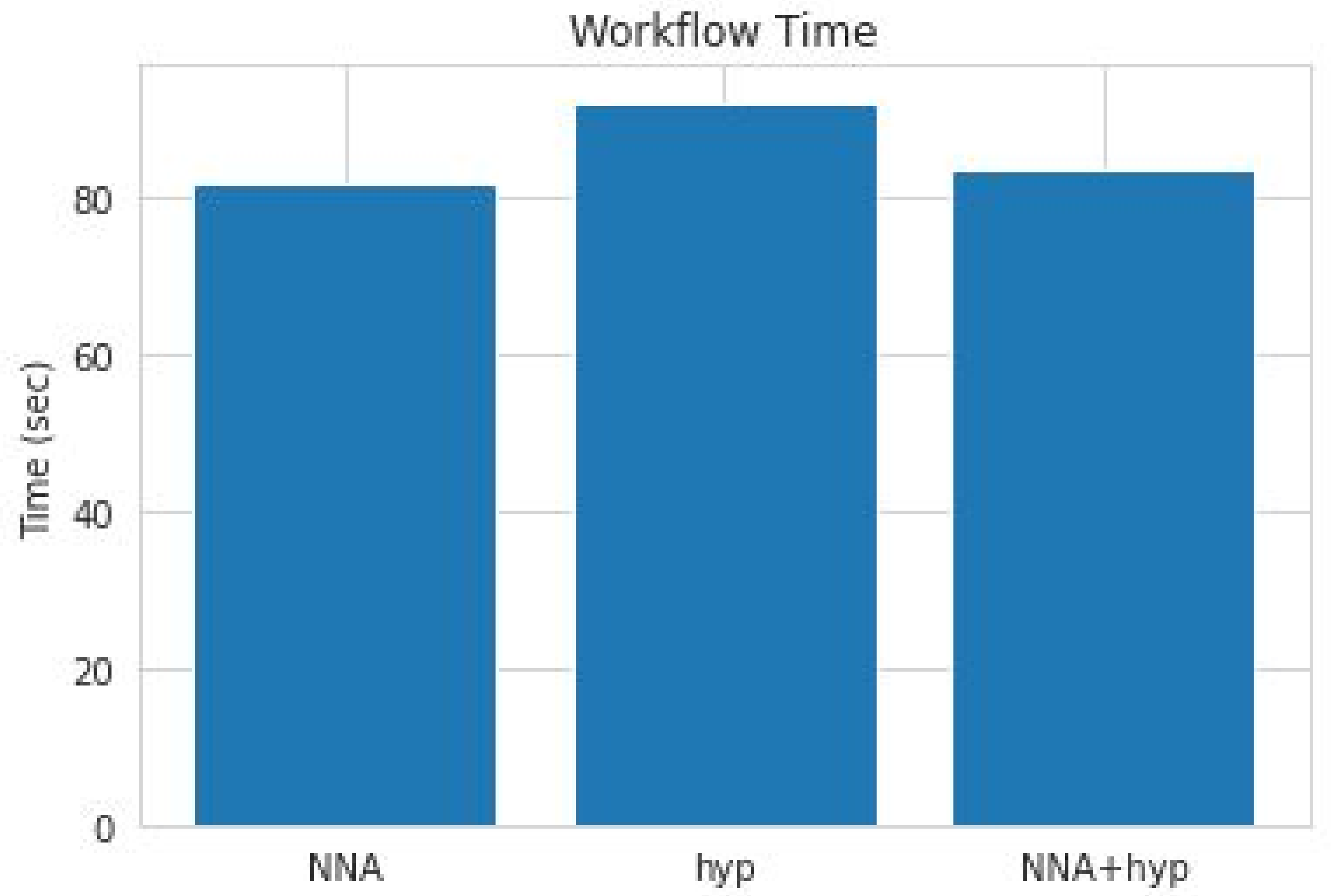
- Recommended Sources
- EA (GA) for Neural Network (NN) Tuning
- Types of NN Tuning
- Classification Problem Example: Dataset + Workflow
- DEMO - Part 1: NN Architecture Tuning Solution
    - Random Seed
    - Dataset (Wine, Iris, Breast Cancer)
    - Max NN Layer Number
- DEMO - Part 2: NN Hyperparameter Tuning Solution
- DEMO - Part 3: NN Architecture + Hyperparameter Tuning Solution
- **Resume**

EC for NN Architecture + NN Hyperparameter Tuning — Comparative Plot — Accuracy



Try to reproduce these results!

EC for NN Architecture + NN Hyperparameter Tuning — Comparative Plot — Time



It does not matter ... in such problem formulation ... but ...

Try to reproduce these results!

EC for Feature Selection — Resume

EC (GA) can be effectively applied to
the classic supervised machine learning problems:

– **regression** (use case of Friedman-1 Regression Problem)
and
– **classification** (use case of UCI-dataset animal classification)

for
– **feature selection**
or
– **dimensionality reduction**

with the purpose of:
– **decrease of MSE**
or
– **increase of mean accuracy**.

Evolutionary Algorithms (EA) Basics

Lecture 6 - DEMO A - OpenAI Gym platform

based on (C) OpenAI, Heaton, Moore, Varoquaux, Grobler, Wirsansky work

Brief Content:

- OpenAI Gym platform
- *Reinforcement Learning (RL) problems:*
    - MountainCar-v0,
    - MountainCarContinuous-v0,
    - CartPole-v1
    - ...
- Functions to visualize Gym-game-scenarios in Colab.

Installation and import of libraries

Library to support RL algorithms

Gym is a toolkit for developing and comparing reinforcement learning algorithms.

It supports teaching agents everything from walking to playing games like Pong or Pinball.

```
! pip install gym
```

```
Requirement already satisfied: gym in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/pythor
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/py
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
```

Libraries to Render OpenAI Gym Environments in Colab

It is possible to visualize the activities performed in Gym (game your agent is playing), even on Colab. This section provides information on how to generate a video in Colab that shows you an episode of the game your agent is playing.

```
%%time
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
```

```
CPU times: user 28.9 ms, sys: 18.1 ms, total: 47 ms
Wall time: 11.6 s
```

```
%%time
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
```

```
Collecting setuptools
  Downloading https://files.pythonhosted.org/packages/60/6a/dd9533a
     |████████████████████████████████| 788kB 7.3MB/s
ERROR: datascience 0.10.6 has requirement folium==0.2.1, but you'll
Installing collected packages: setuptools
  Found existing installation: setuptools 54.0.0
    Uninstalling setuptools-54.0.0:
      Successfully uninstalled setuptools-54.0.0
Successfully installed setuptools-54.1.1
CPU times: user 73.7 ms, sys: 44.5 ms, total: 118 ms
```

IMPORTANT: you should restart runtime!

Part 1.Introduction to the OpenAI Gym

Gym - Advanages and Limitations

OpenAI Gym aims to provide an easy-to-setup general-intelligence benchmark with a wide variety of differentt environments. The goal is to standardize how environments are defined in AI research publications so that published research becomes more easily reproducible. The project claims to provide the user with a simple interface.

OpenAI gym is pip-installed onto your local machine. There are a few significant limitations to be aware of:

- developers can only use Gym with Python (as of June 2017).

- OpenAI Gym can not directly render animated games in Google CoLab.

Because OpenAI Gym requires a graphics display, the only way to display Gym in Google CoLab is an embedded video. The presentation of OpenAI Gym game animations in Google CoLab is discussed later in this module.

Gym - Leaderboard

The OpenAI Gym does have a leaderboard, similar to Kaggle; however, the OpenAI Gym's leaderboard is much more informal compared to Kaggle. The user's local machine performs all scoring. As a result, the OpenAI gym's leaderboard is strictly an "honor's system." The leaderboard is maintained the following GitHub repository:

- OpenAI Gym Leaderboard

If you submit a score, you are required to provide a writeup with sufficient instructions to reproduce your result. A video of your results is suggested, but not required.

Gym - Environments

The centerpiece of Gym is the environment, which defines the "game" in which your reinforcement algorithm will compete. An environment does not need to be a game; however, it describes the following game-like features:

- **action space**: What actions can we take on the environment, at each step/episode, to alter the environment.
- **observation space**: What is the current state of the portion of the environment that we can observe. Usually, we can see the entire environment.

Gym - Basic Termonology

- **Agent** - The machine learning program or model that controls the actions. Step - One round of issuing actions that affect the observation space.
- **Episode** - A collection of steps that terminates when the agent fails to meet the environment's objective, or the episode reaches the maximum number of allowed steps.
- **Render** - Gym can render one frame for display after each episode.
- **Reward** - A positive reinforcement that can occur at the end of each episode, after the agent acts.
- **Nondeterministic** - For some environments, randomness is a factor in deciding what effects actions have on reward and changes to the observation space.

It is important to note that many of the gym environments specify that they are not nondeterministic even though they make use of random numbers to process actions. It is generally agreed upon (based on the gym GitHub issue tracker) that nondeterministic property means that a deterministic environment will still behave randomly even when given consistent seed value. The seed method of an environment can be used by the program to seed the random number generator for the environment.

Environment - Attributes

The Gym library allows us to query some of these attributes from environments. I created the following function to query gym environments.

```
import gym

def query_environment(name):
  env = gym.make(name)
  spec = gym.spec(name)
  print(f"Action Space: {env.action_space}")
  print(f"Observation Space: {env.observation_space}")
  print(f"Max Episode Steps: {spec.max_episode_steps}")
  print(f"Nondeterministic: {spec.nondeterministic}")
  print(f"Reward Range: {env.reward_range}")
  print(f"Reward Threshold: {spec.reward_threshold}")
```

Environment - Examples:

- MountainCar-v0,
- MountainCarContinuous-v0,
- CartPole-v1
- ...

MountainCar-v0

We will begin by looking at the MountainCar-v0 environment, which challenges an underpowered car to escape the valley between two mountains. The following code describes the Mountian Car environment.

```
query_environment("MountainCar-v0")
```

```
Action Space: Discrete(3)
Observation Space: Box(-1.2000000476837158, 0.6000000238418579, (2,), float32
Max Episode Steps: 200
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: -110.0
```

Actions

There are three distinct actions that can be taken:

- accelerate forward,
- decelerate,
- accelerate backwards.

Observation space

The observation space contains two continuous (floating point) values, as evident by the box object.

The observation space contains:

- the position and
- velocity of the car.

The car has 200 steps to escape for each episode.

Reward: The mountian car recieves NO incremental reward. The only reward for the car is given when it escapes the valley.

MountainCarContinuous-v0

There is also a continuous variant of the mountain car. This version does not simply have the motor on or off. For the continuous car the action space is a single floating point number that specifies how much forward or backward force is being applied.

```
query_environment("MountainCarContinuous-v0")
```

```
Action Space: Box(-1.0, 1.0, (1,), float32)
Observation Space: Box(-1.2000000476837158, 0.6000000238418579, (2,), float32
Max Episode Steps: 999
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 90.0
```

Note: ignore the warning above, it is a relativly inconsequential bug in OpenAI Gym.

CartPole-v1

The CartPole-v1 environment challenges the agent to move a cart while keeping a pole balanced.

Observation space

The environment has an observation space of 4 continuous numbers:

- Cart Position
- Cart Velocity
- Pole Angle
- Pole Velocity At Tip

Actions

To achieve this goal, the agent can take the following actions:

- Push cart to the left
- Push cart to the right

```
query_environment("CartPole-v1")
```

```
Action Space: Discrete(2)
Observation Space: Box(-3.4028234663852886e+38, 3.4028234663852886e+38, (4,),
Max Episode Steps: 500
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 475.0
```

Breakout-v0

Atari games, like breakout can use an observation space that is either equal to the size of the Atari screen (210x160) or even use the RAM memory of the Atari (128 bytes) to determine the state of the game. Yes thats bytes, not kilobytes!

```
query_environment("Breakout-v0")
```

```
Action Space: Discrete(4)
Observation Space: Box(0, 255, (210, 160, 3), uint8)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

Breakout-ram-v0

```
query_environment("Breakout-ram-v0")
```

```
Action Space: Discrete(4)
Observation Space: Box(0, 255, (128,), uint8)
Max Episode Steps: 10000
Nondeterministic: False
```

```
Reward Range: (-inf, inf)
Reward Threshold: None
```

Atlantis-v0

```
query_environment("Atlantis-v0")
```

```
Action Space: Discrete(4)
Observation Space: Box(0, 255, (210, 160, 3), uint8)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

Part 2.Functions to visualize Gym-game-scenarios in Colab

Next we define functions used to show the video by adding it to the Colab notebook.

```
import gym
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
Utility functions to enable video recording of gym environment
and displaying it.
To enable video, just do "env = wrap_env(env)""
"""

def show_video():
  mp4list = glob.glob('video/*.mp4')
  if len(mp4list) > 0:
    mp4 = mp4list[0]
    video = io.open(mp4, 'r+b').read()
    encoded = base64.b64encode(video)
    ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                loop controls style="height: 400px;">
                <source src="data:video/mp4;base64,{0}" type="video/mp4" />
             </video>'''.format(encoded.decode('ascii'))))
  else:
    print("Could not find video")


def wrap_env(env):
```

```
  env = Monitor(env, './video', force=True)
  return env
```

Now we are ready to play the game. We use a simple random agent.

MountainCar-v0

```
env = wrap_env(gym.make("MountainCar-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

MountainCarContinuous-v0

```
env = wrap_env(gym.make("MountainCarContinuous-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

CartPole-v1

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

observation = env.reset()

while True:

    env.render()
```

```
    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Breakout-v0

```
env = wrap_env(gym.make("Breakout-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
```

```
      break;

env.close()
show_video()
```

Breakout-ram-v0

```
env = wrap_env(gym.make("Breakout-ram-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Atlantis-v0

```
env = wrap_env(gym.make("Atlantis-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Lecture 7 - Applications of EA for Reinforcement Learning

based on (C) OpenAI, Heaton, Moore, Varoquaux, Grobler, Wirsansky work

Brief Content:

- DEAP installation (**every time after start of Colab VM!**),
- components needed for the GA workflow,
- *Reinforcement Learning (RL) problems:*
    - MountainCar-v0,
    - MountainCarContinuous-v0,
    - CartPole-v1
    - ...
- performance comparison (accuracy and run time).

By the end of this lecture you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the *Reinforcement Learning* problem using a GA-based solutions for search of solutions,
- how to experiment with various settings of the GA and interpret the differences in the results.

Get Figures for Text Description

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

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

```
! cp -r /content/drive/MyDrive/COLAB_EVO/EVO_Lecture07_CartPole/figures .
! ls figures
```

```
MLPRegressor.png
```

Installation and import of libraries

```
! pip install deap
```

```
Collecting deap
  Downloading https://files.pythonhosted.org/packages/99/d1/803c7a387d8a7e686
    |████████████████████████████████| 163kB 6.1MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
Installing collected packages: deap
Successfully installed deap-1.3.1
```

Library to support RL algorithms

Gym is a toolkit for developing and comparing reinforcement learning algorithms.

It supports teaching agents everything from walking to playing games like Pong or Pinball.

```
! pip install gym
```

```
Requirement already satisfied: gym in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/py
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/python
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
```

Libraries to Render OpenAI Gym Environments in Colab

It is possible to visualize the activities performed in Gym (game your agent is playing), even on Colab. This section provides information on how to generate a video in Colab that shows you an episode of the game your agent is playing.

```
%%time
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
```

```
CPU times: user 47 ms, sys: 13 ms, total: 59.9 ms
Wall time: 14.1 s
```

```
%%time
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
```

```
Collecting setuptools
  Downloading https://files.pythonhosted.org/packages/60/6a/dd9533ae9367a1f35
     |████████████████████████████████| 788kB 4.2MB/s
ERROR: datascience 0.10.6 has requirement folium==0.2.1, but you'll have foli
Installing collected packages: setuptools
  Found existing installation: setuptools 54.0.0
    Uninstalling setuptools-54.0.0:
      Successfully uninstalled setuptools-54.0.0
Successfully installed setuptools-54.1.1
CPU times: user 94.1 ms, sys: 46.3 ms, total: 140 ms
```

IMPORTANT: you should restart runtime!

Part 1.Introduction to the OpenAI Gym

Gym - Advanages and Limitations

OpenAI Gym aims to provide an easy-to-setup general-intelligence benchmark with a wide variety of differentt environments. The goal is to standardize how environments are defined in AI research publications so that published research becomes more easily reproducible. The project claims to provide the user with a simple interface.

OpenAI gym is pip-installed onto your local machine. There are a few significant limitations to be aware of:

- developers can only use Gym with Python (as of June 2017).
- OpenAI Gym can not directly render animated games in Google CoLab.

Because OpenAI Gym requires a graphics display, the only way to display Gym in Google CoLab is an embedded video. The presentation of OpenAI Gym game animations in Google CoLab is discussed later in this module.

Gym - Leaderboard

The OpenAI Gym does have a leaderboard, similar to Kaggle; however, the OpenAI Gym's leaderboard is much more informal compared to Kaggle. The user's local machine performs all scoring. As a result, the OpenAI gym's leaderboard is strictly an "honor's system." The leaderboard is maintained the following GitHub repository:

- OpenAI Gym Leaderboard

If you submit a score, you are required to provide a writeup with sufficient instructions to reproduce your result. A video of your results is suggested, but not required.

Gym - Environments

The centerpiece of Gym is the environment, which defines the "game" in which your reinforcement algorithm will compete. An environment does not need to be a game; however, it describes the following game-like features:

- **action space**: What actions can we take on the environment, at each step/episode, to alter the environment.
- **observation space**: What is the current state of the portion of the environment that we can observe. Usually, we can see the entire environment.

Gym - Basic Termonology

- **Agent** - The machine learning program or model that controls the actions. Step - One round of issuing actions that affect the observation space.
- **Episode** - A collection of steps that terminates when the agent fails to meet the environment's objective, or the episode reaches the maximum number of allowed steps.
- **Render** - Gym can render one frame for display after each episode.
- **Reward** - A positive reinforcement that can occur at the end of each episode, after the agent acts.
- **Nondeterministic** - For some environments, randomness is a factor in deciding what effects actions have on reward and changes to the observation space.

It is important to note that many of the gym environments specify that they are not nondeterministic even though they make use of random numbers to process actions. It is generally agreed upon (based on the gym GitHub issue tracker) that nondeterministic property means that a deterministic environment will still behave randomly even when given consistent seed value. The seed method of an environment can be used by the program to seed the random number generator for the environment.

Environment - Attributes

The Gym library allows us to query some of these attributes from environments. I created the following function to query gym environments.

```
import gym

def query_environment(name):
  env = gym.make(name)
  spec = gym.spec(name)
  print(f"Action Space: {env.action_space}")
  print(f"Observation Space: {env.observation_space}")
  print(f"Max Episode Steps: {spec.max_episode_steps}")
  print(f"Nondeterministic: {spec.nondeterministic}")
```

```
  print(f"Reward Range: {env.reward_range}")
```

Environment - Examples:

- MountainCar-v0,
- MountainCarContinuous-v0,
- CartPole-v1
- ...

MountainCar-v0

We will begin by looking at the MountainCar-v0 environment, which challenges an underpowered car to escape the valley between two mountains. The following code describes the Mountian Car environment.

```
query_environment("MountainCar-v0")
```

```
Action Space: Discrete(3)
Observation Space: Box(-1.2000000476837158, 0.6000000238418579, (2,), float32
Max Episode Steps: 200
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: -110.0
```

Actions

There are three distinct actions that can be taken:

- accelerate forward,
- decelerate,
- accelerate backwards.

Observation space

The observation space contains two continuous (floating point) values, as evident by the box object.

The observation space contains:

- the position and
- velocity of the car.

The car has 200 steps to escape for each episode.

Reward: The mountian car recieves NO incremental reward. The only reward for the car is given when it escapes the valley.

MountainCarContinuous-v0

There is also a continuous variant of the mountain car. This version does not simply have the motor on or off. For the continuous car the action space is a single floating point number that specifies how much forward or backward force is being applied.

```
query_environment("MountainCarContinuous-v0")
```

```
Action Space: Box(-1.0, 1.0, (1,), float32)
Observation Space: Box(-1.2000000476837158, 0.6000000238418579, (2,), float32
Max Episode Steps: 999
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 90.0
```

Note: ignore the warning above, it is a relativly inconsequential bug in OpenAI Gym.

CartPole-v1

The CartPole-v1 environment challenges the agent to move a cart while keeping a pole balanced.

Observation space

The environment has an observation space of 4 continuous numbers:

- Cart Position
- Cart Velocity
- Pole Angle
- Pole Velocity At Tip

Actions

To achieve this goal, the agent can take the following actions:

- Push cart to the left
- Push cart to the right

```
query_environment("CartPole-v1")
```

```
Action Space: Discrete(2)
Observation Space: Box(-3.4028234663852886e+38, 3.4028234663852886e+38, (4,),
Max Episode Steps: 500
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 475.0
```

Breakout-v0

Atari games, like breakout can use an observation space that is either equal to the size of the Atari screen (210x160) or even use the RAM memory of the Atari (128 bytes) to determine the state of the game. Yes thats bytes, not kilobytes!

```
query_environment("Breakout-v0")
```

```
Action Space: Discrete(4)
Observation Space: Box(0, 255, (210, 160, 3), uint8)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

Breakout-ram-v0

```
query_environment("Breakout-ram-v0")
```

```
Action Space: Discrete(4)
Observation Space: Box(0, 255, (128,), uint8)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

Atlantis-v0

```
query_environment("Atlantis-v0")
```

```
Action Space: Discrete(4)
Observation Space: Box(0, 255, (210, 160, 3), uint8)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

Functions to visualize game-scenarios in Colab

Next we define functions used to show the video by adding it to the Colab notebook.

```
import gym
from gym.wrappers import Monitor
import glob
import io
```

```
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
Utility functions to enable video recording of gym environment
and displaying it.
To enable video, just do "env = wrap_env(env)""
"""

def show_video():
  mp4list = glob.glob('video/*.mp4')
  if len(mp4list) > 0:
    mp4 = mp4list[0]
    video = io.open(mp4, 'r+b').read()
    encoded = base64.b64encode(video)
    ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                loop controls style="height: 400px;">
                <source src="data:video/mp4;base64,{0}" type="video/mp4" />
             </video>'''.format(encoded.decode('ascii'))))
  else:
    print("Could not find video")


def wrap_env(env):
  env = Monitor(env, './video', force=True)
  return env
```

Now we are ready to play the game. We use a simple random agent.

MountainCar-v0

```
env = wrap_env(gym.make("MountainCar-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
```

```
      break;

env.close()
show_video()
```

0:00 / 0:06

MountainCarContinuous-v0

```
env = wrap_env(gym.make("MountainCarContinuous-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

0:00 / 0:33

CartPole-v1

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

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Breakout-v0

```
env = wrap_env(gym.make("Breakout-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Breakout-ram-v0

```
env = wrap_env(gym.make("Breakout-ram-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

0:00 / 0:08

Atlantis-v0

```
env = wrap_env(gym.make("Atlantis-v0"))
```

```
observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Part 3. GA Solution for RL problem - CartPole-v1

CartPole-v1 - Problem Description

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over.

Reward A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson:

AG Barto, RS Sutton and CW Anderson, *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem*, IEEE Transactions on Systems, Man, and Cybernetics, 1983.

Cited in 4062 sources

Let's try it as a self-guided learning!

Use the following CartPole-v1 resources:

- description at Gym,
- Python-codes at github

Import Python libraries

In these and other lectures, we will use various Python packages:

- NumPy
- Matplotlib
- Seaborn

They are already pre-installed in Colab. Let's import them by the following code.

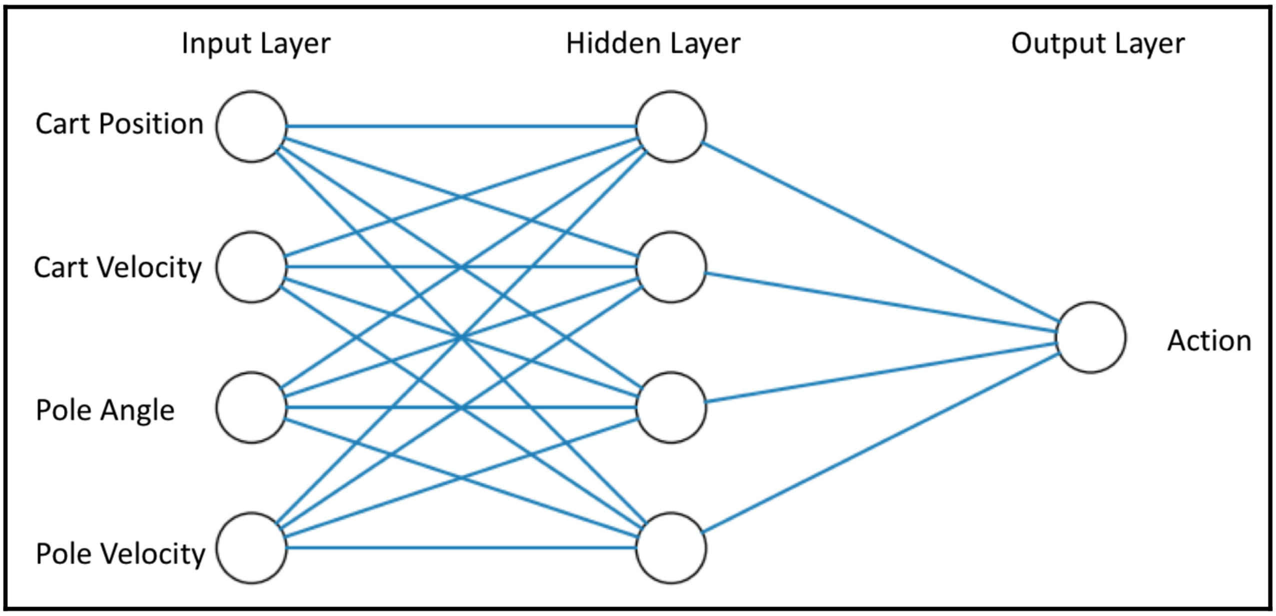
```
import gym
import time
import pickle
import random
import numpy

# for plotting
import matplotlib.pyplot as plt
import seaborn as sns
```

```
! rm -r ./video
! rm ./*pickle
```

```
rm: cannot remove './video': No such file or directory
rm: cannot remove './*pickle': No such file or directory
```

Actors - CartPole

```
from IPython.display import Image
Image('./figures/MLPRegressor.png')
```




```
import gym
import time

import numpy as np
import pickle

from sklearn.neural_network import MLPRegressor
```

```
from sklearn.exceptions import ConvergenceWarning
from sklearn.utils.testing import ignore_warnings

INPUTS = 4
HIDDEN_LAYER = 4
OUTPUTS = 1

class CartPole:

    def __init__(self, randomSeed=None):

        #self.env = gym.make('CartPole-v1')
        self.env = wrap_env(gym.make('CartPole-v1'))
        self.env.seed(randomSeed)

        if randomSeed is not None:
            self.env.seed(randomSeed)

    def __len__(self):
        return INPUTS * HIDDEN_LAYER + HIDDEN_LAYER * OUTPUTS + HIDDEN_LAYER + OUT

    @ignore_warnings(category=ConvergenceWarning)
    def initMlp(self, netParams):
        """
        initializes a MultiLayer Perceptron (MLP) Regressor with the desired netwo
        and network parameters (weights and biases).
        :param netParams: a list of floats representing the network parameters (we
        :return: initialized MLP Regressor
        """

        # create the initial MLP:
        mlp = MLPRegressor(hidden_layer_sizes=(HIDDEN_LAYER,), max_iter=1)

        # This will initialize input and output layers, and nodes weights and bias
        # we are not otherwise interested in training the MLP here, hence the sett
        mlp.fit(np.random.uniform(low=-1, high=1, size=INPUTS).reshape(1, -1), np.

        # weights are represented as a list of 2 ndarrays:
        # - hidden layer weights: INPUTS x HIDDEN_LAYER
        # - output layer weights: HIDDEN_LAYER x OUTPUTS
        numWeights = INPUTS * HIDDEN_LAYER + HIDDEN_LAYER * OUTPUTS
        weights = np.array(netParams[:numWeights])
        mlp.coefs_ = [
            weights[0:INPUTS * HIDDEN_LAYER].reshape((INPUTS, HIDDEN_LAYER)),
            weights[INPUTS * HIDDEN_LAYER:].reshape((HIDDEN_LAYER, OUTPUTS))
        ]

        # biases are represented as a list of 2 ndarrays:
        # - hidden layer biases: HIDDEN_LAYER x 1
        # - output layer biases: OUTPUTS x 1
        biases = np.array(netParams[numWeights:])
        mlp.intercepts_ = [biases[:HIDDEN_LAYER], biases[HIDDEN_LAYER:]]

        return mlp
```

```
    def getScore(self, netParams):
        """
        calculates the score of a given solution, represented by the list of float
        by creating a corresponding MLP Regressor, initiating an episode of the Ca
        running it with the MLP controlling the actions, while using the observati
        Higher score is better.
        :param netParams: a list of floats representing the network parameters (we
        :return: the calculated score value
        """

        mlp = self.initMlp(netParams)

        self.env.reset()

        actionCounter = 0
        totalReward = 0
        observation = self.env.reset()
        action = int(mlp.predict(observation.reshape(1, -1)) > 0)

        while True:
            actionCounter += 1
            observation, reward, done, info = self.env.step(action)
            totalReward += reward

            if done:
                break
            else:
                action = int(mlp.predict(observation.reshape(1, -1)) > 0)
                #print(action)

        return totalReward

    def saveParams(self, netParams):
        """
        serializes and saves a list of network parameters using pickle
        :param netParams: a list of floats representing the network parameters (we
        """
        savedParams = []
        for param in netParams:
            savedParams.append(param)

        pickle.dump(savedParams, open("cart-pole-data.pickle", "wb"))

    def replayWithSavedParams(self):
        """
        deserializes a saved list of network parameters and uses it to replay an e
        """
        savedParams = pickle.load(open("cart-pole-data.pickle", "rb"))
        self.replay(savedParams)

    def replay(self, netParams):
        """
        renders the environment and uses the given network parameters to replay an
        :param netParams: a list of floats representing the network parameters (we
        """
```

```
        mlp = self.initMlp(netParams)

        self.env.render()

        actionCounter = 0
        totalReward = 0
        observation = self.env.reset()
        action = int(mlp.predict(observation.reshape(1, -1)) > 0)

        while True:
            actionCounter += 1
            self.env.render()
            observation, reward, done, info = self.env.step(action)
            totalReward += reward

            print(actionCounter, ": --------------------------")
            print("action = ", action)
            print("observation = ", observation)
            print("reward = ", reward)
            print("totalReward = ", totalReward)
            print("done = ", done)
            print()

            if done:
                break
            else:
                time.sleep(0.03)
                action = int(mlp.predict(observation.reshape(1, -1)) > 0)

        self.env.close()

    def replayVideo(self):
        #self.env.close()
        show_video()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:144: Futu
  warnings.warn(message, FutureWarning)
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
cartPole = CartPole(RANDOM_SEED)

NUM_OF_PARAMS = len(cartPole)
# boundaries for layer size parameters:

# weight and bias values are bound between -1 and 1:
BOUNDS_LOW, BOUNDS_HIGH = -1.0, 1.0  # boundaries for all dimensions
```

GA Solution

```
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
```

```
# Genetic Algorithm constants:
POPULATION_SIZE = 100
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5   # probability for mutating an individual
MAX_GENERATIONS = 40
HALL_OF_FAME_SIZE = 3
CROWDING_FACTOR = 10.0  # crowding factor for crossover and mutation
```

Genetic Tools

```
toolbox = base.Toolbox()

# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)

# helper function for creating random real numbers uniformly distributed within a
# it assumes that the range is the same for every dimension
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] * NUM_OF_PARAMS, [up] * NUM

# create an operator that randomly returns a float in the desired range:
toolbox.register("attrFloat", randomFloat, BOUNDS_LOW, BOUNDS_HIGH)

# create an operator that fills up an Individual instance:
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attrFloat)

# create an operator that generates a list of individuals:
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
                 toolbox.individualCreator)



# fitness calculation using the CrtPole class:
def score(individual):
    return cartPole.getScore(individual),
```

```
toolbox.register("evaluate", score)


# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

```
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
```

Elitism Tools

```
def eaSimpleWithElitism(population, toolbox, cxpb, mutpb, ngen, stats=None,
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm, with the modificati
    halloffame is used to implement an elitism mechanism. The individuals containe
    halloffame are directly injected into the next generation and are not subject
    genetic operators of selection, crossover and mutation.
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is None:
        raise ValueError("halloffame parameter must not be empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
```

```
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) - hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the logbook
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)

    return population, logbook
```

GA Workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
```

```
                                                       toolbox,
                                                       cxpb=P_CROSSOVER,
                                                       mutpb=P_MUTATION,
                                                       ngen=MAX_GENERATIONS,
                                                       stats=stats,
                                                       halloffame=hof,
                                                       verbose=True)
end = time.time()
time_NNA = end - start
print("Time Elapsed = ", time_NNA)
```

```
*************************
gen     nevals  max     avg
0       100     500     18.35
1       94      500     20.6
2       93      500     23.76
3       94      500     38.88
4       93      500     48.58
5       93      500     50.68
6       92      500     62.93
7       93      500     47.7
8       92      500     59.98
9       94      500     75.68
10      95      500     55.81
11      85      500     76.89
12      93      500     79.83
13      92      500     73.48
14      85      500     66.83
15      96      500     69.02
16      94      500     98.28
17      95      500     91.89
18      93      500     92.16
19      93      500     100.52
20      90      500     112.75
21      89      500     112.59
22      92      500     149.5
23      95      500     177.98
24      96      500     228.95
25      89      500     239.97
26      91      500     298.21
27      91      500     277.71
28      95      500     356.53
29      93      500     426.83
30      94      500     426.92
31      93      500     371.19
32      92      500     415.99
33      91      500     447.65
34      90      500     431.27
35      94      500     445.97
36      91      500     448.41
37      91      500     449.5
38      88      500     470.01
39      93      500     473.17
40      90      500     460.75
Time Elapsed =  102.10462594032288
```

```
# print best solution found:
best = hof.items[0]
```

```
print("Best solution: ", best)
print("Best FitnessMax = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA, meanFitnessValues_GA = logbook.select("max", "avg")
print('History of maxFitnessValues_GA =',minFitnessValues_GA)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA)

# save best solution for a replay:
#car.saveActions(best)
cartPole.saveParams(best)
```

```
Best solution:  [0.9039409992284728, 0.06655498740733878, -0.669913677453291
Best FitnessMax = 500.00000
History of maxFitnessValues_GA = [500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 5
History of meanFitnessValues_GA = [18.35, 20.6, 23.76, 38.88, 48.58, 50.68, 6
```

```
# Replay the best solution - TEXT version
#car.replaySavedActions()
cartPole.replayWithSavedParams()
```

```
1 : -------------------------
action =  1
observation =  [ 0.00669915  0.22729017  0.02071759 -0.2462488 ]
reward =  1.0
totalReward =  1.0
done =  False

2 : -------------------------
action =  0
observation =  [0.01124495 0.03187854 0.01579262 0.05289627]
reward =  1.0
totalReward =  2.0
done =  False

3 : -------------------------
action =  1
observation =  [ 0.01188253  0.22677053  0.01685054 -0.23476242]
reward =  1.0
totalReward =  3.0
done =  False

4 : -------------------------
action =  0
observation =  [0.01641794 0.03141193 0.01215529 0.06318771]
reward =  1.0
totalReward =  4.0
done =  False

5 : -------------------------
action =  1
observation =  [ 0.01704617  0.22635751  0.01341905 -0.2256355 ]
reward =  1.0
totalReward =  5.0
done =  False
```

```
6 : -------------------------
action =  0
observation =  [0.02157332 0.03104637 0.00890634 0.07124991]
reward =  1.0
totalReward =  6.0
done =  False

7 : -------------------------
action =  1
observation =  [ 0.02219425  0.22603951  0.01033134 -0.21860977]
reward =  1.0
totalReward =  7.0
done =  False

8 : -------------------------
action =  0
observation =  [0.02671504 0.03077141 0.00595914 0.07731411]
reward =  1.0
totalReward =  8.0
done =  False

9 : -------------------------
```

```
cartPole.replayVideo()
```

0:00 / 0:00

```
# Replay the best solution - VIDEO version
#env.close()
show_video()
```

0:00 / 0:00

```
# find average score of 100 episodes using the best solution found:
print("Running 100 episodes using the best solution...")
scores = []
for test in range(100):
  scores.append(cartPole.getScore(best))
  print("scores = ", scores)
  print("Avg. score = ", sum(scores) / len(scores))
```

Part 4.What about the solution dependence on GA conditions?

... with various RANDOM_SEED ...

Results for various RANDOM_SEEDs

RANDOM_SEED = 42

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
```

```
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                          toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                          stats=stats,
                                          halloffame=hof,
                                          verbose=True)
end = time.time()
time_42 = end - start
print("Time Elapsed = ", time_42)
```

```
***************************
gen	nevals	max	avg
0	100	500	18.35
1	94	500	20.6
2	93	500	23.76
3	94	500	38.88
4	93	500	48.58
5	93	500	50.68
6	92	500	62.93
7	93	500	47.7
8	92	500	59.98
9	94	500	75.68
10	95	500	55.81
11	85	500	76.89
12	93	500	79.83
13	92	500	73.48
14	85	500	66.83
15	96	500	69.02
16	94	500	98.28
17	95	500	91.89
18	93	500	92.16
19	93	500	100.52
20	90	500	112.75
21	89	500	112.59
22	92	500	149.5
23	95	500	177.98
24	96	500	228.95
25	89	500	239.97
26	91	500	298.21
```

```
27      91      500     277.71
28      95      500     356.53
29      93      500     426.83
30      94      500     426.92
31      93      500     371.19
32      92      500     415.99
33      91      500     447.65
34      90      500     431.27
35      94      500     445.97
36      91      500     448.41
37      91      500     449.5
38      88      500     470.01
39      93      500     473.17
40      90      500     460.75
Time Elapsed =  100.9089777469635
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMax = %1.5f" % best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_42, meanFitnessValues_GA_42 = logbook.select("max", "avg")
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_42)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_42)
```

```
Best solution:  [0.9039409992284728, 0.06655498740733878, -0.669913677453291
Best FitnessMax = 500.00000
History of maxFitnessValues_GA = [500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 5
History of meanFitnessValues_GA = [18.35, 20.6, 23.76, 38.88, 48.58, 50.68, 6
```

RANDOM_SEED = 666

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 666
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

```
print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_666 = end - start
print("Time Elapsed = ", time_666)
```

```
***************************
gen     nevals  max     avg
0       100     107     13.16
1       94      107     15.3
2       90      195     21.15
3       92      195     22.12
4       90      309     28.75
5       93      500     40.27
6       93      500     51.37
7       92      500     55.92
8       89      500     60.71
9       91      500     52.74
10      90      500     46.15
11      89      500     48.56
12      94      500     48.64
13      96      500     50.98
14      92      500     72.16
15      91      500     63.82
16      92      500     77.75
17      90      500     82.13
18      96      500     101.53
19      93      500     109.37
20      88      500     127.58
21      93      500     177.51
22      94      500     218.88
23      94      500     238.59
24      95      500     248.42
25      95      500     243.61
26      93      500     254.23
27      90      500     286.46
28      94      500     338.73
29      94      500     376.32
30      94      500     411.44
31      89      500     434.94
32      92      500     437.33
33      91      500     434.47
34      92      500     440.55
35      92      500     461.34
36      88      500     460.73
37      91      500     445.49
38      87      500     451.93
39      95      500     433.93
```

```
40      93      500     446.73
Time Elapsed    100.75126065483093
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMax = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_666, meanFitnessValues_GA_666 = logbook.select("max", "avg")
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_666)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_666)
```

```
Best solution:  [-0.2582487265171046, 0.24179682019576307, -0.15088607952951
Best FitnessMax = 500.00000
History of minFitnessValues_GA = [107.0, 107.0, 195.0, 195.0, 309.0, 500.0, 5
History of meanFitnessValues_GA = [13.16, 15.3, 21.15, 22.12, 28.75, 40.27, 5
```

RANDOM_SEED = 1042

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('****************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
```

```
time_1042 = end - start
print("Time Elapsed = ", time_1042)
```

```
***************************
gen     nevals  max     avg
0       100     98      14.05
1       93      369     20.27
2       90      369     24.69
3       95      500     37.39
4       90      500     34
5       94      500     36.67
6       90      500     47.13
7       91      500     58.5
8       95      500     85.11
9       92      500     81.75
10      94      500     83.59
11      92      500     98.02
12      94      500     120.35
13      89      500     136.98
14      87      500     136.72
15      91      500     134.35
16      93      500     128.28
17      95      500     140.19
18      93      500     149.64
19      90      500     161.43
20      95      500     204.21
21      93      500     208.19
22      91      500     234.49
23      90      500     283.42
24      93      500     336.07
25      92      500     339.78
26      89      500     353.78
27      85      500     360.43
28      88      500     376.7
29      95      500     374.63
30      88      500     394.9
31      94      500     357.41
32      90      500     370.26
33      92      500     378.91
34      89      500     362.35
35      83      500     402.14
36      89      500     372.56
37      93      500     393.47
38      96      500     431.22
39      93      500     402.09
40      85      500     411.7
Time Elapsed =  108.02887892723083
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_1042, meanFitnessValues_GA_1042 = logbook.select("max", "avg")
```

```
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_1042)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042)
```

```
Best solution:  [0.5889028101863627, -0.7057154551790351, 0.42332444618582854
Best FitnessMin = 500.00000
History of maxFitnessValues_GA = [98.0, 369.0, 369.0, 500.0, 500.0, 500.0, 50
History of meanFitnessValues_GA = [14.05, 20.27, 24.69, 37.39, 34.0, 36.67, 4
```

RESUME

For various RANDOM_SEED we can obtain **different**:

- **solutions** :) ... of course,
- **performance** (fitness function value),
- **history**.

The reason is the stochastic manner of parameter change during evolution.

... with various GA parameters ... like Crossover Probability

It takes a small change in *P_CROSSOVER* variable.

P_CROSSOVER = 0.1

```
P_CROSSOVER = 0.1  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
```

```
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                          toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                          stats=stats,
                                          halloffame=hof,
                                          verbose=True)
end = time.time()
time_1042_CR0p1 = end - start
print("Time Elapsed = ", time_1042_CR0p1)
```

```
***************************
gen     nevals  max     avg
0       100     98      14.05
1       93      369     20.27
2       90      369     24.69
3       95      500     37.39
4       90      500     34
5       94      500     36.67
6       90      500     47.13
7       91      500     58.5
8       95      500     85.11
9       92      500     81.75
10      94      500     83.59
11      92      500     98.02
12      94      500     120.35
13      89      500     136.98
14      87      500     136.72
15      91      500     134.35
16      93      500     128.28
17      95      500     140.19
18      93      500     149.64
19      90      500     161.43
20      95      500     204.21
21      93      500     208.19
22      91      500     234.49
23      90      500     283.42
24      93      500     336.07
25      92      500     339.78
26      89      500     353.78
27      85      500     360.43
28      88      500     376.7
29      95      500     374.63
30      88      500     394.9
31      94      500     357.41
32      90      500     370.26
33      92      500     378.91
34      89      500     362.35
35      83      500     402.14
36      89      500     372.56
37      93      500     393.47
38      96      500     431.22
39      93      500     402.09
40      85      500     411.7
Time Elapsed =  106.09052872657776
```

```
# print best solution found:
```

```
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_1042_CR0p1, meanFitnessValues_GA_1042_CR0p1 = logbook.select("
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_1042_CR0p1)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p1)
```

```
Best solution:  [0.5889028101863627, -0.7057154551790351, 0.4233244461858285
Best FitnessMin = 500.00000
History of minFitnessValues_GA = [98.0, 369.0, 369.0, 500.0, 500.0, 500.0, 50
History of meanFitnessValues_GA = [14.05, 20.27, 24.69, 37.39, 34.0, 36.67, 4
```

P_CROSSOVER = 0.2

```
P_CROSSOVER = 0.2  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
```

```
                                                              halloffame=hof,
                                                              verbose=True)
end = time.time()
time_1042_CR0p2 = end - start
print("Time Elapsed = ", time_1042_CR0p2)
```

```
***************************
gen     nevals  max     avg
0       100     61      12.76
1       84      139     18.63
2       87      139     20.28
3       86      139     20.38
4       88      262     27.71
5       87      500     36.11
6       89      500     46.12
7       76      500     50.12
8       83      500     69.22
9       83      500     102.08
10      90      500     112.98
11      90      500     135.09
12      92      500     151.42
13      79      500     196.46
14      89      500     206.88
15      85      500     254.88
16      87      500     307.1
17      92      500     343.59
18      87      500     368.36
19      89      500     361.83
20      90      500     386.21
21      84      500     401.56
22      87      500     450.47
23      88      500     428.24
24      89      500     446.1
25      86      500     427.84
26      90      500     465.95
27      87      500     464.36
28      89      500     468.08
29      83      500     461.77
30      88      500     457.34
31      83      500     487.66
32      83      500     485.6
33      83      500     457.34
34      80      500     478.28
35      88      500     470.93
36      87      500     464.49
37      89      500     479.88
38      87      500     462.48
39      90      500     492.46
40      85      500     472.49
Time Elapsed =  137.8567316532135
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
```

```
maxFitnessValues_GA_1042_CR0p2, meanFitnessValues_GA_1042_CR0p2 = logbook.select("
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_1042_CR0p2)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p2)
```

```
Best solution:  [0.8956039608596099, 0.013569898457665541, 0.833010987378023
Best FitnessMin = 500.00000
History of maxFitnessValues_GA = [61.0, 139.0, 139.0, 139.0, 262.0, 500.0, 50
History of meanFitnessValues_GA = [12.76, 18.63, 20.28, 20.38, 27.71, 36.11,
```

P_CROSSOVER = 0.4

```
P_CROSSOVER = 0.4  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042_CR0p4 = end - start
print("Time Elapsed = ", time_1042_CR0p4)
```

```
**************************
gen     nevals  max     avg
0       100     98      14.05
1       57      98      15.38
2       76      413     23.66
3       66      413     32.9
4       65      413     40.32
5       71      413     54.39
6       65      413     76.21
7       57      500     101.52
8       65      500     127.46
9       62      500     165.06
10      74      500     164.39
11      66      500     185.03
12      74      500     168.55
13      59      500     244.16
14      63      500     310.33
15      70      500     359.5
16      61      500     422.41
17      73      500     444.25
18      63      500     481.18
19      66      500     467.12
20      60      500     470.3
21      74      500     484.17
22      69      500     481.37
23      64      500     479.4
24      71      500     484.64
25      65      500     470.95
26      64      500     468.52
27      74      500     472.32
28      67      500     476.02
29      66      500     477.52
30      61      500     492.61
31      72      500     469.78
32      59      500     469.91
33      68      500     464.12
34      68      500     478.64
35      64      500     464.44
36      72      500     481.83
37      65      500     474.61
38      67      500     470.35
39      69      500     453.34
40      64      500     455.02
Time Elapsed =  116.45055270195007
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_1042_CR0p4, meanFitnessValues_GA_1042_CR0p4 = logbook.select("
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_1042_CR0p4)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p4)
```

```
Best solution:  [0.6272811775924462, 0.4841432610995863, 0.6791695047075679,
Best FitnessMin = 500.00000
```

```
History of maxFitnessValues_GA = [98.0, 98.0, 413.0, 413.0, 413.0, 413.0, 413
History of meanFitnessValues_GA = [14.05, 15.38, 23.66, 32.9, 40.32, 54.39, 7
```

P_CROSSOVER = 0.8

```
P_CROSSOVER = 0.8  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042_CR0p8 = end - start
print("Time Elapsed = ", time_1042_CR0p8)
```

```
***************************
gen     nevals  max     avg
0       100     98      14.05
1       91      228     17.72
2       86      228     24.53
3       85      228     31.98
4       84      228     30.83
```

```
5       90      310     34.15
6       86      500     43.3
7       87      500     45.03
8       82      500     44.2
9       88      500     50.58
10      90      500     72.82
11      84      500     69.19
12      86      500     84.47
13      88      500     103.33
14      90      500     119.05
15      91      500     156.08
16      86      500     210.4
17      90      500     244.17
18      85      500     282.28
19      84      500     347.16
20      90      500     369.82
21      86      500     421.07
22      86      500     390.78
23      91      500     405.6
24      86      500     412.67
25      81      500     429.95
26      89      500     431.46
27      84      500     423.54
28      93      500     437.7
29      88      500     448.48
30      84      500     452.05
31      84      500     470.85
32      86      500     456.84
33      84      500     475.83
34      92      500     457.2
35      89      500     461.42
36      81      500     459.49
37      92      500     470.96
38      91      500     460.97
39      88      500     457.45
40      88      500     464.49
Time Elapsed =  126.49325156211853
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_1042_CR0p8, meanFitnessValues_GA_1042_CR0p8 = logbook.select("
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_1042_CR0p8)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p8)
```

```
Best solution:  [-0.3228837696778911, -0.053312503649015824, 0.2949724212959
Best FitnessMin = 500.00000
History of maxFitnessValues_GA = [98.0, 228.0, 228.0, 228.0, 228.0, 310.0, 50
History of meanFitnessValues_GA = [14.05, 17.72, 24.53, 31.98, 30.83, 34.15,
```

P_CROSSOVER = 0.9

```
P_CROSSOVER = 0.9  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCartPole class:
#car = MountainCar(RANDOM_SEED)
cartPole = CartPole(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042_CR0p9 = end - start
print("Time Elapsed = ", time_1042_CR0p9)
```

```
***************************
gen     nevals  max     avg
0       100     98      14.05
1       93      369     20.27
2       90      369     24.69
3       95      500     37.39
4       90      500     34
5       94      500     36.67
6       90      500     47.13
7       91      500     58.5
8       95      500     85.11
9       92      500     81.75
10      94      500     83.59
11      92      500     98.02
12      94      500     120.35
13      89      500     136.98
14      87      500     136.72
```

```
15      91      500     134.35
16      93      500     128.28
17      95      500     140.19
18      93      500     149.64
19      90      500     161.43
20      95      500     204.21
21      93      500     208.19
22      91      500     234.49
23      90      500     283.42
24      93      500     336.07
25      92      500     339.78
26      89      500     353.78
27      85      500     360.43
28      88      500     376.7
29      95      500     374.63
30      88      500     394.9
31      94      500     357.41
32      90      500     370.26
33      92      500     378.91
34      89      500     362.35
35      83      500     402.14
36      89      500     372.56
37      93      500     393.47
38      96      500     431.22
39      93      500     402.09
40      85      500     411.7
Time Elapsed =  107.74227976799011
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
maxFitnessValues_GA_1042_CR0p9, meanFitnessValues_GA_1042_CR0p9 = logbook.select("
print('History of maxFitnessValues_GA =',maxFitnessValues_GA_1042_CR0p9)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p9)
```

```
Best solution:  [0.5889028101863627, -0.7057154551790351, 0.42332444618582854
Best FitnessMin = 500.00000
History of maxFitnessValues_GA = [98.0, 369.0, 369.0, 500.0, 500.0, 500.0, 50
History of meanFitnessValues_GA = [14.05, 20.27, 24.69, 37.39, 34.0, 36.67, 4
```

RESUME

Again ... for various P_CROSSOVER we can obtain **different**:

- **solutions** :) ... of course,
- **performance** (fitness function value),
- **history**.

The reasons are

- the stochastic manner of parameter change during evolution,

- BUT ... more important ... different levels of gene exchange.

... with various GA parameters ... like Mutation Probability

(let's try it as a self-guided learning!)

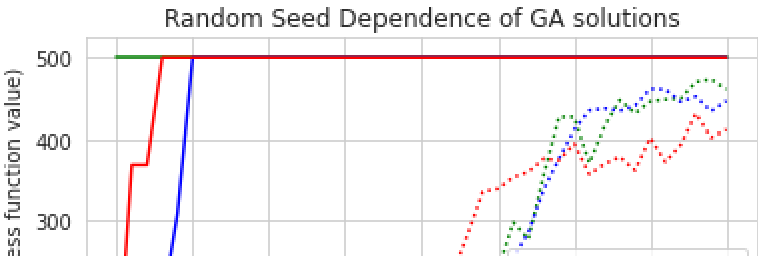
It takes a small change in *P_MUTATION* variable.

Comparison Plots

Random Seed Dependence

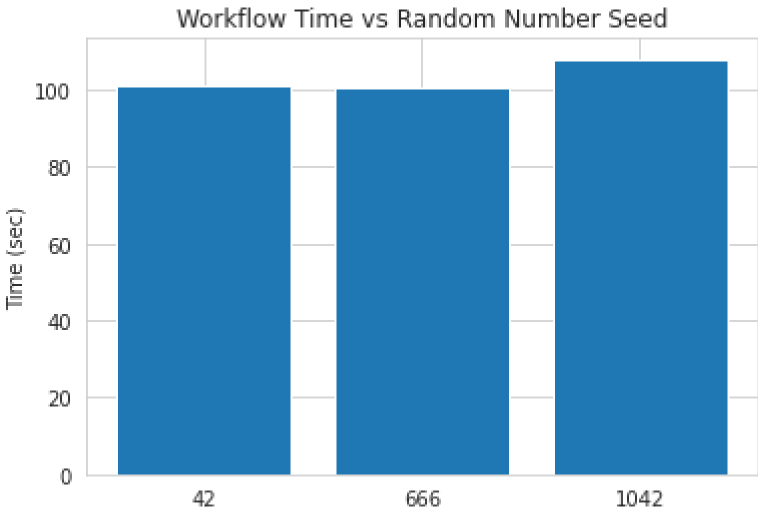
Fitness Function

```
sns.set_style("whitegrid")

# Classic grid search solution
#plt.hlines(accuracy_classic_solution, 0, 5, linestyle = 'solid', label='Classic g

# NN architecture
plt.plot(maxFitnessValues_GA_42, color='green', label='42 (max)')
plt.plot(meanFitnessValues_GA_42, color='green', linestyle = 'dotted', label='42 (

# NN hyperparameter
plt.plot(maxFitnessValues_GA_666, color='blue', label='666 (max)')
plt.plot(meanFitnessValues_GA_666, color='blue', linestyle = 'dotted', label='666

# NN architecture + hyperparameter
plt.plot(maxFitnessValues_GA_1042, color='red', label='1042 (max)')
plt.plot(meanFitnessValues_GA_1042, color='red', linestyle = 'dotted', label='1042

plt.xlabel('Generation')
plt.ylabel('Max / Average (fitness function value)')
plt.title('Random Seed Dependence of GA solutions')
plt.legend(title='Random Number Seed')
plt.show()
```



Time

```
import matplotlib.pyplot as plt

x = ['42','666','1042']
y = [time_42,time_666,time_1042]
plt.bar(x,y)
plt.ylabel('Time (sec)')
plt.title('Workflow Time vs Random Number Seed')
plt.show()
```

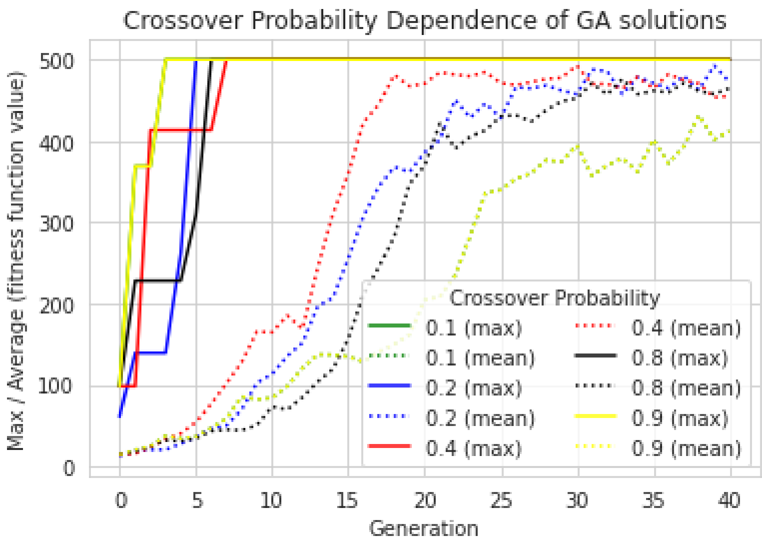


Crossover Probability Dependence

Fitness Function

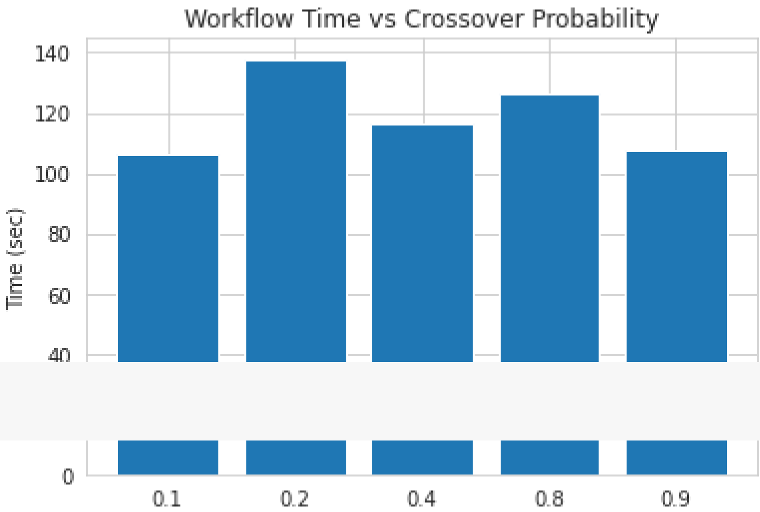
```
sns.set_style("whitegrid")

# Classic grid search solution
#plt.hlines(accuracy_classic_solution, 0, 5, linestyle = 'solid', label='Classic g

# NN architecture
plt.plot(maxFitnessValues_GA_1042_CR0p1, color='green', label='0.1 (max)')
plt.plot(meanFitnessValues_GA_1042_CR0p1, color='green', linestyle = 'dotted', lab
```

```
# NN hyperparameter
plt.plot(maxFitnessValues_GA_1042_CR0p2, color='blue', label='0.2 (max)')
plt.plot(meanFitnessValues_GA_1042_CR0p2, color='blue', linestyle = 'dotted', labe

# NN architecture + hyperparameter
plt.plot(maxFitnessValues_GA_1042_CR0p4, color='red', label='0.4 (max)')
plt.plot(meanFitnessValues_GA_1042_CR0p4, color='red', linestyle = 'dotted', label

# NN architecture + hyperparameter
plt.plot(maxFitnessValues_GA_1042_CR0p8, color='black', label='0.8 (max)')
plt.plot(meanFitnessValues_GA_1042_CR0p8, color='black', linestyle = 'dotted', lab

# NN architecture + hyperparameter
plt.plot(maxFitnessValues_GA_1042_CR0p9, color='yellow', label='0.9 (max)')
plt.plot(meanFitnessValues_GA_1042_CR0p9, color='yellow', linestyle = 'dotted', la

plt.xlabel('Generation')
plt.ylabel('Max / Average (fitness function value)')
plt.title('Crossover Probability Dependence of GA solutions')
plt.legend(title='Crossover Probability', ncol=2)
plt.show()
```



Time

```
import matplotlib.pyplot as plt

x = ['0.1','0.2','0.4','0.8','0.9']
y = [time_1042_CR0p1, time_1042_CR0p2, time_1042_CR0p4, time_1042_CR0p8, time_1042
plt.bar(x,y)
plt.ylabel('Time (sec)')
plt.title('Workflow Time vs Crossover Probability')
plt.show()
```


Workflow Time vs Crossover Probability
Time (sec)
140
120
100
80
60
40
0
0.1
0.2
0.4
0.8
0.9

Lecture 08 - Neuroevolution - EvoJAX

based on (C) Google Brain, Yujin Tang, Yingtao Tian, David Ha works

Brief Content:

- EvoJAX installation (**every time after start of Colab VM!**),
- components needed for the EA workflow,
- *Reinforcement Learning (RL) problem:*
    - CartPole-v1
- policy gradients with parameter-based exploration,
- and others.

By the end of this lecture you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the *Reinforcement Learning* problem using a EA-based solutions for search of solutions,
- how to use policy gradients with parameter-based exploration,
- how to experiment with various settings of the GA and interpret the differences in the results.

Pre-requisite

Before we start, we need to install EvoJAX from [EvoJAX-github] and import some libraries.
Note In our [paper], we ran the experiments on NVIDIA V100 GPU(s). Your results can be different from ours.

```
from IPython.display import clear_output, Image

!pip install evojax

clear_output()
```

```
import os
import numpy as np
import jax
import jax.numpy as jnp

from evojax.task.cartpole import CartPoleSwingUp
```

```
PPolicy
from evojax.algo import PGPE
from evojax import Trainer
from evojax.util import create_logger
```

```
# Let's create a directory to save logs and models.
log_dir = './log'
logger = create_logger(name='EvoJAX', log_dir=log_dir)
logger.info('Welcome to the tutorial on Neuroevolution algorithm creation!')

logger.info('Jax backend: {}'.format(jax.local_devices()))
!nvidia-smi --query-gpu=name --format=csv,noheader
```

```
EvoJAX: 2022-02-11 02:06:40,128 [INFO] Welcome to the tutorial on Neuroevolut
absl: 2022-02-11 02:06:40,137 [INFO] Unable to initialize backend 'tpu_driver
absl: 2022-02-11 02:06:40,322 [INFO] Unable to initialize backend 'tpu': INVA
EvoJAX: 2022-02-11 02:06:40,324 [INFO] Jax backend: [GpuDevice(id=0, process_
Tesla K80
```

Introduction

EvoJAX has three major components: the *task*, the *policy network* and the *neuroevolution algorithm*. Once these components are implemented and instantiated, we can use a trainer to start the training process. The following code snippet provides an example of how we use EvoJAX.

```
seed = 42  # Wish me luck!

# We use the classic cart-pole swing up as our tasks, see
# https://github.com/google/evojax/tree/main/evojax/task for more example tasks.
# The test flag provides the opportunity for a user to
# 1. Return different signals as rewards. For example, in our MNIST example,
#    we use negative cross-entropy loss as the reward in training tasks, and the
#    classification accuracy as the reward in test tasks.
# 2. Perform reward shaping. It is common for RL practitioners to modify the
#    rewards during training so that the agent learns more efficiently. But this
#    modification should not be allowed in tests for fair evaluations.
hard = False
train_task = CartPoleSwingUp(harder=hard, test=False)
test_task = CartPoleSwingUp(harder=hard, test=True)

# We use a feedforward network as our policy.
# By default, MLPPolicy uses "tanh" as its activation function for the output.
policy = MLPPolicy(
    input_dim=train_task.obs_shape[0],
    hidden_dims=[64, 64],
    output_dim=train_task.act_shape[0],
    logger=logger,
)
```

Saved successfully!

```
lgorithm.
# If you want to know more about the algorithm, please take a look at the paper:
# https://people.idsia.ch/~juergen/nn2010.pdf
solver = PGPE(
    pop_size=64,
    param_size=policy.num_params,
    optimizer='adam',
    center_learning_rate=0.05,
    seed=seed,
)

# Now that we have all the three components instantiated, we can create a
# trainer and start the training process.
trainer = Trainer(
    policy=policy,
    solver=solver,
    train_task=train_task,
    test_task=test_task,
    max_iter=600,
    log_interval=100,
    test_interval=200,
    n_repeats=5,
    n_evaluations=128,
    seed=seed,
    log_dir=log_dir,
    logger=logger,
)
_ = trainer.run()
```

```
EvoJAX: 2022-02-11 02:06:43,518 [INFO] MLPPolicy.num_params = 4609
EvoJAX: 2022-02-11 02:06:43,687 [INFO] Start to train for 600 iterations.
EvoJAX: 2022-02-11 02:07:10,038 [INFO] Iter=100, size=64, max=712.8441, avg=6
EvoJAX: 2022-02-11 02:07:29,392 [INFO] Iter=200, size=64, max=782.5107, avg=7
EvoJAX: 2022-02-11 02:07:31,972 [INFO] [TEST] Iter=200, #tests=128, max=816.3
EvoJAX: 2022-02-11 02:07:51,419 [INFO] Iter=300, size=64, max=920.6417, avg=8
EvoJAX: 2022-02-11 02:08:10,756 [INFO] Iter=400, size=64, max=921.8397, avg=8
EvoJAX: 2022-02-11 02:08:10,907 [INFO] [TEST] Iter=400, #tests=128, max=934.5
EvoJAX: 2022-02-11 02:08:30,258 [INFO] Iter=500, size=64, max=932.7117, avg=8
EvoJAX: 2022-02-11 02:08:49,644 [INFO] [TEST] Iter=600, #tests=128, max=955.2
EvoJAX: 2022-02-11 02:08:49,652 [INFO] Training done, best_score=935.1467
```

```
# Let's visualize the learned policy.

def render(task, algo, policy):
    """Render the learned policy."""

    task_reset_fn = jax.jit(test_task.reset)
    policy_reset_fn = jax.jit(policy.reset)
    step_fn = jax.jit(test_task.step)
    act_fn = jax.jit(policy.get_actions)

    params = algo.best_params[None, :]
    task_s = task_reset_fn(jax.random.PRNGKey(seed=seed)[None, :])
    policy_s = policy_reset_fn(task_s)
```

```
    images = [CartPoleSwingUp.render(task_s, 0)]
    done = False
    step = 0
    reward = 0
    while not done:
        act, policy_s = act_fn(task_s, params, policy_s)
        task_s, r, d = step_fn(task_s, act)
        step += 1
        reward = reward + r
        done = bool(d[0])
        if step % 3 == 0:
            images.append(CartPoleSwingUp.render(task_s, 0))
    print('reward={}'.format(reward))
    return images


imgs = render(test_task, solver, policy)
gif_file = os.path.join(log_dir, 'cartpole.gif')
imgs[0].save(
    gif_file, save_all=True, append_images=imgs[1:], duration=40, loop=0)
Image(open(os.path.join(log_dir, 'cartpole.gif'),'rb').read())
```

This tutorial walks you through the process of creating a new neuroevolution algoritm.

To contribute an algorithm implementation to EvoJAX, all you need to do is to implement the `NEAlgorithm` interface.
The interface is defined as the following and you can see the related Python file here:

```
class NEAlgorithm(ABC):
    """Interface of all Neuro-evolution algorithms in EvoJAX."""

    pop_size: int

    @abstractmethod
    def ask(self) -> jnp.ndarray:
        """Ask the algorithm for a population of parameters.
        Returns
            A Jax array of shape (population_size, param_size).
        """
        raise NotImplementedError()

    @abstractmethod
    def tell(self, fitness: Union[np.ndarray, jnp.ndarray]) -> None:
        """Report the fitness of the population to the algorithm.
        Args:
            fitness - The fitness scores array.
        """
        raise NotImplementedError()

    @property
    def best_params(self) -> jnp.ndarray:
        raise NotImplementedError()

    @best_params.setter
    def best_params(self, params: Union[np.ndarray, jnp.ndarray]) -> None:
        raise NotImplementedError()
```

Wrap an existing implementation

`NEAlgorithm` adopts the well-known “ask” and “tell” interfaces, where the former requests the algorithm to generate a population of parameters and the latter reports the parameters' fitness scores so that the algorithm can update its internal states. We think the conventional interface for the neuroevolution algorithms brings familiarity to the developers and thus reduces the interface is also used by many existing algorithms, it is therefore possible for the practitioners to quickly plug in existing algorithms for sanity checks. In the first part of this tutorial, we will create an implementation that wraps CMA-ES. Please take a look at this wonderful tutorial for more information about CMA-ES.

```
import cma
from evojax.algo.base import NEAlgorithm


class CMAWrapper(NEAlgorithm):
    """This is a wrapper of CMA-ES."""

    def __init__(self, param_size, pop_size, init_stdev=0.1, seed=0):

        self.pop_size = pop_size
        self.params = None
        self._best_params = None

        # We create CMA-ES in a simplest form.
        self.cma = cma.CMAEvolutionStrategy(
            x0=np.zeros(param_size),
            sigma0=init_stdev,
            inopts={
                'popsize': pop_size,
                'seed': seed if seed > 0 else 42,
                'randn': np.random.randn,
            },
        )

        # We jit-compile some utility functions.
        self.jnp_array = jax.jit(jnp.array)
        self.jnp_stack = jax.jit(jnp.stack)

    def ask(self):
        self.params = self.cma.ask()
        return self.jnp_stack(self.params)

    def tell(self, fitness):
        # CMA-ES minimizes, so we negate the fitness.
        self.cma.tell(self.params, -np.array(fitness))
        self._best_params = np.array(self.cma.result.xfavorite)

    @property
    def best_params(self):
        return self.jnp_array(self._best_params)

    @best_params.setter
    def best_params(self, params):
        self._best_params = np.array(params)
```

 is extremely simple, we haven't used many options or functions provided by CMA-ES.
But let's plug in this implementation to our cart-pole earlier example and see how it works.
Alert Depending on your CPUs, running the following cell may take some time.

```
# Instead of PGPE, we use our CMAWrapper now.
solver = CMAWrapper(
    pop_size=64,
    param_size=policy.num_params,
    seed=seed,
)
trainer = Trainer(
    policy=policy,
    solver=solver,
    train_task=train_task,
    test_task=test_task,
    max_iter=600,
    log_interval=100,
    test_interval=200,
    n_repeats=5,
    n_evaluations=128,
    seed=seed,
    log_dir=log_dir,
    logger=logger,
)
_ = trainer.run()
```

```
EvoJAX: 2022-02-11 02:08:59,845 [INFO] Start to train for 600 iterations.
(32_w,64)-aCMA-ES (mu_w=17.6,w_1=11%) in dimension 4609 (seed=42, Fri Feb 11
EvoJAX: 2022-02-11 02:14:31,792 [INFO] Iter=100, size=64, max=643.9105, avg=4
EvoJAX: 2022-02-11 02:20:00,840 [INFO] Iter=200, size=64, max=692.3575, avg=5
EvoJAX: 2022-02-11 02:20:01,894 [INFO] [TEST] Iter=200, #tests=128, max=751.8
EvoJAX: 2022-02-11 02:25:27,575 [INFO] Iter=300, size=64, max=718.8022, avg=5
EvoJAX: 2022-02-11 02:31:06,983 [INFO] Iter=400, size=64, max=747.0325, avg=5
EvoJAX: 2022-02-11 02:31:07,139 [INFO] [TEST] Iter=400, #tests=128, max=706.6
EvoJAX: 2022-02-11 02:36:32,660 [INFO] Iter=500, size=64, max=725.8452, avg=6
EvoJAX: 2022-02-11 02:41:55,297 [INFO] [TEST] Iter=600, #tests=128, max=764.3
EvoJAX: 2022-02-11 02:41:55,305 [INFO] Training done, best_score=743.6188
```

The simple CMA-ES wrapper worked! However, we also notice that the training time increased significantly.
Although the task and the policy networks are accelerated by GPUs, the `cma.CMAEvolutionStrategy` implementation we used in the code above relies on CPUs, and that is why we see the drop in training speed.

Nevertheless, being able to wrapper an existing algorithm and plug that in EvoJAX's training pipeline serves as sanity checks and helps debugging when you migrate algorithms to EvoJAX. Next, we will show you how to implement an algorithm in JAX from scratch.

We are going to implement a very simple version of PGPE, users interested in the algorithm can take a look at the paper and also check out some popular implementations (example1, example2).

In a nutshell, PGPE samples the policy network parameters θ from Gaussian distributions. It maintains the means μ and the standard deviations σ of the Gaussian distributions, and then estimates the gradients of these parameters using the following formulae:

$$\Delta\mu_i = \alpha(r-b)(\theta_i-\mu_i)\,, \Delta\sigma_i = \alpha(r-b)\frac{(\theta_i-\mu_i)^2-{\sigma_i}^2}{\sigma_i}$$

where α is the learning rate and b is a baseline from the reward r.

The following code snippet provides a sample implementation of PGPE.
Note This simplified version ignores popular tricks such as converting the rewards to ranks, using modern optimizers for parameter update, etc.

```
from evojax.algo.base import NEAlgorithm


class SimplePGPE(NEAlgorithm):
    """A simplified version of PGPE."""

    def __init__(self, param_size, pop_size,
                 lr_mu=0.05, lr_sigma=0.1, init_stdev=0.1, seed=0):

        self.pop_size = pop_size
        assert pop_size % 2 == 0, "pop_size must be a multpile of 2."
        n_directs = pop_size // 2
        self.noises = jnp.zeros(param_size)
        self.params = jnp.zeros(param_size)
        self.mu = jnp.zeros(param_size)
        self.sigma = jnp.ones(param_size) * init_stdev
        self.rand_key = jax.random.PRNGKey(seed=seed)

        def ask_fn(key, mu, sigma):
            next_key, sample_key = jax.random.split(key=key, num=2)
            perturbations = jax.random.normal(
                key=sample_key, shape=(n_directs, param_size)) * sigma[None, :]
            params = jnp.vstack([perturbations, -perturbations]) + mu[None, :]
            return params, perturbations, next_key

        self.ask_fn = jax.jit(ask_fn)

        def tell_fn(rewards, mu, sigma, perturbations):
            fitness = jnp.array(rewards).reshape([2, n_directs])

            # To map to the formulae above:
                                        ness - b) and (theta - mu) = perturbations
```

```
s.mean(axis=0)
        b = jnp.mean(fitness)

        # Update the means.
        grad_mu = (
            (avg_fitness - b)[:, None] * perturbations
        ).mean(axis=0)
        new_mu = mu + lr_mu * grad_mu

        # Update the sigmas.
        # We constrain the change of sigma to prevent numerical errors.
        grad_sigma = (
            (avg_fitness - b)[:, None] *
            (perturbations ** 2 - (sigma ** 2)[None, :]) / sigma[None, :]
        ).mean(axis=0)
        new_sigma = jnp.clip(
            sigma + lr_sigma * grad_sigma, 0.8 * sigma, 1.2 * sigma)

        return new_mu, new_sigma

    self.tell_fn = jax.jit(tell_fn)

def ask(self):
    self.params, self.noises, self.rand_key = self.ask_fn(
        self.rand_key, self.mu, self.sigma)
    return self.params

def tell(self, fitness):
    self.mu, self.sigma = self.tell_fn(
        fitness, self.mu, self.sigma, self.noises)

@property
def best_params(self):
    return self.mu

@best_params.setter
def best_params(self, params):
    self.mu = jnp.array(params)
```

```
# Let's test our simple PGPE.
solver = SimplePGPE(
    pop_size=64,
    param_size=policy.num_params,
    seed=seed,
)
trainer = Trainer(
    policy=policy,
    solver=solver,
    train_task=train_task,
    test_task=test_task,
    max_iter=1000,
    log_interval=100,
    test_interval=200,
    n_repeats=5,
```

Saved successfully!

```
    log_dir=log_dir,
    logger=logger,
)
_ = trainer.run()
```

```
EvoJAX: 2022-02-11 02:41:55,585 [INFO] Start to train for 1000 iterations.
EvoJAX: 2022-02-11 02:42:16,350 [INFO] Iter=100, size=64, max=413.9725, avg=1
EvoJAX: 2022-02-11 02:42:35,561 [INFO] Iter=200, size=64, max=512.9973, avg=3
EvoJAX: 2022-02-11 02:42:36,577 [INFO] [TEST] Iter=200, #tests=128, max=556.6
EvoJAX: 2022-02-11 02:42:55,783 [INFO] Iter=300, size=64, max=523.7679, avg=4
EvoJAX: 2022-02-11 02:43:14,984 [INFO] Iter=400, size=64, max=567.3138, avg=5
EvoJAX: 2022-02-11 02:43:15,137 [INFO] [TEST] Iter=400, #tests=128, max=585.5
EvoJAX: 2022-02-11 02:43:34,341 [INFO] Iter=500, size=64, max=586.1516, avg=5
EvoJAX: 2022-02-11 02:43:53,552 [INFO] Iter=600, size=64, max=567.7144, avg=5
EvoJAX: 2022-02-11 02:43:53,705 [INFO] [TEST] Iter=600, #tests=128, max=636.5
EvoJAX: 2022-02-11 02:44:13,604 [INFO] Iter=700, size=64, max=592.1466, avg=4
EvoJAX: 2022-02-11 02:44:32,814 [INFO] Iter=800, size=64, max=603.3476, avg=5
EvoJAX: 2022-02-11 02:44:32,966 [INFO] [TEST] Iter=800, #tests=128, max=665.1
EvoJAX: 2022-02-11 02:44:52,172 [INFO] Iter=900, size=64, max=632.6639, avg=5
EvoJAX: 2022-02-11 02:45:11,339 [INFO] [TEST] Iter=1000, #tests=128, max=643.
EvoJAX: 2022-02-11 02:45:11,346 [INFO] Training done, best_score=592.5771
```

Despite its simplicity, the training and test scores rise steadily. You can see our complete implementation of PGPE here.

We hope this tutorial helps. Please let us (evojax-dev@google.com) know if you have any problems or suggestions, thanks!

Saved successfully!

roducts - Cancel contracts here

Lecture 08 - Neuroevolution – EvoJAX – Additional Materials

****
Neuroevolution
https://en.wikipedia.org/wiki/Neuroevolution
****

EvoJAX

****
Paper:
https://arxiv.org/abs/2202.05008
https://arxiv.org/pdf/2202.05008.pdf

****
Codes + notebooks:
EvoJAX: Hardware-Accelerated Neuroevolution
https://github.com/google/evojax

****
Blogs:
EvoJAX: A Great Framework For Most Deep Tasks
https://rezayazdanfar.medium.com/evojax-a-great-framework-for-most-deep-tasks-10adf685c152
6 min read

Google Brain's EvoJAX Hardware-Accelerated Toolkit Significantly Improves Neuroevolutionary Computation
https://medium.com/syncedreview/google-brains-evojax-hardware-accelerated-toolkit-significantly-improves-neuroevolutionary-7943f92adb
4 min read

****
Presentation from co-author with reviewers:
EvoJAX: Hardware-Accelerated Neuroevolution
https://www.youtube.com/watch?v=vfz0XfZ_AbM
1:22:08

EvoJAX: Hardware-Accelerated Neuroevolution

Yujin Tang
yujintang@google.com
Google Brain

Yingtao Tian
alantian@google.com
Google Brain

David Ha
hadavid@google.com
Google Brain

ABSTRACT

Evolutionary computation has been shown to be a highly effective method for training neural networks, particularly when employed at scale on CPU clusters. Recent work have also showcased their effectiveness on hardware accelerators, such as GPUs, but so far such demonstrations are tailored for very specific tasks, limiting applicability to other domains. We present EvoJAX, a scalable, general purpose, hardware-accelerated neuroevolution toolkit. Building on top of the JAX library, our toolkit enables neuroevolution algorithms to work with neural networks running in parallel across multiple TPU/GPUs. EvoJAX achieves very high performance by implementing the evolution algorithm, neural network and task all in NumPy, which is compiled just-in-time to run on accelerators. We provide extensible examples of EvoJAX for a wide range of tasks, including supervised learning, reinforcement learning and generative art. Since EvoJAX can find solutions to most of these tasks within minutes on a single accelerator, compared to hours or days when using CPUs, our toolkit can significantly shorten the iteration cycle of evolutionary computation experiments.
EvoJAX is available at **https://github.com/google/evojax**

ACM Reference Format:
Yujin Tang, Yingtao Tian, and David Ha. 2022. EvoJAX: Hardware-Accelerated Neuroevolution. In *2022 Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3528770

1 INTRODUCTION

Hardware accelerators have played an important role in advancing the state-of-the-art for deep learning (DL), enabling rapid training of neural networks and shorter research iteration cycles for their development [12]. But much of this progress is restricted to systems that rely on gradient descent, a highly effective optimization method when we provide it with a well-defined objective function. But in areas such as artificial life, complex systems, computational biology, and even classical physics [18], much of the interesting behaviors we observe take place near the chaotic states, where a system is constantly transitioning between order and disorder. It can be argued that intelligent life and even civilization are all complex systems operating at the *edge of chaos* [3, 16]. If we wish to study these systems, we need efficient methods to simulate and find solutions in complex systems.

GECCO '22, July 9–13, 2022, Boston, USA

ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3528770

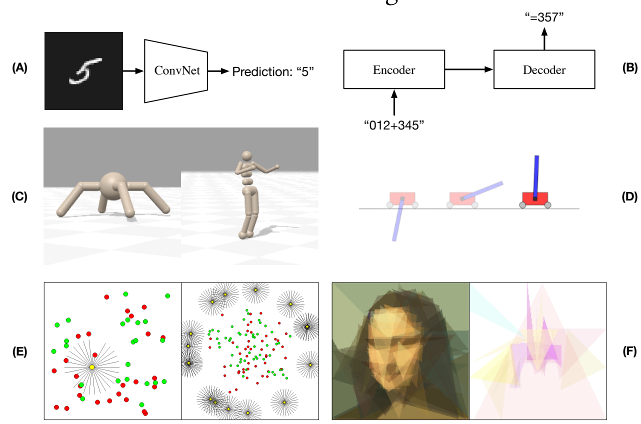


Figure 1: EvoJAX Examples. (A) MNIST classification. (B) Seq2Seq learning. (C) Robotic control. (D) Cart-pole swing up. (E) Left: WaterWorld wherein the agent (yellow) tries to get food (green) while avoiding poison (red). Right: A version of WaterWorld with multiple agents. (F) Abstract painting with only triangles. Left: Painting a concrete image. Right: Painting the concept "Walt Disney World".

Neural networks are a promising approach for modeling complex systems [9, 19], and neuroevolution has made great progress in developing methods for evolving neural networks to solve a wide range of problems. Evolution-based methods have been shown to find state-of-the-art solutions for reinforcement learning (RL) [8, 13, 22, 25, 29]. A policy with non-differentiable operations can solve many more tasks than one that is fully differentiable [20, 27, 28, 33]. More importantly, the removal of the requirement of a differentiable policy also liberates the researchers' mind, enabling higher levels of creativity for looking at problems and directions differently from the mainstream. In a sense, enabling researchers to use neural networks beyond gradient-based methods also enables the broader machine learning (ML) research community to explore in a way that is also less "grad student descent" [7]-based.

However, the progress of hardware-accelerated computational methods for evolution has not kept pace with ML, or even RL. Much of computational evolution is still conducted using CPU clusters, largely ignoring the recent breakthroughs in hardware accelerators such as GPUs/TPUs. Recent work started to demonstrate effectiveness of GPUs for neuroevolution [25], but so far such demonstrations are tailored for specific tasks [24], limiting their applicability to other domains. To enable greater access to hardware accelerators for neuroevolution researchers, we developed EvoJAX, a scalable, general purpose, neuroevolution toolkit. Building on the JAX library [1], our toolkit enables neuroevolution algorithms to work with neural networks running in parallel across multiple TPU/GPUs. EvoJAX achieves very high performance by implementing the evolution algorithm, neural network and task all in NumPy, which is compiled just-in-time to run on accelerators.

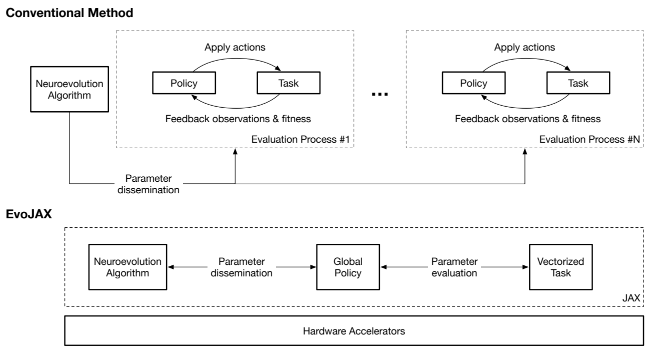


Figure 2: Architectural Overview of EvoJAX.

In this paper, we describe the design of EvoJAX and show how one can use and extend EvoJAX for neuroevolution research. We showcase several extensible examples of EvoJAX for a wide range of tasks, including supervised learning (image classification, seq-to-seq), RL (cart-pole swing-up [6], Brax locomotion [5], multi-agent water world), and generative art (image approximation with shapes, CLIP-guided abstract art [30]). We show that EvoJAX can find solutions to most of these tasks within minutes on GPU/TPUs, compared to hours or days when using CPUs. We believe our toolkit can significantly shorten the experimental iteration cycle for researchers working with evolutionary computation. We have also created several tutorials and notebooks as part of this open-source project to make adapting EvoJAX for novel use cases straightforward.

2 SYSTEM DESIGN

EvoJAX aims to improve the neuroevolution training efficiency by implementing the entire pipeline in modern ML frameworks that support hardware acceleration. We choose JAX[1] in our current implementation due to its wide variety of hardware support and its matured features of auto-vectorization, device-parallelism, just-in-time compilation, etc. As we will see in Section 4, as long as the component interfaces are properly implemented, EvoJAX also allows user extensions with other frameworks.

Figure 2 gives an overview of how EvoJAX works. There are three major components – the neuroevolution algorithm, the policy and the task. Although these components are common in conventional neuroevolution implementations, we highlight the key differences that make EvoJAX much more efficient:

Modern ML Optimizers Researchers and practitioners in the field of DL have been focusing on inventing optimization algorithms [21] and techniques [15, 32, 34] that are both fast and effective. Although these techniques were tailored for gradient-based optimizations, they can be directly applied to gradient estimation-based evolutionary algorithms [17, 23] too. By leveraging JAX-based libraries [1, 10, 11], EvoJAX not only achieves significant speed-up but also provides the users with the tools and the interfaces to develop their own implementations in a mature framework.

Global Policy In conventional neuroevolution implementations, it is a common practice to spawn multiple processes for parameters evaluation. To achieve hardware acceleration, the implementation adopts one of the DL frameworks and then each of the evaluation processes maintains a separate computational graph for the same policy. Unfortunately, most DL frameworks are not designed for multi-process training scenarios and often cause difficulties. Moreover, when these processes are run on the same accelerator, maintaining identical copies of the computational graph is a waste of resource. Conforming to the "Single-Program, Multiple-Data" (SPMD) model [4], EvoJAX solves this by building a global policy and treat both the task observations and the policy parameters as data for the computational graph. This global policy design is easy to implement as it is consistent with DL frameworks, and in the experiments we observe high data-throughput.

Vectorized Tasks Same as the policies, conventional methods also create copies of the tasks in the spawned processes for independent parameters evaluations. To be compliant with EvoJAX's global policy design, we propose to group these tasks in a vectorized form. In terms of implementation, this can be achieved by either creating the task in auto-vectorizaton supported frameworks or by creating a task observations collector on top of all the evaluation processes. EvoJAX adopts the first method.

Device Parallelism Thanks to the device-parallelism support in JAX, EvoJAX is capable of scaling its training procedure almost linearly to the available hardware accelerators. Utilizing EvoJAX's training pipeline, this device parallelism is automatically managed and is transparent to the users. As we will see in Section 3, together with the previously mentioned features, EvoJAX significantly shortens the training time for novel and non-trivial tasks.

EvoJAX defines simple yet functionally complete interfaces for the three components, any implementations that are compliant with the interfaces can be seamlessly integrated (see Section 4).

Finally, in addition to the mentioned major components, EvoJAX also comes with a trainer and a simulation manager that help orchestrate and manage the training process. They contain detailed implementations of task roll-out seeds generation, efficient training loops, time profiling and logistics operations such as logging, testing and periodic model saving. Convenient as they are, we point out that EvoJAX is a flexible toolkit, where it is possible to use any component independently (e.g., using a custom training loop).

3 EVOJAX EXAMPLES

We provide a total of six examples (see Figure 1) to showcase the capacity, efficiency and the usage of EvoJAX online in the format of Python scripts and notebooks. The examples are designed to feature different aspects of EvoJAX and are in three categories: Supervised Learning Tasks, Control Tasks and Novel Tasks. As the experimental setups, "Robotic Control" was trained with TPUs, "Concrete and Abstract Painting" was trained with 8 NVIDIA V100 GPUs, and the rest were trained with 1 NVIDIA V100 GPU.

Supervised Learning Tasks They provide both the data and the ground-truth labels to train the policy. In EvoJAX, supervised learning tasks are modelled as single-step tasks, the examples in this category are thus isolated from other factors to prove the correctness and efficiency of our algorithms' implementation.

- *MNIST Classification*. Here, we train a convolutional neural network (ConvNet) with 10K parameters with EvoJAX. Although MNIST is a solved problem in DL, it is non-trivial for neuroevolution in terms of achieving high test accuracy within a short time (e.g., in minutes). We show that EvoJAX can train the ConvNet to reach $> 98\%$ test accuracy within 5 minutes.

- *Seq2Seq Learning*. It has recently been shown that genetic algorithms (GA) can train large models [20]. Here, we show that EvoJAX is also capable of training a large network with hundreds of thousands of parameters. We adopt a seq-to-seq task where the policy is required to output a sequence after observing a query sequence. Concretely, the query is a sequence that represents the addition of two randomly generated integers (e.g., "012+345=", we pad the numbers with leading 0's so that they have equal lengths) and the result is a sequence representing the answer. Using an LSTM-based seq2seq [26] model, EvoJAX achieves > 99% test accuracy within tens of minutes.

While one would obviously use gradient-descent for such tasks in practice, the point is to show that neuroevolution can also solve them to some degree of accuracy within a short amount of time, which will be useful when these models are adapted within a more complicated task where gradient-based approaches may not work.

Control Tasks The purpose of including control tasks are two-fold: 1) Unlike supervised learning tasks, control tasks in EvoJAX have undetermined number of steps, we thus use these examples to demonstrate the efficiency of our task roll-out loops. 2) We wish to show the speed-up benefit of implementing tasks in JAX and illustrate how to implement one from scratch.

- *Robotic Control*. Brax [5] is a differentiable physics engine implemented in JAX that simulates environments made up of rigid bodies, joints, and actuators. We show that it is easy to wrap Brax tasks in EvoJAX, and it takes EvoJAX tens of minutes to solve a robotic locomotion task on Colab TPUs.
- *Cart-Pole Swing Up*. Through this classic control task, we illustrate how a task is implemented from scratch in JAX and integrated into EvoJAX's training pipeline. In our implementation, a user can command the initial states to be randomly sampled from a narrow (easy version) or a wide (hard version) range of possible settings, with the latter being much harder to solve. EvoJAX solves both versions within minutes.

Novel Tasks In this last category, we go beyond simple illustrations and show examples of novel tasks that are more practical and attractive to researchers in the genetic and evolutionary computation area, with the goal of helping them try out ideas in EvoJAX.

- *WaterWorld*. In this task [14], an agent tries to get as much food as possible while avoiding poisons. EvoJAX is able to train the agent in tens of minutes. Furthermore, we demonstrate that multi-agents training in EvoJAX is possible. Here, we spawn the entire population in the same task roll-out and directly measure each agent's performance in a multi-agent world. This training scheme automatically generates task complexity beyond human design, and is beneficial for learning policies that can deal with interactions between agents and environmental uncertainties.
- *Concrete and Abstract Painting*. We reproduce the results from a computational creativity work [30]. The original work, whose implementation requires multiple CPUs and GPUs, could be accelerated on a single GPU efficiently using EvoJAX, which was not possible before. Moreover, with multiple GPUs/TPUs, EvoJAX can further speed up the mentioned work almost linearly. We also show that the modular design of EvoJAX allows its components be used independently – in this case it is possible to use only the neuroevolution algorithms from EvoJAX while leveraging one's own training loops and environment implantation.

Table 1: Time Comparisons. We report the training time for both methods to achieve widely accepted test scores.

	Baseline	EvoJAX
MNIST	36 min	3 min
Cart-Pole Swing Up (Hard Version)	37 min	2 min
Locomotion (Ant)[1]	201 min	9 min

We summarize EvoJAX's benefit via these examples. First of all, EvoJAX brings significant training speed up. In Table 1 we show the time costs of training some popular tasks with both a conventional setup and EvoJAX.[1] On modest hardware accelerators, EvoJAX trains $10 \sim 20$ times faster which leads to quicker idea iterations. Secondly, the capability of training multi-agents in a complex setting that is beyond human design supplies training environmental richness. And finally, EvoJAX puts the entire pipeline on unified hardware setups and that allows the practitioners to simplify complex hardware arrangements. As an example, for the substantial load of computation in our Abstract Painting example, the baseline needs to use both GPUs and CPUs, while EvoJAX only uses GPUs.

4 EXTENDING EVOJAX

A goal of EvoJAX is to provide researchers with an infrastructure that allows fast idea iterations. With EvoJAX it is possible to devise more effective neuroevolution algorithms, to explore novel policy architectures, and to experiment with new tasks. EvoJAX has carefully defined interfaces, as long as these interfaces are properly implemented, a user extended module can be integrated into the pipeline seamlessly.

```
import jax.numpy as jnp

class TaskState: obs: jnp.ndarray
class PolicyState: keys: jnp.ndarray
class NEAlgorithm:
    def ask(self) -> jnp.ndarray: pass
    def tell(self, fitness: jnp.ndarray) -> None: pass
class PolicyNetwork:
    def reset(self, states: TaskState) -> PolicyState: pass
    def get_actions(self, t_states: TaskState, params: jnp.ndarray,
                    p_states: PolicyState) \
                        -> Tuple[jnp.ndarray, PolicyState]: pass
class VectorizedTask:
      def reset(self, key: jnp.ndarray) -> TaskState: pass
      def step(self, state: TaskState, action: jnp.ndarray) \
                 -> Tuple[TaskState, jnp.ndarray, jnp.ndarray]: pass
```

Figure 3: Major Component Interfaces in EvoJAX.

Devising New Algorithms Users interested in inventing new neuroevolution algorithms should implement *NEAlgorithm* in Figure 3, which serves as the base class for all neuroevolution algorithms in EvoJAX. Being consistent with most conventional implementations, *NEAlgorithm* adopts the "ask" and "tell" interfaces, where the former requests the algorithm to generate a population of parameters and the latter reports the parameters evaluation results back to the algorithm for internal states update. Taking on the conventional interfaces for the neuroevolution algorithms not only brings familiarity to the developers and thus reducing the required

[1]We use the code from [27] as the baseline. For the Locomotion task, we use PyBullet Ant in the baseline and Brax Ant in EvoJAX. The baseline is trained with 96 CPUs.

learning effort, but also allows the practitioners to quickly plug in existing algorithms for sanity checks by writing a simple wrapper.

Exploring Novel Policy Architectures *PolicyNetwork* in Figure 3 defines the policy interface, all policies in EvoJAX implement the *get_actions* method. The method puts no restrictions on what the policy network should be or how it should behave, giving full freedom for neural architecture search (NAS). Because EvoJAX conforms to the SPMD model, *get_actions* accepts three parameters: the vectorized task states, the population parameters and the policy's internal states. At the beginning of a roll-out, each individual in the population sees identical observations, they will then diverge due to the population's different behaviors. Because JAX requires pure functions, the policy's states (e.g., random seeds, LSTM cell states, etc) are passed to *get_actions* via a Flax [10] dataclass *p_states*, which is initialized by *PolicyNetwork.reset*. The method returns the actions and the updated policy states. At runtime, calling *get_actions* is equivalent to passing a batch of data through the model.

Experimenting with More Tasks In Figure 3, *VectorizedTask* forms the base for all EvoJAX tasks. Similar to OpenAI's Gym environments [2], the interface defines the *reset* and the *step* methods. Following the pure-function principle of JAX, one major difference between EvoJAX tasks and Gym environments is that EvoJAX's tasks do not keep internal states. Instead, these states are encapsulated in a *TaskState* instance and carried over the roll-out steps. Similar to *PolicyState*, users can inherit *TaskState* and create one's own task specific state to encapsulate arbitrary information besides the environment observations. In most tasks, the initial states are generated via a procedure of randomness. The *reset* method thus accepts *key*'s that act as seeds for the random process.

5 LIMITATIONS AND FUTURE WORKS

EvoJAX is based on the JAX framework, which is based on the familiar NumPy and is thus friendly to researchers accustomed to such tools. However, practitioners may have to take effort to understand the subtleties of JAX in order to maximize its performance. The time spent on learning the JAX framework may translate to a delayed adoption of EvoJAX, hence much of our focus so far has been on creating examples and tutorials that others can use as templates to build upon. Another limitation of EvoJAX is the compatibility with existing non-parallelizable tasks. Although it is possible to create an observation collector on top of the evaluation processes to mimic the behavior of *VectorizedTask*, the operation involves inter-process communications that becomes a bottleneck, preventing such tasks from the benefit of hardware-acceleration.

In the future, we plan to release more neuroevolution algorithm implementations to EvoJAX in addition to PGPE [23, 31] in the current release. We will add more policies and tasks to both demonstrate a wider variety of examples in order to encourage greater adoption of EvoJAX, with the goal of further enhancing the computation tools available in evolutionary computation research.

REFERENCES

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax
[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. 2016. Openai gym. *arXiv:1606.01540* (2016).
[3] Leon Chua, Valery Sbitnev, and Hyongsuk Kim. 2012. Neurons are poised near the edge of chaos. *International Journal of Bifurcation and Chaos* 22, 04 (2012).
[4] Frederica Darema. 2001. The spmd model: Past, present and future. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer.
[5] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. *Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation*. http://github.com/google/brax
[6] Daniel Freeman, David Ha, and Luke Metz. 2019. Learning to Predict Without Looking Ahead: World Models Without Forward Prediction. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.
[7] Oguzhan Gencoglu, Mark van Gils, Esin Guldogan, Chamin Morikawa, Mehmet Süzen, Mathias Gruber, Jussi Leinonen, and Heikki Huttunen. 2019. HARK Side of Deep Learning–From Grad Student Descent to Automated Machine Learning. *arXiv:1904.07633* (2019).
[8] David Ha. 2020. Slime Volleyball Gym Environment.
[9] David Ha and Yujin Tang. 2021. Collective Intelligence for Deep Learning: A Survey of Recent Developments. *arXiv:2111.14377* (2021).
[10] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. 2020. *Flax: A neural network library and ecosystem for JAX*. http://github.com/google/flax
[11] Matteo Hessel, David Budden, Fabio Viola, Mihaela Rosca, Eren Sezener, and Tom Hennigan. 2020. *Optax: composable gradient transformation and optimisation, in JAX!* http://github.com/deepmind/optax
[12] Sara Hooker. 2021. The hardware lottery. *Commun. ACM* 64, 12 (2021), 58–65.
[13] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv:1711.09846* (2017).
[14] Andrej Karpathy. 2015. *REINFORCEjs*. https://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html
[15] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv:1609.04836* (2016).
[16] Roger Lewin. 1999. *Complexity: Life at the edge of chaos*. University of Chicago.
[17] Horia Mania, Aurelia Guy, and Benjamin Recht. 2018. Simple random search of static linear policies is competitive for reinforcement learning. In *The 32nd Conference on Neural Information Processing Systems*. 1805–1814.
[18] Luke Metz, C Daniel Freeman, Samuel S Schoenholz, and Tal Kachman. 2021. Gradients are Not All You Need. *arXiv:2111.05803* (2021).
[19] Sebastian Risi. 2021. The Future of Artificial Intelligence is Self-Organizing and Self-Assembling. https://sebastianrisi.com/self_assembling_ai.
[20] Sebastian Risi and Kenneth O Stanley. 2019. Deep neuroevolution of recurrent and discrete world models. In *Proceedings of GECCO*. 456–462.
[21] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv:1609.04747* (2016).
[22] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv:1703.03864* (2017).
[23] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. 2010. Parameter-exploring policy gradients. *Neural Networks* 23, 4 (2010), 551–559.
[24] Felipe Such. 2018. *Accelerating Deep Neuroevolution: Train Atari in Hours on a Single Personal Computer*. https://eng.uber.com/accelerated-neuroevolution/
[25] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2017. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv:1712.06567* (2017).
[26] I. Sutskever, O. Vinyals, and Q. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in NIPS*. 3104–3112.
[27] Yujin Tang and David Ha. 2021. The Sensory Neuron as a Transformer: Permutation-Invariant Neural Networks for Reinforcement Learning. In *The 35th Conference on Neural Information Processing Systems*.
[28] Yujin Tang, Duong Nguyen, and David Ha. 2020. Neuroevolution of Self-Interpretable Agents. In *Genetic and Evolutionary Computation Conference*.
[29] Yujin Tang, Jie Tan, and Tatsuya Harada. 2020. Learning agile locomotion via adversarial training. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 6098–6105.
[30] Yingtao Tian and David Ha. 2021. Modern Evolution Strategies for Creativity: Fitting Concrete Images and Abstract Concepts. *arXiv:2109.08857* (2021).
[31] Nihat Engin Toklu, Paweł Liskowski, and Rupesh Kumar Srivastava. 2020. ClipUp: A Simple and Powerful Optimizer for Distribution-Based Policy Evolution. In *International Conference on Parallel Problem Solving from Nature*. 515–527.
[32] Twan Van Laarhoven. 2017. L2 regularization versus batch and weight normalization. *arXiv:1706.05350* (2017).
[33] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. 2019. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv:1901.01753* (2019).
[34] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I Jordan. 2019. How does learning rate decay help modern neural networks? *arXiv:1908.01878* (2019).