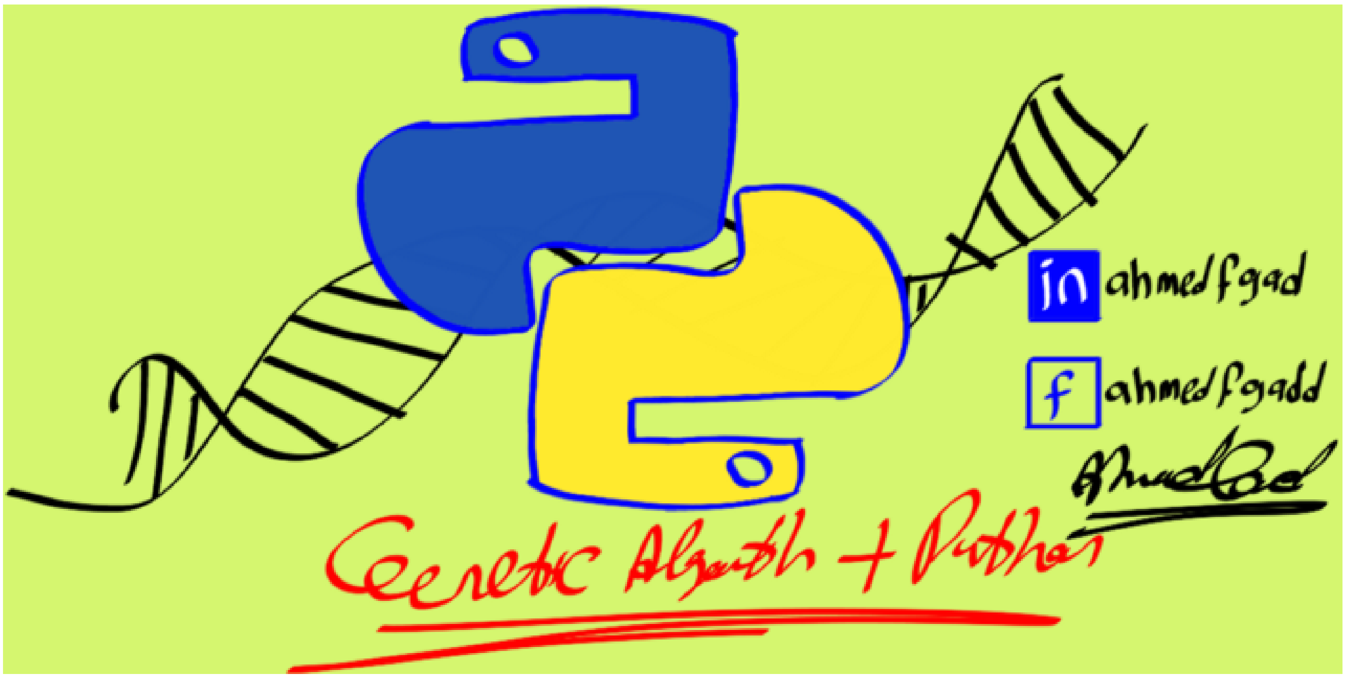
Genetic Algorithm Implementation in Python

Ahmed Gad
Jul 15, 2018 · 11 min read

This tutorial will implement the genetic algorithm optimization technique in Python based on a simple example in which we are trying to maximize the output of an equation. The tutorial uses the decimal representation for genes, one point crossover, and uniform mutation.



Genetic Algorithm Implementation in Python — By Ahmed F. Gad

Genetic Algorithm Overview

Flowchart of the genetic algorithm (GA) is shown in figure 1. Each step involved in the GA has some variations.

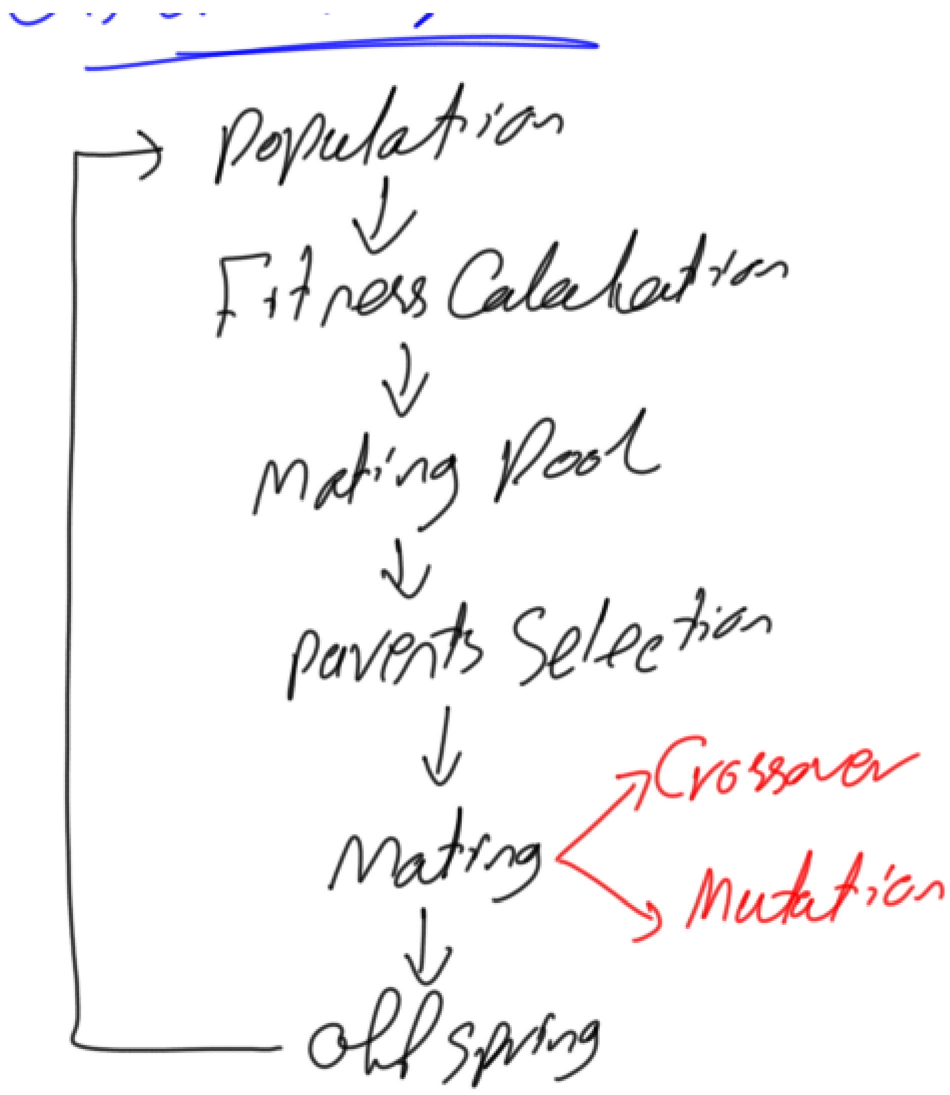


Figure 1. Genetic algorithm flowchart

For example, there are different types of representations for genes such as binary, decimal, integer, and others. Each type is treated differently. There are different types of mutation such as bit flip, swap, inverse, uniform, non-uniform, Gaussian, shrink, and others. Also, crossover has different types such as blend, one point, two points, uniform, and others. This tutorial will not implement all of them but just implements one type of each step involved in GA. The tutorial uses the decimal representation for genes, one point crossover, and uniform mutation. The Reader should have an understanding of how GA works. If not, please read this article titled "Introduction to Optimization with Genetic Algorithm" found in these links:

LinkedIn: https://www.linkedin.com/pulse/introduction-optimization-genetic-algorithm-ahmed-gad/

KDnuggets: https://www.kdnuggets.com/2018/03/introduction-optimization-with-genetic-algorithm.html

TowardsDataScience: https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b

SlideShare: https://www.slideshare.net/AhmedGadFCIT/introduction-to-optimization-with-genetic-algorithm-ga

Tutorial Example

The tutorial starts by presenting the equation that we are going to implement. The equation is shown below:

Y = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + w6x6

The equation has 6 inputs (x1 to x6) and 6 weights (w1 to w6) as shown and inputs values are (x1,x2,x3,x4,x5,x6)=(4,-2,7,5,11,1). We are looking to find the parameters (weights) that maximize such equation. The idea of maximizing such equation seems simple. The positive input is to be multiplied by the largest possible positive number and the negative number is to be multiplied by the smallest possible negative number. But the idea we are looking to implement is how to make GA do that its own in order to know that it is better to use positive weight with positive inputs and negative weights with negative inputs. Let us start implementing GA.

At first, let us create a list of the 6 inputs and a variable to hold the number of weights as follows:

```
# Inputs of the equation.
 equation_inputs = [4,-2,3.5,5,-11,-4.7]
 # Number of the weights we are looking to optimize.
 num_weights = 6
```

The next step is to define the initial population. Based on the number of weights, each chromosome (solution or individual) in the population will definitely have 6 genes, one gene for each weight. But the question is how many solutions per the population? There is no fixed value for that and we can select the value that fits well with our problem. But we could leave it generic so that it can be changed in the code. Next, we create a variable that holds the number of solutions per population, another to hold the size of the population, and finally, a variable that holds the actual initial population:

```
import numpy

sol_per_pop = 8

# Defining the population size.

pop_size = (sol_per_pop,num_weights) # The population will have
sol_per_pop chromosome where each chromosome has num_weights genes.

#Creating the initial population.

new_population = numpy.random.uniform(low=-4.0, high=4.0,
size=pop_size)
```

After importing the numpy library, we are able to create the initial population randomly using the numpy.random.uniform function. According to the selected parameters, it will be of shape (8, 6). That is 8 chromosomes and each one has 6 genes, one for each weight. After running this code, the population is as follows:

```
[[-2.19134006 -2.88907857  2.02365737 -3.97346034  3.45160502
2.05773249]

[ 2.12480298  2.97122243  3.60375452  3.78571392  0.28776565
3.5170347 ]

[ 1.81098962  0.35130155  1.03049548 -0.33163294  3.52586421
2.53845644]

[-0.63698911 -2.8638447   2.93392615 -1.40103767 -1.20313655
0.30567304]

[-1.48998583 -1.53845766  1.11905299 -3.67541087  1.33225142
2.86073836]

[ 1.14159503  2.88160332  1.74877772 -3.45854293  0.96125878
2.99178241]

[ 1.96561297  0.51030292  0.52852716 -1.56909315 -2.35855588
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -0.72163167  0.7516408
0.00677938]]
```

Note that it is generated randomly and thus it will definitely change when get run again.

After preparing the population, next is to follow the flowchart in figure 1. Based on the fitness function, we are going to select the best individuals within the current population as parents for mating. Next is to apply the GA variants (crossover and mutation) to produce the offspring of the next generation, creating the new population by appending both parents and offspring, and repeating such steps for a number of iterations/generations. The next code applies these steps:

```
import ga

num_generations = 5

num_parents_mating = 4
for generation in range(num_generations):
     # Measuring the fitness of each chromosome in the population.
     fitness = ga.cal_pop_fitness(equation_inputs, new_population)
    # Selecting the best parents in the population for mating.
     parents = ga.select_mating_pool(new_population, fitness,
                                      num_parents_mating)

     # Generating next generation using crossover.
     offspring_crossover = ga.crossover(parents,
                                       offspring_size=(pop_size[0]-
parents.shape[0], num_weights))

     # Adding some variations to the offsrping using mutation.
     offspring_mutation = ga.mutation(offspring_crossover)

# Creating the new population based on the parents and offspring.
     new_population[0:parents.shape[0], :] = parents
     new_population[parents.shape[0]:, :] = offspring_mutation
```

The current number of generations is 5. It is selected to be small for presenting results of all generations within the tutorial. There is a module named GA that holds the implementation of the algorithm.

The first step is to find the fitness value of each solution within the population using the ga `.cal_pop_fitness` function. The implementation of such function inside the GA module is as follows:

```
def cal_pop_fitness(equation_inputs, pop):
     # Calculating the fitness value of each solution in the current
population.
     # The fitness function calculates the sum of products between
each input and its corresponding weight.
```

```
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness
```

The fitness function accepts both the equation inputs values (x1 to x6) in addition to the population. The fitness value is calculated as the sum of product (SOP) between each input and its corresponding gene (weight) according to our function. According to the number of solutions per population, there will be a number of SOPs. As we previously set the number of solutions to 8 in the variable named `sol_per_pop`, there will be 8 SOPs as shown below:

```
[-63.41070188  14.40299221 -42.22532674  18.24112489 -45.44363278
-37.00404311  15.99527402  17.0688537 ]
```

Note that the higher the fitness value the better the solution.

After calculating the fitness values for all solutions, next is to select the best of them as parents in the mating pool according to the next function `ga.select_mating_pool`. Such function accepts the population, the fitness values, and the number of parents needed. It returns the parents selected. Its implementation inside the GA module is as follows:

```
def select_mating_pool(pop, fitness, num_parents):

    # Selecting the best individuals in the current generation as
parents for producing the offspring of the next generation.

    parents = numpy.empty((num_parents, pop.shape[1]))

    for parent_num in range(num_parents):

        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))

        max_fitness_idx = max_fitness_idx[0][0]

        parents[parent_num, :] = pop[max_fitness_idx, :]

        fitness[max_fitness_idx] = -99999999999

    return parents
```

Based on the number of parents required as defined in the variable `num_parents_mating`, the function creates an empty array to hold them as in this line:

```
parents = numpy.empty((num_parents, pop.shape[1]))
```

Looping through the current population, the function gets the index of the highest fitness value because it is the best solution to be selected according to this line:

```
max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
```

This index is used to retrieve the solution that corresponds to such fitness value using this line:

```
parents[parent_num, :] = pop[max_fitness_idx, :]
```

To avoid selecting such solution again, its fitness value is set to a very small value that is likely to not be selected again which is **-99999999999**. The **parents** array is returned finally which will be as follows according to our example:

```
[[-0.63698911 -2.8638447   2.93392615 -1.40103767 -1.20313655
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -0.72163167  0.7516408
0.00677938]

[ 1.96561297  0.51030292  0.52852716 -1.56909315 -2.35855588
2.29682254]
[ 2.12480298  2.97122243  3.60375452  3.78571392  0.28776565
3.5170347 ]]
```

Note that these three parents are the best individuals within the current population based on their fitness values which are 18.24112489, 17.0688537, 15.99527402, and 14.40299221, respectively.

Next step is to use such selected parents for mating in order to generate the offspring. The mating starts with the crossover operation according to the `ga.crossover` function. This function accepts the parents and the offspring size. It uses the offspring size to know the number of offspring to produce from such parents. Such a function is implemented as follows inside the GA module:

```
def crossover(parents, offspring_size):
     offspring = numpy.empty(offspring_size)
     # The point at which crossover takes place between two parents.
Usually, it is at the center.
     crossover_point = numpy.uint8(offspring_size[1]/2)

     for k in range(offspring_size[0]):
         # Index of the first parent to mate.
         parent1_idx = k%parents.shape[0]
         # Index of the second parent to mate.
         parent2_idx = (k+1)%parents.shape[0]
         # The new offspring will have its first half of its genes
taken from the first parent.
         offspring[k, 0:crossover_point] = parents[parent1_idx,
0:crossover_point]
         # The new offspring will have its second half of its genes
taken from the second parent.
         offspring[k, crossover_point:] = parents[parent2_idx,
crossover_point:]
     return offspring
```

The function starts by creating an empty array based on the offspring size as in this line:

```
offspring = numpy.empty(offspring_size)
```

Because we are using single point crossover, we need to specify the point at which crossover takes place. The point is selected to divide the solution into two equal halves according to this line:

```
crossover_point = numpy.uint8(offspring_size[1]/2)
```

Then we need to select the two parents to crossover. The indices of these parents are selected according to these two lines:

```
parent1_idx = k%parents.shape[0]
parent2_idx = (k+1)%parents.shape[0]
```

The parents are selected in a way similar to a ring. The first with indices 0 and 1 are selected at first to produce two offspring. If there still remaining offspring to produce, then we select the parent 1 with parent 2 to produce another two offspring. If we are in need of more offspring, then we select the next two parents with indices 2 and 3. By index 3, we reached the last parent. If we need to produce more offspring, then we select parent with index 3 and go back to the parent with index 0, and so on.

The solutions after applying the crossover operation to the parents are stored into the `offspring` variable and they are as follows:

```
[[-0.63698911 -2.8638447   2.93392615 -0.72163167  0.7516408
0.00677938]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.35855588
2.29682254]

[ 1.96561297  0.51030292  0.52852716  3.78571392  0.28776565
3.5170347 ]

[ 2.12480298  2.97122243  3.60375452 -1.40103767 -1.20313655
0.30567304]]
```

Next is to apply the second GA variant, mutation, to the results of the crossover stored in the `offspring` variable using the `ga.mutation` function inside the GA module. Such function accepts the crossover offspring and returns them after applying uniform mutation. That function is implemented as follows:

```python
def mutation(offspring_crossover):

    # Mutation changes a single gene in each offspring randomly.

    for idx in range(offspring_crossover.shape[0]):

        # The random value to be added to the gene.

        random_value = numpy.random.uniform(-1.0, 1.0, 1)

        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] +
random_value

    return offspring_crossover
```

It loops through each offspring and adds a uniformly generated random number in the range from -1 to 1 according to this line:

```
random_value = numpy.random.uniform(-1.0, 1.0, 1)
```

Such random number is then added to the gene with index 4 of the offspring according to this line:

```
offspring_crossover[idx, 4] = offspring_crossover[idx, 4] +
random_value
```

Note that the index could be changed to any other index. The offspring after applying mutation are as follows:

```
[[-0.63698911 -2.8638447   2.93392615 -0.72163167  1.66083721
0.00677938]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]

[ 1.96561297  0.51030292  0.52852716  3.78571392  0.45337472
3.5170347 ]

[ 2.12480298  2.97122243  3.60375452 -1.40103767 -1.5781162
0.30567304]]
```

Such results are added to the variable `offspring_crossover` and got returned by the function.

At this point, we successfully produced 4 offspring from the 4 selected parents and we are ready to create the new population of the next generation.

Note that GA is a random-based optimization technique. It tries to enhance the current solutions by applying some random changes to them. Because such changes are random, we are not sure that they will produce better solutions. For such reason, it is preferred to keep the previous best solutions (parents) in the new population. In the worst case when all the new offspring are worse than such parents, we will continue using such parents. As a result, we guarantee that the new generation will at least preserve the previous good results and will not go worse. The new population will have its first 4 solutions from the previous parents. The last 4 solutions come from the offspring created after applying crossover and mutation:

```
new_population[0:parents.shape[0], :] = parents
new_population[parents.shape[0]:, :] = offspring_mutation
```

By calculating the fitness of all solutions (parents and offspring) of the first generation, their fitness is as follows:

```
[ 18.24112489  17.0688537   15.99527402  14.40299221  -8.46075629
31.73289712   6.10307563  24.08733441]
```

The highest fitness previously was **18.24112489** but now it is **31.7328971158**. That means that the random changes moved towards a better solution. This is GREAT. But such results could be enhanced by going through more generations. Below are the results of each step for another 4 generations:

```
Generation :  1

Fitness values:

[ 18.24112489  17.0688537   15.99527402  14.40299221  -8.46075629
31.73289712   6.10307563  24.08733441]

Selected parents:

[[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]

[ 2.12480298  2.97122243  3.60375452 -1.40103767 -1.5781162
0.30567304]

[-0.63698911 -2.8638447   2.93392615 -1.40103767 -1.20313655
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -0.72163167  0.7516408
0.00677938]]

Crossover result:

[[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.5781162
0.30567304]

[ 2.12480298  2.97122243  3.60375452 -1.40103767 -1.20313655
0.30567304]

[-0.63698911 -2.8638447   2.93392615 -0.72163167  0.7516408
0.00677938]
[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]]

Mutation result:

[[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.2392086
0.30567304]

[ 2.12480298  2.97122243  3.60375452 -1.40103767 -0.38610586
0.30567304]

[-0.63698911 -2.8638447   2.93392615 -0.72163167  1.33639943
0.00677938]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.13941727
2.29682254]]

Best result after generation 1 :  34.1663669207

Generation :  2

Fitness values:

[ 31.73289712  24.08733441  18.24112489  17.0688537   34.16636692
10.97522073  -4.89194068  22.86998223]

Selected Parents:

[[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.2392086
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]

[ 2.12480298  2.97122243  3.60375452 -1.40103767 -1.5781162
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.13941727
2.29682254]]

Crossover result:

[[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.5781162
0.30567304]

[ 2.12480298  2.97122243  3.60375452 -1.56909315 -1.13941727
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.2392086
0.30567304]]
```

```
Mutation result:

[[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.20515009
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -0.73543721
0.30567304]

[ 2.12480298  2.97122243  3.60375452 -1.56909315 -0.50581509
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.20089639
0.30567304]]

Best result after generation 2:  34.5930432629

Generation :  3

Fitness values:

[ 34.16636692  31.73289712  24.08733441  22.86998223  34.59304326
28.6248816    2.09334217  33.7449326 ]

Selected parents:

[[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.20515009
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.2392086
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.20089639
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]]

Crossover result:

[[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.2392086
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.20089639
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -1.94513681
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.20515009
2.29682254]]

Mutation result:
```

```
[[ 3.00912373 -2.745417    3.27131287 -1.40103767 -2.20744102
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.16589294
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.37553107
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.44124005
2.29682254]]

Best result after generation 3:  44.8169235189

Generation :  4

Fitness values

[ 34.59304326  34.16636692  33.7449326   31.73289712  44.81692352

33.35989464  36.46723397  37.19003273]

Selected parents:

[[ 3.00912373 -2.745417    3.27131287 -1.40103767 -2.20744102
0.30567304]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.44124005
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.37553107
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.20515009
2.29682254]]

Crossover result:

[[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.37553107
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.20515009
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -2.20744102
0.30567304]]

Mutation result:

[[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.13382082
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.98105233
2.29682254]
[ 3.00912373 -2.745417    3.27131287 -1.56909315 -2.27638584
2.29682254]

[ 3.00912373 -2.745417    3.27131287 -1.40103767 -1.70558545
0.30567304]]

Best result after generation 4:  44.8169235189
```

After the above 5 generations, the best result now has a fitness value equal to **44.8169235189** compared to the best result after the first generation which is **18.24112489**.

The best solution has the following weights:

```
[3.00912373 -2.745417    3.27131287 -1.40103767 -2.20744102
0.30567304]
```

Complete Python Implementation

The complete code is available in my GitHub account here https://github.com/ahmedfgad/GeneticAlgorithmPython

It will be listed in the tutorial too.

Here is the implementation of the example:

```python
import numpy
import ga

"""
The y=target is to maximize this equation ASAP:
    y = w1x1+w2x2+w3x3+w4x4+w5x5+6wx6
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)
    What are the best values for the 6 weights w1 to w6?
    We are going to use the genetic algorithm for the best possible values after a numb
"""

# Inputs of the equation.
equation_inputs = [4,-2,3.5,5,-11,-4.7]

# Number of the weights we are looking to optimize.
num_weights = 6

```

```python
"""
Genetic algorithm parameters:
    Mating pool size
    Population size
"""
sol_per_pop = 8
num_parents_mating = 4

# Defining the population size.
pop_size = (sol_per_pop,num_weights) # The population will have sol_per_pop chromosome
#Creating the initial population.
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)

num_generations = 5
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measing the fitness of each chromosome in the population.
    fitness = ga.cal_pop_fitness(equation_inputs, new_population)

    # Selecting the best parents in the population for mating.
    parents = ga.select_mating_pool(new_population, fitness,
                                      num_parents_mating)

    # Generating next generation using crossover.
    offspring_crossover = ga.crossover(parents,
                                       offspring_size=(pop_size[0]-parents.shape[0], nu

    # Adding some variations to the offsrping using mutation.
    offspring_mutation = ga.mutation(offspring_crossover)

    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation

    # The best result in the current iteration.
    print("Best result : ", numpy.max(numpy.sum(new_population*equation_inputs, axis=1

# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final generation.
fitness = ga.cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])
```

The GA module is as follows:

```python
import numpy

def cal_pop_fitness(equation_inputs, pop):
    # Calculating the fitness value of each solution in the current population.
    # The fitness function caulcuates the sum of products between each input and its c
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness

def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents for producin
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents

def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    # The point at which crossover takes place between two parents. Usually it is at t
    crossover_point = numpy.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken from the first
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        # The new offspring will have its second half of its genes taken from the seco
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring

def mutation(offspring_crossover):
    # Mutation changes a single gene in each offspring randomly.
    for idx in range(offspring_crossover.shape[0]):
        # The random value to be added to the gene.
        random_value = numpy.random.uniform(-1.0, 1.0, 1)
        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] + random_value
    return offspring_crossover
```

ga.py hosted with ❤ by **GitHub** view raw

. . .

The original article is available at LinkedIn at this page: https://www.linkedin.com/pulse/genetic-algorithm-implementation-python-ahmed-gad/

. . .

For contacting the author:

LinkedIn: https://linkedin.com/in/ahmedfgad

E-mail: ahmed.f.gad@gmail.com

Machine Learning Data Science Optimization Genetic Algorithm Evolutionary Algorithms