# Лабораторная робота №4

**Тема:** Предсказание временных рядов с помощью генетического алгоритма (ГА).

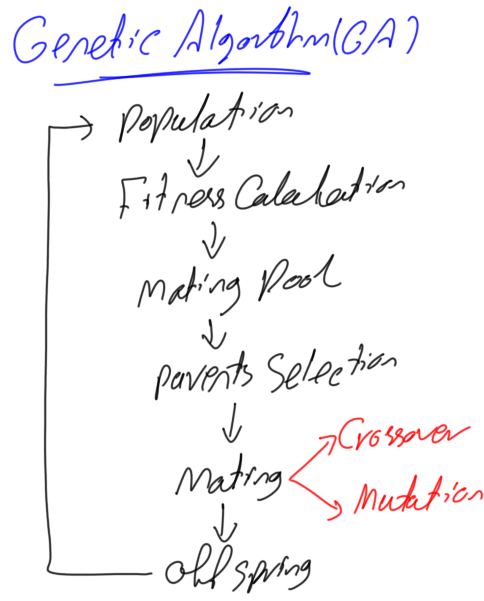
**Цель:** получение практических навыков генетической настройки нейросети.

**Ход работы:**

Итак, начнём с того, что нейронная сети, построенная в прошлой лабораторной работе это, конечно, бесспорно хорошее дело, но больно уж хлопотное и долгое. Под последним понимается не сколько сам процесс создания, сколько обучения. Решением данной проблемы, как раз и стал генетический алгоритм.

В данной лабораторной будет реализован метод оптимизации генетического алгоритма в Python, основанный на простом примере, в котором мы пытаемся максимизировать вывод уравнения. Стоит уточнить, что мы будем использовать десятичное представление для генов, пересечение одной точки и равномерную мутацию.

Блок-схема генетического алгоритма (GA) показана на рисунке ниже. Что интересно, каждый шаг в GA имеет свои вариации.



Например, существуют разные типы представлений для генов, такие как двоичные, десятичные, целые и другие. Каждый тип рассматривается по-разному. Существуют различные типы мутаций, такие как переворот, перестановка, инверсия, равномерность, неравномерность, гауссов, усадка и другие. Кроме того, кроссовер имеет разные типы, такие как смесь, одна точка, две точки, униформа и другие.

В лабораторной не будут реализованы все-все из них, а будет реализован только один тип каждого шага в GA.

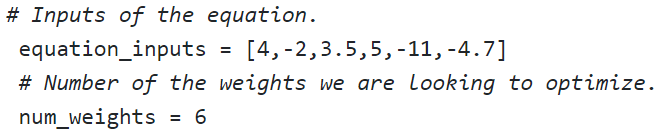
Итак, начнём!

Вот наша исходная формула:

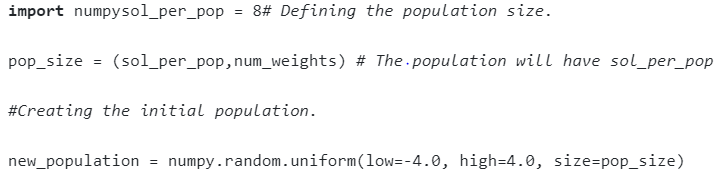
**Y = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + w6x6**

Уравнение имеет 6 входных данных (от x1 до x6) и 6 весовых коэффициентов (от w1 до w6), как показано, а входные значения (x1, x2, x3, x4, x5, x6) = (4, -2,7,5,11, 1). Мы ищем параметры (веса), которые максимизируют такое уравнение. Идея максимизации такого уравнения кажется простой. Положительное значение должно быть умножено на максимально возможное положительное число, а отрицательное число должно быть умножено на наименьшее возможное отрицательное число. Но идея, которую мы ищем, заключается в том, чтобы заставить GA делать это самостоятельно, чтобы знать, что лучше использовать положительный вес с положительными входами и отрицательный вес с отрицательными входами.

Сначала давайте создадим список из 6 входов и переменную для хранения количества весов следующим образом:



Следующим шагом является определение начальной популяции. Исходя из количества весов, каждая хромосома (раствор или индивидуум) в популяции определённо будет иметь 6 генов, по одному гену на каждый вес. Но вопрос в том, сколько решений на население? Для этого нет фиксированного значения, и мы можем выбрать значение, которое хорошо соответствует нашей проблеме. Но мы можем оставить его универсальным, чтобы его можно было изменить в коде. Затем мы создаём переменную, которая содержит количество решений для каждой совокупности, другую - для хранения размера совокупности и, наконец, переменную, которая содержит фактическую начальную совокупность:



После импорта библиотеки numpy мы можем создать начальную популяцию случайным образом с помощью функции numpy.random.uniform. По выбранным параметрам он будет иметь форму (8, 6). Это 8 хромосом, и у каждой есть 6 генов, по одному на каждый вес. После запуска этого кода, население выглядит следующим образом:

[[-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]]

После подготовки популяции следующим будет следовать блок-схеме на рисунке 1. На основе функции пригодности мы выберем лучших людей из текущей популяции в качестве родителей для спаривания. Далее следует применить варианты GA (кроссовер и мутация) для получения потомства следующего поколения, создания новой популяции путем добавления обоих родителей и потомства и повторения таких шагов для ряда итераций / поколений. Следующий код применяет эти шаги:

**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

Текущее количество поколений - 5. Существует модуль с именем GA, который содержит реализацию алгоритма.

Первым шагом является определение пригодности каждого решения в популяции с использованием.cal\_pop\_fitnessфункция. Реализация такой функции внутри модуля GA выглядит следующим образом:

**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

Функция пригодности принимает оба входных значения уравнения (от х1 до х6) в дополнение к совокупности. Значение пригодности рассчитывается как сумма продукта (СОП) между каждым входом и его соответствующим геном (весом) в соответствии с нашей функцией В зависимости от количества решений на одну группу населения будет определено количество СОП. Как мы ранее установили количество решений до 8 в переменной с именемsol\_per\_popбудет 8 СОП, как показано ниже:

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

Обратите внимание, что чем выше значение пригодности, тем лучше решение.

После расчета значений пригодности для всех решений, следующим является выбор лучших из них в качестве родителей в пуле в соответствии со следующей функцией.ga.select\_mating\_pool, Такая функция принимает население, показатели пригодности и количество родителей, необходимых. Возвращает выбранных родителей. Его реализация внутри модуля GA выглядит следующим образом:

**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

На основе количества родителей, требуемого, как определено в переменной num\_parents\_mating, функция создает пустой массив для их хранения, как в этой строке:

parents = numpy.empty((num\_parents, pop.shape[1]))

Зацикливаясь на текущей совокупности, функция получает индекс наибольшего значения пригодности, потому что это лучшее решение, которое будет выбрано согласно этой строке:

max\_fitness\_idx = numpy.where(fitness == numpy.max(fitness))

Этот индекс используется для получения решения, соответствующего значению пригодности, с использованием этой строки:

parents[parent\_num, :] = pop[max\_fitness\_idx, :]

Чтобы избежать повторного выбора такого решения, его пригодность установлена ​​на очень маленькое значение, которое, скорее всего, больше не будет выбрано, что -99999999999, родители. В конце возвращается массив, который будет выглядеть следующим образом:

[[-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 ]]

Обратите внимание, что эти три родителя являются лучшими людьми в текущей популяции на основании их значений пригодности, которые составляют 18,24112489, 17,0688537, 15,99527402 и 14,40299221 соответственно.

Следующим шагом будет использование таких отобранных родителей для спаривания, чтобы произвести потомство. Спаривание начинается с операции кроссовера в соответствии сga.crossoverфункция. Эта функция принимает родителей и размер потомства. Он использует размер потомства, чтобы узнать количество потомков, произведенных от таких родителей. Такая функция реализована в модуле GA следующим образом:

**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

Функция начинается с создания пустого массива на основе размера потомства, как в этой строке:

offspring = numpy.empty(offspring\_size)

Поскольку мы используем пересечение в одной точке, нам нужно указать точку, в которой происходит пересечение. Точка выбирается для разделения решения на две равные половины в соответствии с этой линией:

crossover\_point = numpy.uint8(offspring\_size[1]/2)

Затем нам нужно выбрать двух родителей для кроссовера. Индексы этих родителей выбираются в соответствии с этими двумя линиями:

parent1\_idx = k%parents.shape[0]  
parent2\_idx = (k+1)%parents.shape[0]

Родители выбираются так же, как кольцо. Первые с индексами 0 и 1 выбираются сначала, чтобы произвести двух потомков. Если еще есть потомство для производства, тогда мы выбираем родителя 1 с родителем 2, чтобы произвести еще двух потомков. Если нам нужно больше потомства, то мы выбираем следующих двух родителей с индексами 2 и 3. По индексу 3 мы достигли последнего родителя. Если нам нужно произвести больше потомков, то мы выбираем родителя с индексом 3 и возвращаемся к родителю с индексом 0 и так далее.

Решения после применения кроссовера для родителей сохраняются вoffspring переменные и они следующие:

[[-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]]

Далее следует применить второй вариант GA, мутацию, к результатам кроссовера, сохраненным вoffspring переменная с помощью ga.mutationфункция внутри модуля GA. Такая функция принимает потомство кроссовера и возвращает его после применения равномерной мутации. Эта функция реализована следующим образом:

**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

Он перебирает каждого потомка и добавляет равномерно сгенерированное случайное число в диапазоне от -1 до 1 в соответствии с этой строкой:

random\_value = numpy.random.uniform(-1.0, 1.0, 1)

Такое случайное число затем добавляется к гену с индексом 4 потомства в соответствии с этой строкой:

offspring\_crossover[idx, 4] = offspring\_crossover[idx, 4] + random\_value

Обратите внимание, что индекс может быть изменен на любой другой индекс. Потомки после применения мутации выглядят следующим образом:

[[-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]]

Такие результаты добавляются в переменнуюoffspring\_crossoverи был возвращен функцией.

На данный момент мы успешно произвели 4 потомства от 4 отобранных родителей, и мы готовы создать новую популяцию следующего поколения.

Обратите внимание, что GA является методом случайной оптимизации. Он пытается улучшить текущие решения, применяя к ним некоторые случайные изменения. Поскольку такие изменения являются случайными, мы не уверены, что они приведут к лучшим решениям. По этой причине предпочтительнее сохранить предыдущие лучшие решения (родителей) в новой популяции. В худшем случае, когда все новые дети хуже, чем такие родители, мы будем продолжать использовать таких родителей. В результате мы гарантируем, что новое поколение, по крайней мере, сохранит предыдущие хорошие результаты и не станет хуже. У нового населения будут первые 4 решения от предыдущих родителей. Последние 4 решения получены от потомков, созданных после применения кроссовера и мутации:

new\_population[0:parents.shape[0], :] = parents  
new\_population[parents.shape[0]:, :] = offspring\_mutation

При расчете пригодности всех решений (родителей и потомков) первого поколения их пригодность выглядит следующим образом:

[ 18.24112489 17.0688537 15.99527402 14.40299221 -8.46075629 31.73289712 6.10307563 24.08733441]

Самая высокая пригодность ранее была 18.24112489 но теперь это +31,7328971158, Это означает, что случайные изменения сместились в сторону лучшего решения. Это здорово. Но такие результаты можно улучшить, пройдя через несколько поколений Ниже приведены результаты каждого шага еще для 4 поколений:

**Generation : 1Fitness 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 : 34.1663669207Generation : 2Fitness 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: 34.5930432629Generation : 3Fitness 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: 44.8169235189Generation : 4Fitness values**[ 34.59304326 34.16636692 33.7449326 31.73289712 44.8169235233.35989464 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**

После вышеуказанных 5 поколений лучший результат теперь имеет значение пригодности, равное +44,8169235189 по сравнению с лучшим результатом после первого поколения, которое 18.24112489.

Лучшее решение имеет следующие веса:

[3.00912373 -2.745417 3.27131287 -1.40103767 -2.20744102 0.30567304]

Ниже представлена полная реализация самого примера:

|  |
| --- |
| importnumpy |
|  | 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 number of generations. |
|  | """ |
|  |  |
|  | # 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 |
|  |  |
|  | """ |
|  | 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 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) |
|  | 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], 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 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]) |

А также и самого модуля GA:

|  |
| --- |
| Importnumpy |
|  |  |
|  | 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 corresponding weight. |
|  | 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 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 |
|  |  |
|  | 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 |
|  |  |
|  | 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 |