**Genetic Algorithm**

The [Genetic Algorithm](https://en.wikipedia.org/wiki/Genetic_algorithm) is a stochastic global search optimization algorithm.

It is inspired by the biological theory of evolution by means of natural selection. Specifically, the new synthesis that combines an understanding of genetics with the theory.

*Genetic algorithms (algorithm 9.4) borrow inspiration from biological evolution, where fitter individuals are more likely to pass on their genes to the next generation.*

— Page 148, [Algorithms for Optimization](https://amzn.to/2Traqek), 2019.

The algorithm uses analogs of a genetic representation (bitstrings), fitness (function evaluations), genetic recombination (crossover of bitstrings), and mutation (flipping bits).

The algorithm works by first creating a population of a fixed size of random bitstrings. The main loop of the algorithm is repeated for a fixed number of iterations or until no further improvement is seen in the best solution over a given number of iterations.

One iteration of the algorithm is like an evolutionary generation.

First, the population of bitstrings (candidate solutions) are evaluated using the objective function. The objective function evaluation for each candidate solution is taken as the fitness of the solution, which may be minimized or maximized.

Then, parents are selected based on their fitness. A given candidate solution may be used as parent zero or more times. A simple and effective approach to selection involves drawing *k* candidates from the population randomly and selecting the member from the group with the best fitness. This is called tournament selection where *k* is a hyperparameter and set to a value such as 3. This simple approach simulates a more costly fitness-proportionate selection scheme.

*In tournament selection, each parent is the fittest out of k randomly chosen chromosomes of the population*

— Page 151, [Algorithms for Optimization](https://amzn.to/2Traqek), 2019.

Parents are used as the basis for generating the next generation of candidate points and one parent for each position in the population is required.

Parents are then taken in pairs and used to create two children. Recombination is performed using a crossover operator. This involves selecting a random split point on the bit string, then creating a child with the bits up to the split point from the first parent and from the split point to the end of the string from the second parent. This process is then inverted for the second child.

For example the two parents:

* parent1 = 00000
* parent2 = 11111

May result in two cross-over children:

* child1 = 00011
* child2 = 11100

This is called one point crossover, and there are many other variations of the operator.

Crossover is applied probabilistically for each pair of parents, meaning that in some cases, copies of the parents are taken as the children instead of the recombination operator. Crossover is controlled by a hyperparameter set to a large value, such as 80 percent or 90 percent.

*Crossover is the Genetic Algorithm’s distinguishing feature. It involves mixing and matching parts of two parents to form children. How you do that mixing and matching depends on the representation of the individuals.*

— Page 36, [Essentials of Metaheuristics](https://amzn.to/2HxZVn4), 2011.

Mutation involves flipping bits in created children candidate solutions. Typically, the mutation rate is set to *1/L*, where *L* is the length of the bitstring.

*Each bit in a binary-valued chromosome typically has a small probability of being flipped. For a chromosome with m bits, this mutation rate is typically set to 1/m, yielding an average of one mutation per child chromosome.*

— Page 155, [Algorithms for Optimization](https://amzn.to/2Traqek), 2019.

For example, if a problem used a bitstring with 20 bits, then a good default mutation rate would be (1/20) = 0.05 or a probability of 5 percent.

This defines the simple genetic algorithm procedure. It is a large field of study, and there are many extensions to the algorithm.

Now that we are familiar with the simple genetic algorithm procedure, let’s look at how we might implement it from scratch.

**Genetic Algorithm From Scratch**

In this section, we will develop an implementation of the genetic algorithm.

The first step is to create a population of random bitstrings. We could use boolean values *True* and *False*, string values ‘0’ and ‘1’, or integer values 0 and 1. In this case, we will use integer values.

We can generate an array of integer values in a range using the [randint() function](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html), and we can specify the range as values starting at 0 and less than 2, e.g. 0 or 1. We will also represent a candidate solution as a list instead of a NumPy array to keep things simple.

An initial population of random bitstring can be created as follows, where “*n\_pop*” is a hyperparameter that controls the population size and “*n\_bits*” is a hyperparameter that defines the number of bits in a single candidate solution:

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# initial population of random bitstring**  **pop = [randint(0, 2, n\_bits).tolist() for \_ in range(n\_pop)]** |

Next, we can enumerate over a fixed number of algorithm iterations, in this case, controlled by a hyperparameter named “*n\_iter*“.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **...**  **# enumerate generations**  **for gen in range(n\_iter):**  **...** |

The first step in the algorithm iteration is to evaluate all candidate solutions.

We will use a function named *objective()* as a generic objective function and call it to get a fitness score, which we will minimize.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# evaluate all candidates in the population**  **scores = [objective(c) for c in pop]** |

We can then select parents that will be used to create children.

The tournament selection procedure can be implemented as a function that takes the population and returns one selected parent. The *k* value is fixed at 3 with a default argument, but you can experiment with different values if you like.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **# tournament selection**  **def selection(pop, scores, k=3):**  **# first random selection**  **selection\_ix = randint(len(pop))**  **for ix in randint(0, len(pop), k-1):**  **# check if better (e.g. perform a tournament)**  **if scores[ix] < scores[selection\_ix]:**  **selection\_ix = ix**  **return pop[selection\_ix]** |

We can then call this function one time for each position in the population to create a list of parents

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# select parents**  **selected = [selection(pop, scores) for \_ in range(n\_pop)]** |

We can then create the next generation.

This first requires a function to perform crossover. This function will take two parents and the crossover rate. The crossover rate is a hyperparameter that determines whether crossover is performed or not, and if not, the parents are copied into the next generation. It is a probability and typically has a large value close to 1.0.

The *crossover()* function below implements crossover using a draw of a random number in the range [0,1] to determine if crossover is performed, then selecting a valid split point if crossover is to be performed.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **# crossover two parents to create two children**  **def crossover(p1, p2, r\_cross):**  **# children are copies of parents by default**  **c1, c2 = p1.copy(), p2.copy()**  **# check for recombination**  **if rand() < r\_cross:**  **# select crossover point that is not on the end of the string**  **pt = randint(1, len(p1)-2)**  **# perform crossover**  **c1 = p1[:pt] + p2[pt:]**  **c2 = p2[:pt] + p1[pt:]**  **return [c1, c2]** |

We also need a function to perform mutation.

This procedure simply flips bits with a low probability controlled by the “*r\_mut*” hyperparameter.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **# mutation operator**  **def mutation(bitstring, r\_mut):**  **for i in range(len(bitstring)):**  **# check for a mutation**  **if rand() < r\_mut:**  **# flip the bit**  **bitstring[i] = 1 - bitstring[i]** |

We can then loop over the list of parents and create a list of children to be used as the next generation, calling the crossover and mutation functions as needed.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **...**  **# create the next generation**  **children = list()**  **for i in range(0, n\_pop, 2):**  **# get selected parents in pairs**  **p1, p2 = selected[i], selected[i+1]**  **# crossover and mutation**  **for c in crossover(p1, p2, r\_cross):**  **# mutation**  **mutation(c, r\_mut)**  **# store for next generation**  **children.append(c)** |

We can tie all of this together into a function named *genetic\_algorithm()* that takes the name of the objective function and the hyperparameters of the search, and returns the best solution found during the search

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31** | **# genetic algorithm**  **def genetic\_algorithm(objective, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut):**  **# initial population of random bitstring**  **pop = [randint(0, 2, n\_bits).tolist() for \_ in range(n\_pop)]**  **# keep track of best solution**  **best, best\_eval = 0, objective(pop[0])**  **# enumerate generations**  **for gen in range(n\_iter):**  **# evaluate all candidates in the population**  **scores = [objective(c) for c in pop]**  **# check for new best solution**  **for i in range(n\_pop):**  **if scores[i] < best\_eval:**  **best, best\_eval = pop[i], scores[i]**  **print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))**  **# select parents**  **selected = [selection(pop, scores) for \_ in range(n\_pop)]**  **# create the next generation**  **children = list()**  **for i in range(0, n\_pop, 2):**  **# get selected parents in pairs**  **p1, p2 = selected[i], selected[i+1]**  **# crossover and mutation**  **for c in crossover(p1, p2, r\_cross):**  **# mutation**  **mutation(c, r\_mut)**  **# store for next generation**  **children.append(c)**  **# replace population**  **pop = children**  **return [best, best\_eval]** |

Now that we have developed an implementation of the genetic algorithm, let’s explore how we might apply it to an objective function.

## Genetic Algorithm for OneMax

In this section, we will apply the genetic algorithm to a binary string-based optimization problem.

The problem is called OneMax and evaluates a binary string based on the number of 1s in the string. For example, a bitstring with a length of 20 bits will have a score of 20 for a string of all 1s.

Given we have implemented the genetic algorithm to minimize the objective function, we can add a negative sign to this evaluation so that large positive values become large negative values.

The onemax() function below implements this and takes a bitstring of integer values as input and returns the negative sum of the values.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **# objective function**  **def onemax(x):**  **return -sum(x)** |

Next, we can configure the search.

The search will run for 100 iterations and we will use 20 bits in our candidate solutions, meaning the optimal fitness will be -20.0.

The population size will be 100, and we will use a crossover rate of 90 percent and a mutation rate of 5 percent. This configuration was chosen after a little trial and error.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11** | **...**  **# define the total iterations**  **n\_iter = 100**  **# bits**  **n\_bits = 20**  **# define the population size**  **n\_pop = 100**  **# crossover rate**  **r\_cross = 0.9**  **# mutation rate**  **r\_mut = 1.0 / float(n\_bits)** |

The search can then be called and the best result reported.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **...**  **# perform the genetic algorithm search**  **best, score = genetic\_algorithm(onemax, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut)**  **print('Done!')**  **print('f(%s) = %f' % (best, score))** |

Tying this together, the complete example of applying the genetic algorithm to the OneMax objective function is listed below.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78**  **79**  **80**  **81**  **82**  **83**  **84**  **85** | **# genetic algorithm search of the one max optimization problem**  **from numpy.random import randint**  **from numpy.random import rand**    **# objective function**  **def onemax(x):**  **return -sum(x)**    **# tournament selection**  **def selection(pop, scores, k=3):**  **# first random selection**  **selection\_ix = randint(len(pop))**  **for ix in randint(0, len(pop), k-1):**  **# check if better (e.g. perform a tournament)**  **if scores[ix] < scores[selection\_ix]:**  **selection\_ix = ix**  **return pop[selection\_ix]**    **# crossover two parents to create two children**  **def crossover(p1, p2, r\_cross):**  **# children are copies of parents by default**  **c1, c2 = p1.copy(), p2.copy()**  **# check for recombination**  **if rand() < r\_cross:**  **# select crossover point that is not on the end of the string**  **pt = randint(1, len(p1)-2)**  **# perform crossover**  **c1 = p1[:pt] + p2[pt:]**  **c2 = p2[:pt] + p1[pt:]**  **return [c1, c2]**    **# mutation operator**  **def mutation(bitstring, r\_mut):**  **for i in range(len(bitstring)):**  **# check for a mutation**  **if rand() < r\_mut:**  **# flip the bit**  **bitstring[i] = 1 - bitstring[i]**    **# genetic algorithm**  **def genetic\_algorithm(objective, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut):**  **# initial population of random bitstring**  **pop = [randint(0, 2, n\_bits).tolist() for \_ in range(n\_pop)]**  **# keep track of best solution**  **best, best\_eval = 0, objective(pop[0])**  **# enumerate generations**  **for gen in range(n\_iter):**  **# evaluate all candidates in the population**  **scores = [objective(c) for c in pop]**  **# check for new best solution**  **for i in range(n\_pop):**  **if scores[i] < best\_eval:**  **best, best\_eval = pop[i], scores[i]**  **print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))**  **# select parents**  **selected = [selection(pop, scores) for \_ in range(n\_pop)]**  **# create the next generation**  **children = list()**  **for i in range(0, n\_pop, 2):**  **# get selected parents in pairs**  **p1, p2 = selected[i], selected[i+1]**  **# crossover and mutation**  **for c in crossover(p1, p2, r\_cross):**  **# mutation**  **mutation(c, r\_mut)**  **# store for next generation**  **children.append(c)**  **# replace population**  **pop = children**  **return [best, best\_eval]**    **# define the total iterations**  **n\_iter = 100**  **# bits**  **n\_bits = 20**  **# define the population size**  **n\_pop = 100**  **# crossover rate**  **r\_cross = 0.9**  **# mutation rate**  **r\_mut = 1.0 / float(n\_bits)**  **# perform the genetic algorithm search**  **best, score = genetic\_algorithm(onemax, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut)**  **print('Done!')**  **print('f(%s) = %f' % (best, score))** |

Running the example will report the best result as it is found along the way, then the final best solution at the end of the search, which we would expect to be the optimal solution.

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that the search found the optimal solution after about eight generations.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | **>0, new best f([1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]) = -14.000**  **>0, new best f([1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0]) = -15.000**  **>1, new best f([1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1]) = -16.000**  **>2, new best f([0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]) = -17.000**  **>2, new best f([1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -19.000**  **>8, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000**  **Done!**  **f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000** |

## Genetic Algorithm for Continuous Function Optimization

Optimizing the OneMax function is not very interesting; we are more likely to want to optimize a continuous function.

For example, we can define the x^2 minimization function that takes input variables and has an optima at  f(0, 0) = 0.0.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **# objective function**  **def objective(x):**  **return x[0]\*\*2.0 + x[1]\*\*2.0** |

We can minimize this function with a genetic algorithm.

First, we must define the bounds of each input variable.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# define range for input**  **bounds = [[-5.0, 5.0], [-5.0, 5.0]]** |

We will take the “n\_bits” hyperparameter as a number of bits per input variable to the objective function and set it to 16 bits.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# bits per variable**  **n\_bits = 16** |

This means our actual bit string will have (16 \* 2) = 32 bits, given the two input variables.

We must update our mutation rate accordingly.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# mutation rate**  **r\_mut = 1.0 / (float(n\_bits) \* len(bounds))** |

Next, we need to ensure that the initial population creates random bitstrings that are large enough.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **...**  **# initial population of random bitstring**  **pop = [randint(0, 2, n\_bits\*len(bounds)).tolist() for \_ in range(n\_pop)]** |

Finally, we need to decode the bitstrings to numbers prior to evaluating each with the objective function.

We can achieve this by first decoding each substring to an integer, then scaling the integer to the desired range. This will give a vector of values in the range that can then be provided to the objective function for evaluation.

The decode() function below implements this, taking the bounds of the function, the number of bits per variable, and a bitstring as input and returns a list of decoded real values.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17** | **# decode bitstring to numbers**  **def decode(bounds, n\_bits, bitstring):**  **decoded = list()**  **largest = 2\*\*n\_bits**  **for i in range(len(bounds)):**  **# extract the substring**  **start, end = i \* n\_bits, (i \* n\_bits)+n\_bits**  **substring = bitstring[start:end]**  **# convert bitstring to a string of chars**  **chars = ''.join([str(s) for s in substring])**  **# convert string to integer**  **integer = int(chars, 2)**  **# scale integer to desired range**  **value = bounds[i][0] + (integer/largest) \* (bounds[i][1] - bounds[i][0])**  **# store**  **decoded.append(value)**  **return decoded** |

We can then call this at the beginning of the algorithm loop to decode the population, then evaluate the decoded version of the population.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **...**  **# decode population**  **decoded = [decode(bounds, n\_bits, p) for p in pop]**  **# evaluate all candidates in the population**  **scores = [objective(d) for d in decoded]** |

Tying this together, the complete example of the genetic algorithm for continuous function optimization is listed below.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78**  **79**  **80**  **81**  **82**  **83**  **84**  **85**  **86**  **87**  **88**  **89**  **90**  **91**  **92**  **93**  **94**  **95**  **96**  **97**  **98**  **99**  **100**  **101**  **102**  **103**  **104**  **105**  **106**  **107**  **108** | **# genetic algorithm search for continuous function optimization**  **from numpy.random import randint**  **from numpy.random import rand**    **# objective function**  **def objective(x):**  **return x[0]\*\*2.0 + x[1]\*\*2.0**    **# decode bitstring to numbers**  **def decode(bounds, n\_bits, bitstring):**  **decoded = list()**  **largest = 2\*\*n\_bits**  **for i in range(len(bounds)):**  **# extract the substring**  **start, end = i \* n\_bits, (i \* n\_bits)+n\_bits**  **substring = bitstring[start:end]**  **# convert bitstring to a string of chars**  **chars = ''.join([str(s) for s in substring])**  **# convert string to integer**  **integer = int(chars, 2)**  **# scale integer to desired range**  **value = bounds[i][0] + (integer/largest) \* (bounds[i][1] - bounds[i][0])**  **# store**  **decoded.append(value)**  **return decoded**    **# tournament selection**  **def selection(pop, scores, k=3):**  **# first random selection**  **selection\_ix = randint(len(pop))**  **for ix in randint(0, len(pop), k-1):**  **# check if better (e.g. perform a tournament)**  **if scores[ix] < scores[selection\_ix]:**  **selection\_ix = ix**  **return pop[selection\_ix]**    **# crossover two parents to create two children**  **def crossover(p1, p2, r\_cross):**  **# children are copies of parents by default**  **c1, c2 = p1.copy(), p2.copy()**  **# check for recombination**  **if rand() < r\_cross:**  **# select crossover point that is not on the end of the string**  **pt = randint(1, len(p1)-2)**  **# perform crossover**  **c1 = p1[:pt] + p2[pt:]**  **c2 = p2[:pt] + p1[pt:]**  **return [c1, c2]**    **# mutation operator**  **def mutation(bitstring, r\_mut):**  **for i in range(len(bitstring)):**  **# check for a mutation**  **if rand() < r\_mut:**  **# flip the bit**  **bitstring[i] = 1 - bitstring[i]**    **# genetic algorithm**  **def genetic\_algorithm(objective, bounds, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut):**  **# initial population of random bitstring**  **pop = [randint(0, 2, n\_bits\*len(bounds)).tolist() for \_ in range(n\_pop)]**  **# keep track of best solution**  **best, best\_eval = 0, objective(pop[0])**  **# enumerate generations**  **for gen in range(n\_iter):**  **# decode population**  **decoded = [decode(bounds, n\_bits, p) for p in pop]**  **# evaluate all candidates in the population**  **scores = [objective(d) for d in decoded]**  **# check for new best solution**  **for i in range(n\_pop):**  **if scores[i] < best\_eval:**  **best, best\_eval = pop[i], scores[i]**  **print(">%d, new best f(%s) = %f" % (gen,  decoded[i], scores[i]))**  **# select parents**  **selected = [selection(pop, scores) for \_ in range(n\_pop)]**  **# create the next generation**  **children = list()**  **for i in range(0, n\_pop, 2):**  **# get selected parents in pairs**  **p1, p2 = selected[i], selected[i+1]**  **# crossover and mutation**  **for c in crossover(p1, p2, r\_cross):**  **# mutation**  **mutation(c, r\_mut)**  **# store for next generation**  **children.append(c)**  **# replace population**  **pop = children**  **return [best, best\_eval]**    **# define range for input**  **bounds = [[-5.0, 5.0], [-5.0, 5.0]]**  **# define the total iterations**  **n\_iter = 100**  **# bits per variable**  **n\_bits = 16**  **# define the population size**  **n\_pop = 100**  **# crossover rate**  **r\_cross = 0.9**  **# mutation rate**  **r\_mut = 1.0 / (float(n\_bits) \* len(bounds))**  **# perform the genetic algorithm search**  **best, score = genetic\_algorithm(objective, bounds, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut)**  **print('Done!')**  **decoded = decode(bounds, n\_bits, best)**  **print('f(%s) = %f' % (decoded, score))** |

Running the example reports the best decoded results along the way and the best decoded solution at the end of the run.

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that the algorithm discovers an input very close to f(0.0, 0.0) = 0.0.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35** | **>0, new best f([-0.785064697265625, -0.807647705078125]) = 1.268621**  **>0, new best f([0.385894775390625, 0.342864990234375]) = 0.266471**  **>1, new best f([-0.342559814453125, -0.1068115234375]) = 0.128756**  **>2, new best f([-0.038909912109375, 0.30242919921875]) = 0.092977**  **>2, new best f([0.145721435546875, 0.1849365234375]) = 0.055436**  **>3, new best f([0.14404296875, -0.029754638671875]) = 0.021634**  **>5, new best f([0.066680908203125, 0.096435546875]) = 0.013746**  **>5, new best f([-0.036468505859375, -0.10711669921875]) = 0.012804**  **>6, new best f([-0.038909912109375, -0.099639892578125]) = 0.011442**  **>7, new best f([-0.033111572265625, 0.09674072265625]) = 0.010455**  **>7, new best f([-0.036468505859375, 0.05584716796875]) = 0.004449**  **>10, new best f([0.058746337890625, 0.008087158203125]) = 0.003517**  **>10, new best f([-0.031585693359375, 0.008087158203125]) = 0.001063**  **>12, new best f([0.022125244140625, 0.008087158203125]) = 0.000555**  **>13, new best f([0.022125244140625, 0.00701904296875]) = 0.000539**  **>13, new best f([-0.013885498046875, 0.008087158203125]) = 0.000258**  **>16, new best f([-0.011444091796875, 0.00518798828125]) = 0.000158**  **>17, new best f([-0.0115966796875, 0.00091552734375]) = 0.000135**  **>17, new best f([-0.004730224609375, 0.00335693359375]) = 0.000034**  **>20, new best f([-0.004425048828125, 0.00274658203125]) = 0.000027**  **>21, new best f([-0.002288818359375, 0.00091552734375]) = 0.000006**  **>22, new best f([-0.001983642578125, 0.00091552734375]) = 0.000005**  **>22, new best f([-0.001983642578125, 0.0006103515625]) = 0.000004**  **>24, new best f([-0.001373291015625, 0.001068115234375]) = 0.000003**  **>25, new best f([-0.001373291015625, 0.00091552734375]) = 0.000003**  **>26, new best f([-0.001373291015625, 0.0006103515625]) = 0.000002**  **>27, new best f([-0.001068115234375, 0.0006103515625]) = 0.000002**  **>29, new best f([-0.000152587890625, 0.00091552734375]) = 0.000001**  **>33, new best f([-0.0006103515625, 0.0]) = 0.000000**  **>34, new best f([-0.000152587890625, 0.00030517578125]) = 0.000000**  **>43, new best f([-0.00030517578125, 0.0]) = 0.000000**  **>60, new best f([-0.000152587890625, 0.000152587890625]) = 0.000000**  **>65, new best f([-0.000152587890625, 0.0]) = 0.000000**  **Done!**  **f([-0.000152587890625, 0.0]) = 0.000000** |

## Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### Books

* [Genetic Algorithms in Search, Optimization, and Machine Learning](https://amzn.to/3jADHgZ), 1989.
* [An Introduction to Genetic Algorithms](https://amzn.to/3kK8Osd), 1998.
* [Algorithms for Optimization](https://amzn.to/2Traqek), 2019.
* [Essentials of Metaheuristics](https://amzn.to/2HxZVn4), 2011.
* [Computational Intelligence: An Introduction](https://amzn.to/2HzjbjV), 2007.

### API

* [numpy.random.randint API](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html).

### Articles

* [Genetic algorithm, Wikipedia](https://en.wikipedia.org/wiki/Genetic_algorithm).
* [Genetic algorithms, Scholarpedia](http://www.scholarpedia.org/article/Genetic_algorithms).