Genetic Algorithm

The 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, 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, 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, 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, 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, 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:

```
...
# 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*".

```
...
# 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.

```
...
# 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.

```
# 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

```
...
# 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.

```
# 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.

```
# 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.

```
...
# 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

```
# 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.

```
# 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.

```
...
# 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.

```
...
# 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.

```
# 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])
```

```
46  # enumerate generations
47  for gen in range(n_iter):
48  # evaluate all candidates in the population
49  scores = [objective(c) for c in pop]
50  # check for new best solution
51  for i in range(n_pop):
52  if scores[i] < best_eval:
53  best, best_eval = pop[i], scores[i]
54  print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
55  # select parents
56  selected = [selection(pop, scores) for _ in range(n_pop)]
57  # create the next generation
58  children = list()
59  for i in range(0, n_pop, 2):
60  # get selected parents in pairs
61  p1, p2 = selected[i], selected[i+1]
62  # crossover and mutation
63  for c in crossover(p1, p2, r_cross):
64  # mutation
65  mutation(c, r_mut)
66  # store for next generation
67  children.append(c)
68  # replace population
69  pop = children
70  return [best, best_eval]
71
72 # define the total iterations
73 n_iter = 100
74 # bits
75 n_bits = 20
76 # define the population size
77 n_pop = 100
78 # crossover rate
79 r_cross = 0.9
80 # 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 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.

```
>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, 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, 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]) = -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]) = -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]) = -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.

```
# 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.

```
...
# define range for input
```

```
3 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 # bits per variable
3 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 # mutation rate
3 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 # initial population of random bitstring
3 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  # decode bitstring to numbers
2  def decode(bounds, n_bits, bitstring):
3   decoded = list()
4   largest = 2**n_bits
5   for i in range(len(bounds)):
6   # extract the substring
7   start, end = i * n_bits, (i * n_bits)+n_bits
8   substring = bitstring[start:end]
9   # 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.

```
...
# 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.

```
# 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
```

```
18   chars = ''.join([str(s) for s in substring])
19   # convert string to integer
20   integer = int(chars, 2)
21   # scale integer to desired range
22   value = bounds[i][0] + (integer/largest) * (bounds[i][1] - bounds[i][0])
23   # store
24   decoded.append(value)
25   return decoded
26
27  # tournament selection
28  def selection(pop, scores, k=3):
29   # first random selection
30   selection_ix = randint(len(pop))
31   for ix in randint(0, len(pop), k-1):
32   # check if better (e.g. perform a tournament)
33   if scores[ix] < scores[selection_ix]:
34   selection_ix = ix
35   return pop[selection_ix]
36
37  # crossover two parents to create two children
38  def crossover(p1, p2, r_cross):
39   # children are copies of parents by default
40   c1, c2 = p1.copy(), p2.copy()
41   # check for recombination
42   if rand() < r_cross:
43   # select crossover point that is not on the end of the string
44   pt = randint(1, len(p1)-2)
45   # perform crossover
46   c1 = p1[:pt] + p2[pt:]
47   c2 = p2[:pt] + p1[pt:]
48   return [c1, c2]
49
50  # mutation operator
51  def mutation(bitstring, r_mut):
52   for i in range(len(bitstring)):
```

```
53  # check for a mutation
54  if rand() < r_mut:
55  # flip the bit
56  bitstring[i] = 1 - bitstring[i]
57
58 # genetic algorithm
59 def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross, r_mut):
60  # initial population of random bitstring
61  pop = [randint(0, 2, n_bits*len(bounds)).tolist() for _ in range(n_pop)]
62  # keep track of best solution
63  best, best_eval = 0, objective(pop[0])
64  # enumerate generations
65  for gen in range(n_iter):
66  # decode population
67  decoded = [decode(bounds, n_bits, p) for p in pop]
68  # evaluate all candidates in the population
69  scores = [objective(d) for d in decoded]
70  # check for new best solution
71  for i in range(n_pop):
72  if scores[i] < best_eval:
73  best, best_eval = pop[i], scores[i]
74  print(">%d, new best f(%s) = %f" % (gen,  decoded[i], scores[i]))
75  # select parents
76  selected = [selection(pop, scores) for _ in range(n_pop)]
77  # create the next generation
78  children = list()
79  for i in range(0, n_pop, 2):
80  # get selected parents in pairs
81  p1, p2 = selected[i], selected[i+1]
82  # crossover and mutation
83  for c in crossover(p1, p2, r_cross):
84  # mutation
85  mutation(c, r_mut)
86  # store for next generation
87  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 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.

```
>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
```

```
 7 >5, new best f([0.066680908203125, 0.096435546875]) = 0.013746
 8 >5, new best f([-0.036468505859375, -0.10711669921875]) = 0.012804
 9 >6, new best f([-0.038909912109375, -0.099639892578125]) = 0.011442
10 >7, new best f([-0.033111572265625, 0.09674072265625]) = 0.010455
11 >7, new best f([-0.036468505859375, 0.05584716796875]) = 0.004449
12 >10, new best f([0.058746337890625, 0.008087158203125]) = 0.003517
13 >10, new best f([-0.031585693359375, 0.008087158203125]) = 0.001063
14 >12, new best f([0.022125244140625, 0.008087158203125]) = 0.000555
15 >13, new best f([0.022125244140625, 0.00701904296875]) = 0.000539
16 >13, new best f([-0.013885498046875, 0.008087158203125]) = 0.000258
17 >16, new best f([-0.011444091796875, 0.00518798828125]) = 0.000158
18 >17, new best f([-0.0115966796875, 0.00091552734375]) = 0.000135
19 >17, new best f([-0.004730224609375, 0.00335693359375]) = 0.000034
20 >20, new best f([-0.004425048828125, 0.00274658203125]) = 0.000027
21 >21, new best f([-0.002288818359375, 0.00091552734375]) = 0.000006
22 >22, new best f([-0.001983642578125, 0.00091552734375]) = 0.000005
23 >22, new best f([-0.001983642578125, 0.0006103515625]) = 0.000004
24 >24, new best f([-0.001373291015625, 0.001068115234375]) = 0.000003
25 >25, new best f([-0.001373291015625, 0.00091552734375]) = 0.000003
26 >26, new best f([-0.001373291015625, 0.0006103515625]) = 0.000002
27 >27, new best f([-0.001068115234375, 0.0006103515625]) = 0.000002
28 >29, new best f([-0.000152587890625, 0.00091552734375]) = 0.000001
29 >33, new best f([-0.0006103515625, 0.0]) = 0.000000
30 >34, new best f([-0.000152587890625, 0.00030517578125]) = 0.000000
31 >43, new best f([-0.00030517578125, 0.0]) = 0.000000
32 >60, new best f([-0.000152587890625, 0.000152587890625]) = 0.000000
33 >65, new best f([-0.000152587890625, 0.0]) = 0.000000
34 Done!
35 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, 1989.
- An Introduction to Genetic Algorithms, 1998.
- Algorithms for Optimization, 2019.

- Essentials of Metaheuristics, 2011.
- Computational Intelligence: An Introduction, 2007.

API

- numpy.random.randint API.

Articles

- Genetic algorithm, Wikipedia.
- Genetic algorithms, Scholarpedia.