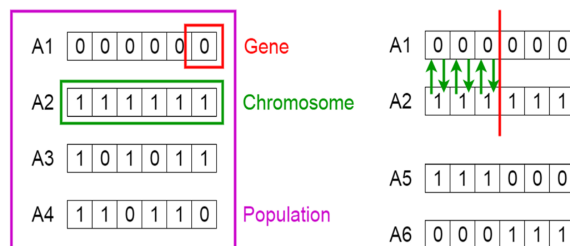
Genetic Algorithms

Soham Gandhi

March 27, 2019

1 Introduction

Genetic algorithms are based off of Charles Darwin's Theory of Evolution. In a genetic algorithm, there are four main parts: fitness, selection, crossover, and mutation.



2 Fitness

The first part is the fitness function. This determines how close the object is from the target. If an object has a high fitness then it returns a higher value and if the object has a lower fitness it returns a lower value. This ensures the fittest will have offsprings while the others will be removed. For example with neural networks the functions that has the best loss values are chosen and then are used to create off-springs.

3 Selection

The selection process is quite simple and this is where you choose the best objects that you would like to move on with and the ones you would like to drop. Normally the ones that are chosen are the ones with the highest fitness score and the lower ones are dropped.

4 Crossover

The crossover process varies depending on exactly the application. For example if we were trying to see how many iterations it will take for the algorithm to generate a certain word we would do every combination of a string. If we are trying to get the best neural network then we may have every single possible combination of features for the input. This is a very customized method for the specific application that the genetic algorithm would be used for.

5 Mutation

This is the last step. This part insures that in no point in time our code becomes obsolete or stops learning. In this method you usually select around ten percent of the algorithms and then make some mutation to it. For neural nets this maybe simply an addition or a deletion of a feature. However, for other applications you could swap characters or remove characters. The sole purpose of this method is to make sure that the code does not at any point in time become stagnant.

6 Example

This example is using randomly generated words and is trying to find the target word.

```
import random
import string
import itertools
from difflib import SequenceMatcher
```

These are simply all the basic imports that are already a part of the python library.

```
target = "hsasd".upper()
```

This is the word we are trying to find. The word is in upper case so that the process is quicker but it is also possible to do it without it being uppercase and it could even include special characters.

```
class Word:
    def __init__(self, word):
        self.word = word
    def fitLevel(self, fitness=0.0):
        self.fitness = fitness
    def getFit(self):
        return self.fitness
    def getWord(self):
        return self.word
    def setWord(self, word):
        self.word = word
```

This is a small class that I created to help out with the fitness process. It is not required but helps out a lot when it comes to organization.

```python
def populate(x=[]):
    for i in range(5):
        p1 =
            Word(word=(''.join([random.choice(string.ascii_letters)
            for n in range(len(target))])).upper())
        x.append(p1)
    return x
```

This method creates randomly 5 words which are used to start off with.

```python
def fitness(x=[]):
    z = [0.0, 0.0]
    maxrate = 0.0
    smaxrate = 0.0
    for i in x:
        d= SequenceMatcher(None, target, i.getWord()).ratio()
        i.fitLevel(d)
    for i in x:
        if(i.getFit() > smaxrate):
            if(i.getFit() > maxrate):
                smaxrate = maxrate
                maxrate = i.getFit()
                z[1] = z[0]
                z[0] = i
            else:
                smaxrate = i.getFit()
                z[1] = i
    return z
```

This method calculates how much the word matches with the target word and selects the two words that match up with the target word the most.

```python
def offsprings(x=[]):
    f = list(x[1].getWord())
    x = list(x[0].getWord())
    for i in x:
        f.append(i)
    final = list(itertools.permutations(f))
    final2 = []
    for i in final:
        p = i[0] + i[1] + i[2] + i[3] + i[4]
        final2.append(p)
        p = i[5] + i[6] + i[7] + i[8] + i[9]
        final2.append(p)
    return final2
```

This method creates all of the offsprings. This is not the most efficient method for memory or CPU since it creates a lot of possibilities. However, it is easy to understand and does the job for this case.

```
def mutate(x=[]):
    for i in range(int(len(x)/100)):
        d = random.randint(0, len(x))
        p = list(x[d])
        p[random.randint(0, len(p)-1)] = "" +
            random.choice(string.ascii_letters).upper()
        x[d] = ''.join(p)
    return x
```

This method mutates one percent of all the words. It randomly chooses a word and then an index and finally a character. This ensures, as mentioned previously, that the code does not become obsolete.

```
x = populate()
state = False
right = ""
while(state == False):
    x = offsprings(x=x)
    x = mutate(x=x)
    for i in x:
        if(i == target):
            state = True
            right = i
            break
    d = []
    for i in x:
        d.append(Word(i))
    x = d
print(right)
```

Finally this is the actual code that runs all of the methods above until the target word is achieved. Depending on the length, number of characters, and other features this could take minutes, hours or even days for something like a neural net. However, genetic algorithms offer many benefits such as being able to figure out the best neural net for a certain problem.