genetic-algo

April 4, 2020

In [1]:
```
## Writing a code for a genetic algorithm
from random import random as rnd
import numpy as np
from numpy.random import randint
import matplotlib.pyplot as plt
import seaborn as sns

import client
```

In [2]:
```
## Global Constants
SECRET_KEY = 'S7CoJ8wqziPbpykVW1SUhhW4bs9UtFCOTSvbSsvD2RRrKE60vH'
ITERATIONS = 20
POPULATION_SIZE = 20

## The values to submit
BEST_WEIGHTS = []
BEST_ERROR = 10**10

# Weights around which the genes are generated
OVERFIT_WEIGHTS = np.array([0.0,0.1240317450077846,-6.211941063144333,0.04933903144709
                            0.03810848157715883,8.132366097133624e-05,-6.0187691609169
                            -1.251585565299179e-07,3.484096383229681e-08,4.16149249934
                            -6.732420176902565e-12])
```

0.1 Ideology of the Algorithm

The basic ideology of the genetic algorithm is the idea of survival of the fittest.

This algorithm creates a group of genes which are given a rating metric known as fitness score around which their "fitness" or suitability is decided.

The genes which perform poor in this metric are thrown awat and a new generation of the genes is created by mating the "fittest" of the previous generation and are competed with

This continues till the fitness is minimized. ($\implies$ The fittest the gene can achieve) This is the naive version of the algorithm.

There is a major flaw with the algo that it can be stuck in a loop where the fittest is not achieved but the children are not fitter than the parents. To overcome this flaw, a minor variation is randomly added to the gene, thus "mutating" it, thus not allowing the gene to be stck in a local minima.

```
In [3]: class Gene:
            '''
            This is a variation of Genetic algorithm.
            '''
            def __init__(self, number_of_genes: int = 11,
                         default: list = OVERFIT_WEIGHTS):
                '''
                Creating a gene for the population.
                '''
                self.fitness = float(0)
                self.genes = np.multiply(np.random.normal(loc=1.0, scale=0.0001,
                                                          size=(number_of_genes)), default)
                np.clip(self.genes, -10, 10)

            def birth(self, parent1 = None, parent2 = None):
                '''
                Creating a new generation of the genes by mating the parents.
                '''
                if parent1 is not None and parent2 is not None:
                    for i in range(len(self.genes)):
                        self.genes[i] = np.random.choice([parent2.genes[i], parent1.genes[i],
                                                          parent1.genes[i] + parent2.genes[i]]
                                                         p = [0.45, 0.45, 0.10])
                self.mutation()
                self.genes = np.clip(self.genes, -10, 10)
                self.update_fitness()
                return self

            def update_fitness(self, real_data: bool = False,fn=lambda train,
                               val: -(train + val + 10 * abs(val - train))) -> float:
                '''
                Updating the fitness metric for every generation.
                '''
                global BEST_ERROR
                global BEST_WEIGHTS

                train_error, validation_error = client.get_errors(SECRET_KEY,
                                                                  list(self.genes))

                # Getting the best error
                if BEST_ERROR > train_error:
                    BEST_ERROR = train_error
                    BEST_WEIGHTS = self.genes
                print(self.genes, train_error, validation_error)
                self.fitness = fn(train=train_error, val=validation_error)
                return self.fitness

            def mutation(self, muatation_probability: float = 0.1,
```

```
                mutation_amount: float = 0.002):
        '''
        Mutating the child gene.
        '''
        if np.random.random() < muatation_probability:
            self.genes = np.multiply(self.genes, np.random.normal(loc = 1.0,
                                                                  scale = mutation_amou

    def __lt__(self, other):
        '''
        A comparator method.
        '''
        return self.fitness < other.fitness

    '''
    Making methods for processes to make the code more readable
    '''

    @staticmethod
    def generate_population(number_of_individuals: int, number_of_genes: int = 11,
                            default: np.ndarray = OVERFIT_WEIGHTS) -> list:
        '''
        Generating a population from the above defined global hyperparameter
        '''
        return [Gene(number_of_genes, default).birth() for iter_x in range(number_of_i

    @staticmethod
    def selection(generation: list, population_size: int = POPULATION_SIZE) -> list:
        '''
        Selection of the fittest genes for breeding.
        '''
        generation = sorted(generation, reverse=True)
        assert all([type(person) is Gene for person in generation])
        return generation[:POPULATION_SIZE]

    @staticmethod
    def pairing(generation: list, elite_fraction: float = 0.5) -> list:
        '''
        Decision of which genes to pair for further breeding.
        '''
        assert len(generation) % 2 == 0
        elite_count = int(len(generation) * 0.5)
        elites, commoners = generation[:elite_count], generation[elite_count:]
        np.random.shuffle(elites)
        np.random.shuffle(commoners)
        generation = elites + commoners
        couples = list(zip(generation[:len(generation) // 2],
                           generation[len(generation) // 2:]))
```

```
            assert all([type(couple) is tuple and len(couple) == 2 and type(couple[0]) is G
                        and type(couple[1]) is Gene for couple in couples])
            return couples

        @staticmethod
        def mating(parents: list, n_offsprings: int = 3) -> list:
            '''
            The complete process of generating the next generation.
            '''
            offsprings = [Gene().birth(couple[0], couple[1])
                          for _ in range(n_offsprings)
                          for couple in parents]
            assert all([type(person) is Gene for person in offsprings])
            return offsprings

        @staticmethod
        def stats_fitness(generation: list) -> tuple:
            '''
            Calculating the fitness.
            '''
            assert all([type(person) is Gene for person in generation])
            val_avg, val_max = 0.0, -1e100
            for person in generation:
                val_avg += person.fitness
                val_max = max(val_max, person.fitness)
            return (val_avg, val_max)

In [4]: if __name__ == "__main__":
        CHOICE = input(("Do you wish to get errors from the server (Y/n): "))

        # As the amount of requests is limited
        if CHOICE in ('n', 'N'):
            exit(0)

        generation = Gene.generate_population(POPULATION_SIZE)
        assert all([type(person) is Gene for person in generation])

        for i in range(ITERATIONS):
            print('**************************')
            couples = Gene.pairing(generation)
            children = Gene.mating(couples)
            generation = Gene.selection(generation + children)
            assert all([type(person) is Gene for person in generation])

        # Checking if the change in hyperparameters is useful
        print("Best Soln : ")
        print(BEST_WEIGHTS,BEST_ERROR)
```

```
            # Submitting it regardless
            client.submit(SECRET_KEY,list(BEST_WEIGHTS))

Do you wish to get errors from the server (Y/n): n
```

0.2 Sample Ouput

The format is [gene] , training error, validation error.

A single generation output is :

[0.00000000e+00 1.24051937e-01 -6.21157191e+00 4.93390104e-02 3.81066382e-02 8.13198767e-05 -6.01791626e-05 -1.25163350e-07 3.48379719e-08 4.16166196e-11 -6.73266866e-12] 79068.35618656754 3625009.2445584373

[0.00000000e+00 2.48074052e-01 -6.21227586e+00 4.93420788e-02 3.81088826e-02 8.13320094e-05 -6.01862275e-05 -1.25174854e-07 3.48411764e-08 4.16121405e-11 -6.73257744e-12] 78564.99206532972 3599264.6284862114

[0.00000000e+00 1.24051937e-01 -6.21157191e+00 4.93390104e-02 3.81115608e-02 8.13252414e-05 -6.01943185e-05 -1.25163350e-07 3.48441122e-08 4.16173375e-11 -6.73207687e-12] 77405.86752465289 3576377.243394996

[0.00000000e+00 1.24030471e-01 -6.21198215e+00 4.93448793e-02 3.81132432e-02 8.13320094e-05 -6.01726078e-05 -1.25152258e-07 3.48411764e-08 4.16173375e-11 -6.73214355e-12] 73088.05703581433 3367142.0415729606

...

...

...

[0.00000000e+00 1.24458241e-01 -6.23348853e+00 4.94983337e-02 7.65372819e-02 8.17369378e-05 -6.04013979e-05 -1.25812882e-07 3.50148176e-08 4.17519209e-11 -6.75523643e-12] 9008384676.06751 12053620097.058926

[0.00000000e+00 1.24036637e-01 -6.21217479e+00 4.93374546e-02 3.81124115e-02 1.62643279e-04 -6.01883312e-05 -1.25149548e-07 3.48406653e-08 4.16120208e-11 -6.73216378e-12] 83437429.16884294 135323322.62799495

1 Report

1.1 Visualization

Visualization of the same data given above.

```
In [4]: error_train = [[79068.35618656754],[78564.99206532972],[77405.86752465289]
        ,[73088.05703581433],[147229509731.7974],[80519.83525052817],[73915.25133078403]
        ,[142256383242.4591],[890609991.3944016],[101281768.7402446],[453057827.1324145]
        ,[25692851308.77479],[84629.40273073125],[147298851099.8687],[96215184720.34409]
        ,[23005149.66896958],[50196513458.7616 ],[85062.19173518094],[29728632.80512077]
        ,[79176.61267418708],[82758.86059257714],[25696571257.8197 ],[449369508.7334888]
        ,[7496825.825865613],[77555.62112911642],[7496994.716868373],[939053980.081508 ]
        ,[74957.4214821493 ],[9008384676.06751 ],[83437429.16884294]]

        error_validation = [[3625009.2445584373],[3599264.6284862114],[3576377.243394996]
```

```
,[3367142.0415729606],[201058411392.39093],[3629559.491204448],[3357076.8277312005]
,[ 196581781875.3407],[1154115747.9858515],[ 135155141.73198014],[680814471.9594347]
,[36007848599.752846],[3703208.1549647897],[201153639893.69632],[132799081618.30923]
,[ 43338427.304116],[8115074064.41129],[3722539.721714723],[ 54420402.18735669]
,[3629199.5574274748],[3679591.516022193],[6012912933.42186],[ 662687820.973728]
,[17391322.769658454],[3571080.729525005],[17351545.80226638],[299643327.6191967]
,[527803.2506977394],[2053620097.058926],[135323322.62799495]]
```

1.2 Analysis

From the graphs below, there are a couple trends which can be seen very clear. First being the fact that there are random spikes that occur and other being that the function invariably decreases in value after such a spike.

1.2.1 Understanding the spikes

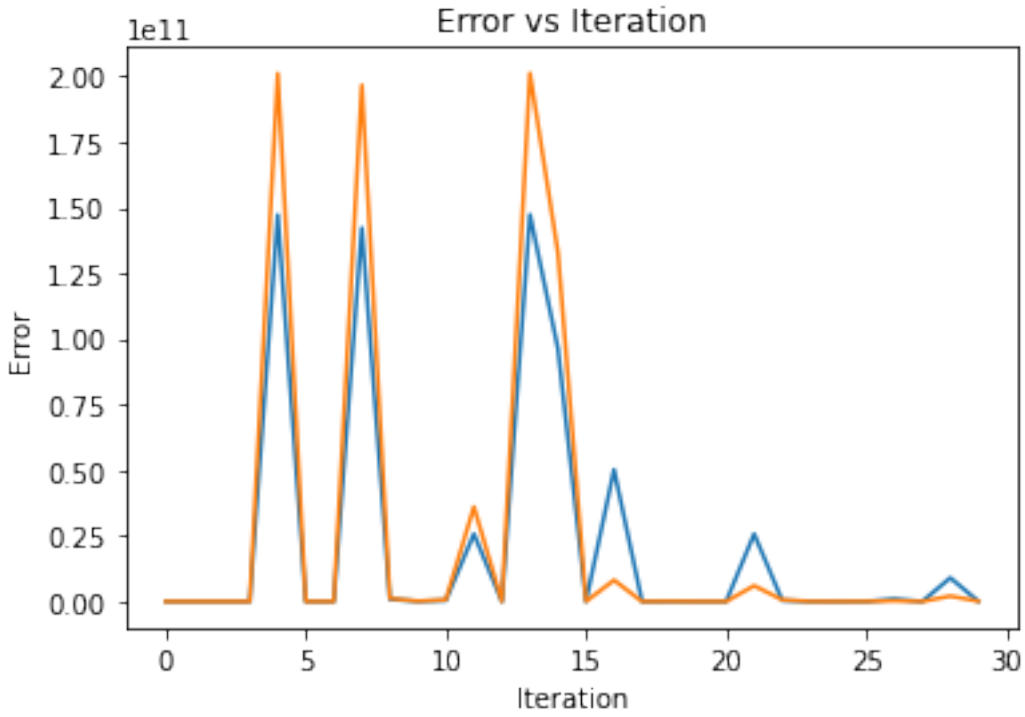
In the above explanation of the genetic algorithm, I pointed out that mutation occurs to stop the algo from ending up in a local minima and not exploring the complete space it is afforded. What mutation essentially does is raise the fitness enough that it can now access the other paths, thus making it easier for it to jump to another fitness well.

$\therefore$ The random spikes are the new population that has mutation.

1.2.2 Understanding the fall

Now, as stated above, there is a certain fall after the spike. This is due to fact that after the mutation, the new population will always improve on the previous (Atleast as an average), thus invariably bringing down the fitness.

```
In [5]: plt.plot(error_train)
        plt.plot(error_validation)
        plt.xlabel('Iteration')
        plt.ylabel('Error')
        plt.title('Error vs Iteration')
        plt.show()
```



1.3 Choices for Hyperparameters

So, the choices of hyperparamters like the population size, the mutation probability and the rate of mutation were to be justified as they are the governing factors that dictate the general behavior of the algorithm

1.3.1 Population Size

Starting with population size. Increasing popultion size does give more options for mating, thus creating a more diverse pool of genes for next generation thus giving more opportunity for the genes to arrive at a minima.

But after a certain point, the benefits start to wane of. This is due to the fact that there are only 11 parameters, so after a point everyone is a stranger to everyone, thus not giving such benifits. The computational cost continues to still rize up and hence an optimum value exists for which the computation is not that high but the maximum benifits are taken.

For my computer I found that taking the size to be around 20 gives a decent result without sacrificing much of the uplift.

1.3.2 Probability of Mutation

This one cannot be chosen too randomly as making the mutation too probable would make the next generation basically random and would remove any inheritance from the parent genes. Similarly if the mutation probability is too low then childen will be exact derivitives of the parents, thus making the gene stuck in a local optimum and not exploring the complete space.

For this question, I found the probability of 10% to be giving decent results. I actually ran the algorithm for probabilies in the range of 5% - 15% before deciding 10% is the optimum in that range.

1.3.3 Mutation Rate

Rate $\implies$ Deviation from original value Mutation rate faces a similar problem to that of the probability of mutation, that if the rate is too high, it has a possibility that the gene will a fitness well thus not exploring it. Also if it too low, the childem will be replicas of the parents.

For this question I found the rate of 0.002 works decently.