Genetic Algo

Team Number : 81

Genetic Algorithms Report

March 14, 2021

Summary

In our genetic algorithm code, firstly the initial population (with pool size =10) is selected with it's values in the range of (-99%,+100%) of the given overfit vector. Then we apply the selection, cross-over, and fitness functions as explained below. Each time the code can be run for any number of generations. After the code has run for the pre-defined number of generations, it will dump the vectors in the last generation to the last_gen.txt file. The next time we run the code, it will automatically take it's initial population (vectors) from the last_gen.txt file.

Fitness Function

```
def cal_error(pop):
    fitness = []
    i = 1
    for p in pop:
        fitness.append(get_errors(SECRET_KEY, list(p)))
    return fitness

def cal_error_weight(fitness):
    for e in fitness:
        e[0] = TRAIN_DATA_WEIGHT*e[0] + (1-TRAIN_DATA_WEIGHT)*e[1]
    print("Fitness = ",fitness)
    return fitness
```

2

```
def cal_fitness(fitness):
    total = 0
    for e in fitness:
        total = total + e[0]
    fitness_val = []
    # for best results keep EXP between 0.5 and 1
    EXP = 0.85
    for e in fitness:
        fitness_val.append(math.pow(total/e[0],EXP))
    return fitness_val
```

First 'cal_error()' function saves the train and validation error values of all the vectors of the population in a list. Then 'cal_error_weight()' function calculates a weighted error value for each vector (where the weight is TRAIN_DATA_WEIGHT). Finally 'cal_fitness()' function, calculates the fitness values for each vector in the population. Here, we have taken, (for each vector) fitness = total/fitness, (i.e, fitness of each vector is the sum of all error values divided by it's error value). This method preserves the original ratio of error values between any two vectors. The EXP is taken, as it reduces the gaps between fitness values (each vector with low fitness value is now more likely to be chosen than before), which helps in maintaining the diversity of the population.

The selection function selects the next vectors based on these fitness values using a completely random roulette wheel concept.

Cross-Over Function

```
def crossover(parents, NUM_PARENTS_MATING):
    offspring = numpy.empty(parents.shape)
    n = offspring.shape[0]
    i = 0
    while i < n:
```

```
        num = random.sample(range(1,offspring.shape[1]),NUM_PARENTS_MATING-1)
        num.sort()
        num.append(offspring.shape[1])
        for idx in range(i, i+NUM_PARENTS_MATING):
            offspring[idx][0:num[0]] = parents[idx][0:num[0]]
            for k in range(0,len(num)-1):
                offspring[idx][num[k]:num[k+1]] =
parents[i+(idx+k+1)%NUM_PARENTS_MATING][num[k]:num[k+1]]
        i = i + NUM_PARENTS_MATING
    return offspring
```

The crossover method used here is one-point crossover which will select n-1 random points (n = number of parents) to divide the gene pool into n sets and then alternately place these gene pool sets in the offspring.

The crossover function is coded such that it allows crossover of any number of parents for an offspring. For our code, currently we have kept NUM_PARENTS_MATING = 2.

Mutation function

```
def mutation(offspring_crossover):
    for idx in range(offspring_crossover.shape[0]):
        for j in range(offspring_crossover.shape[1]):
            random_value = numpy.random.uniform(-1.0, 1.0, 1)
            if(random_value > -MUTATION_PROB and random_value <
MUTATION_PROB ):
                mut = numpy.random.uniform(-MUTATION_CHANGE,MUTATION_CHANGE)
                s = numpy.random.choice([-1,1])
                offspring_crossover[idx, j] = offspring_crossover[idx,
j]*(1+s*mut)
    return offspring_crossover
```

For applying mutation in our offspring, we have first selected our probability of mutation as 0.3, and this mutation probability is independent for each gene in a solution (vector). Then a number is selected (similar to roulette wheel) and if it falls in the given range, we apply mutation. In mutation, we increase or decrease the magnitude of the gene by a random number, which leads to a change in number between -30% to 30%.

Hyper-parameters

```
SOL_PER_POP = 10
NUM_PARENTS_MATING = 2
TRAIN_DATA_WEIGHT = 0.4
MUTATION_PROB = 0.3
MUTATION_CHANGE = 0.3
```

1. **SOL_PER_POP = 10:** This is the number of solutions (vectors) in each generation of our model. It can also be called the population size. We have taken it to be 10, as lower values of it would lead to a model which is more likely to discover a local minima but not the global minimum, whereas if the value was too big the model would take a very large number of generations to converge to the global minima. This value seemed to give optimal results in a considerable number of generations.
2. **NUM_PARNETS_MATING = 2:** This is the number of parents each offspring will have. It can take any value between 2, and SOL_PER_POP. However, for bigger values there is a high chance that the population would mix over, which would cause it to become stagnant (stable) leading to an early convergence point. Thus we chose the simplest value possible.
3. **TRAIN_DATA_WEIGHT = 0.4:** This is the weight given to train error in calculating overall error values from the train error and validation error. Higher value of this

variable would lead to the vectors having better (less) average train error as compared to average validation error and vice versa.
Train error weightage should be comparable to validation error weightage (1 - TRAIN_DATA_WEIGHT) as our aim is to reduce both the erros. However, since we want to prevent overfitting our vectors to the train dataset, the TRAIN_DATA_WEIGHT value is slightly less than 0.5.

4. **MUTATION_PROB = 0.3:** This is the probability of mutation w.r.t. a single gene. A probability value between 0.2 and 0.4 seemed optimal, as it would change around ⅓ of the genes in each vector, without which consecutive generations seemed to give similar results.

5. **MUTATION_CHANGE = 0.3:** This value denotes the maximum change (percent-wise) that can occur due to mutation at a gene.The mutation change value also seemed to be optimal between 0.25 and 0.5.
Here lower values seemed to cause insignificant changes to the population which led to similar consecutive generations, while higher values would also be worse as it would lead to 'bouncing of the population' which might cause the population to go past the global minima which is unwanted

Statistical Information

Our population started to converge from around the 300th generation, where we consistently achieved train and validation error within (2e10, 9e10) range in every generation.

Number of generations: 383

Heuristics applied

Population:

In the starting, our initial population was chosen randomly (each gene $\in$ [-10,10]). But this did not seem to converge (we tried for around 80 generations each in 2-3 iterations) as the population was too random and the population wasn't giving any good results.

To speed-up the initial convergence part a little bit we selected the initial population randomly within (-99%,100%) of the overfit vector. This speeded up the initial convergence significantly.

Crossover:

Our first approach involved using fitness values of the 2 parent vectors in crossover (**Whole Arithmetic Recombination crossover** method), so that the offspring contained more genes from the parent with better fitness value. However, this approach failed as the vectors quickly converged and after that the population showed little to no change.

Then, we tried using 5 parents in crossover, but this caused the values to get averaged over and again resulted in early convergence.

So we used single point crossover with 2 parents which seemed to give acceptable results.

Fitness:

Initially, we tried using a fitness function where all fitness values were raised to a negative power (between -3 to -5) after taking the weighted average, and using them to calculate percentages for the roulette wheel. However, as the error values were very large, the selection probabilities of bad vectors became similar to selection probabilities of good vectors (Since, $\Delta x^{-5} < \Delta x$, $\Delta x \rightarrow$selection probability difference b/w a good and bad vector), which prevented the convergence of the population and caused the population to lose it's good genes over time.

Next, we used fitness value: fitness = 1 - error/total_error , and fitness = total_error/error. Here, in the first case the error ratios were not preserved in the fitness values, and so some good genes got lost over time. While the second case preserved the error ratios in fitness values, due to large differences in initial error values (eg: 1e11,1e14 => error values of 2 vectors) of the vectors, the selection function just selected the best 2 to 3 vectors again and again, and removed the diversity of the population causing it to become stagnant.

Mutation:

The mutation probability was increased from 0.2 to 0.3 as in many iterations the population seemed to converge to the local minima. This new mutation probability helped the vectors to overcome the local minima's and converge on the global minima.

Train Error and Validation Error achieved

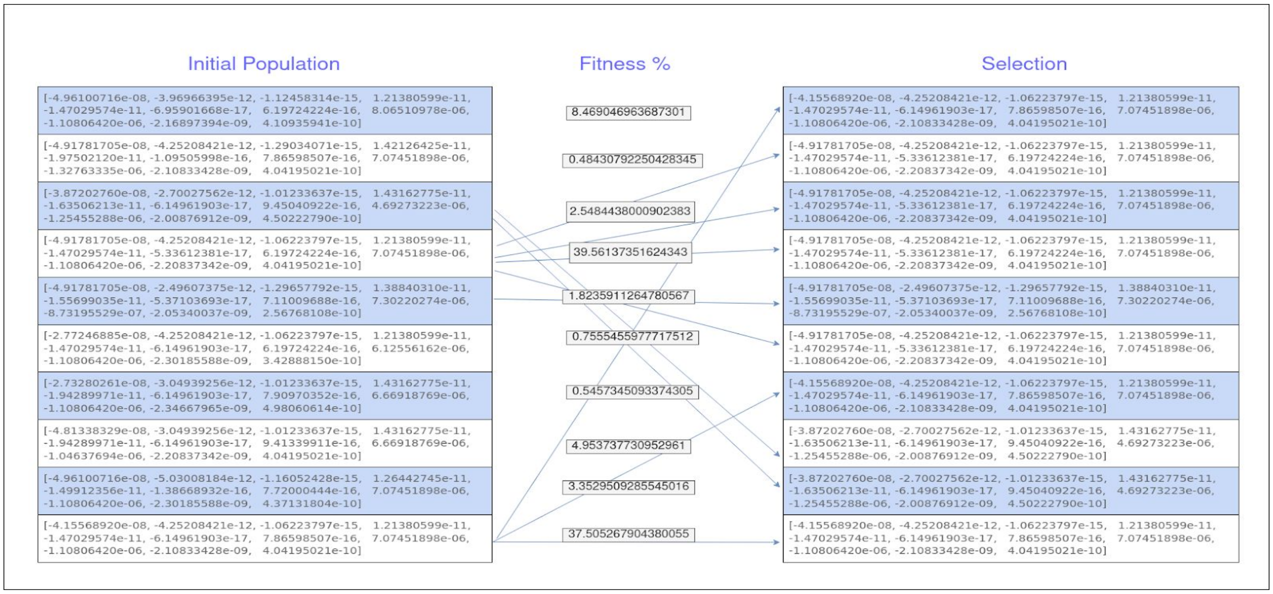
The best vector we achieved [-6.40431139e-08, -4.17710614e-12, -8.81212887e-16, 1.26877952e-11, -1.59340051e-11, -4.39386468e-17, 8.28413802e-16, 6.66918769e-06, -1.1080642e-06, -1.88846174e-09, 4.04195021e-10] had train error = 4.61e10, and validation error = 3.74e10, and this vector performed considerably well on the test data set.

Our model was trained such that it promoted vectors with low errors in the train and validation datasets. Assuming these 2 datasets are considerably different, if a vector performs well on both of these 2 datasets then it is likely that the vector has achieved (or is close to) the global minima (though it is not guaranteed, as if the train and validation sets were similar, then the vector is more likely to achieve a local minima point). Thus, theoretically this vector should be able to perform well on any unseen dataset as long as the dataset is not biased datasets..

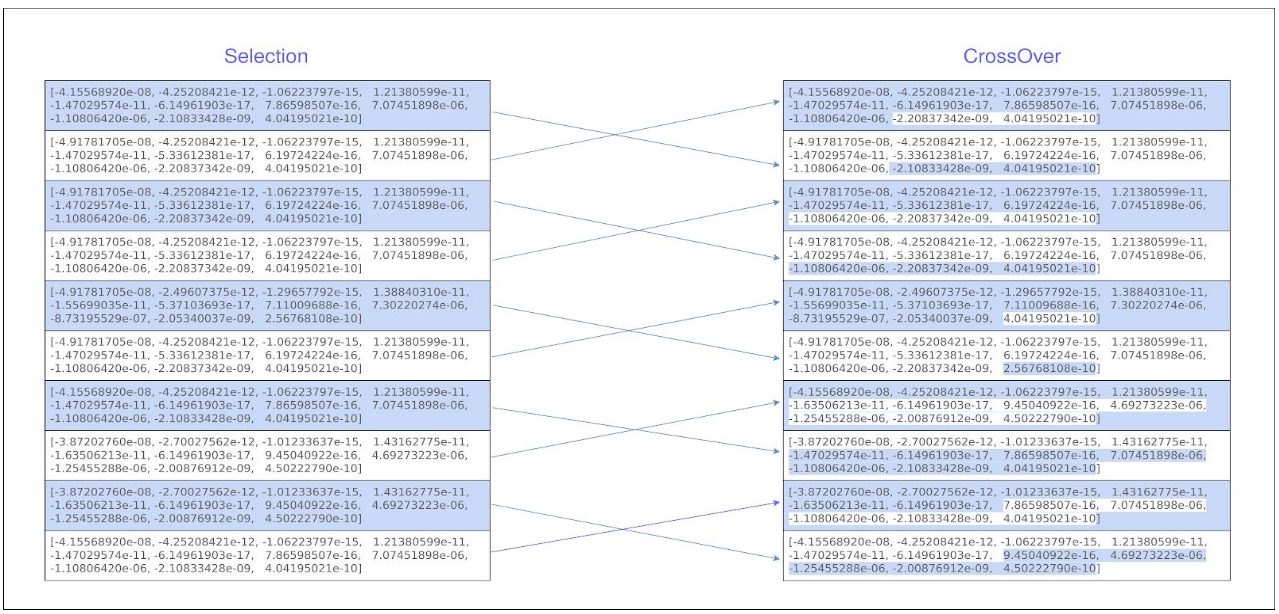
Three Iterations of the Algorithm

Gen # 1 :

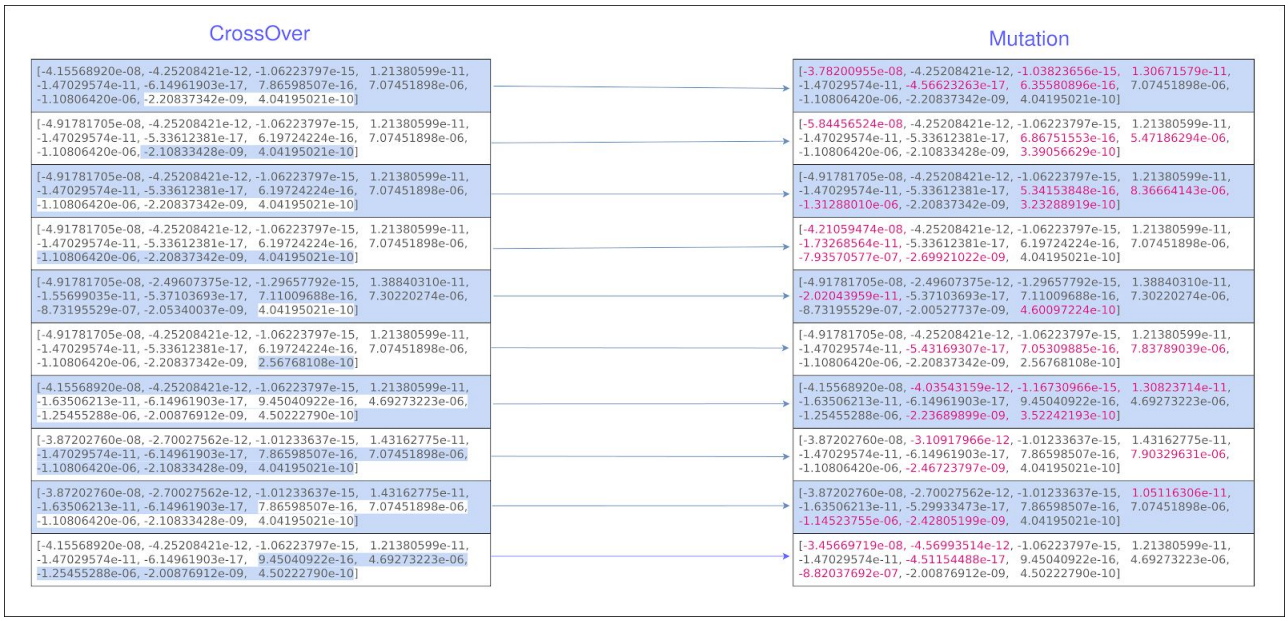
Selection :



Crossover :

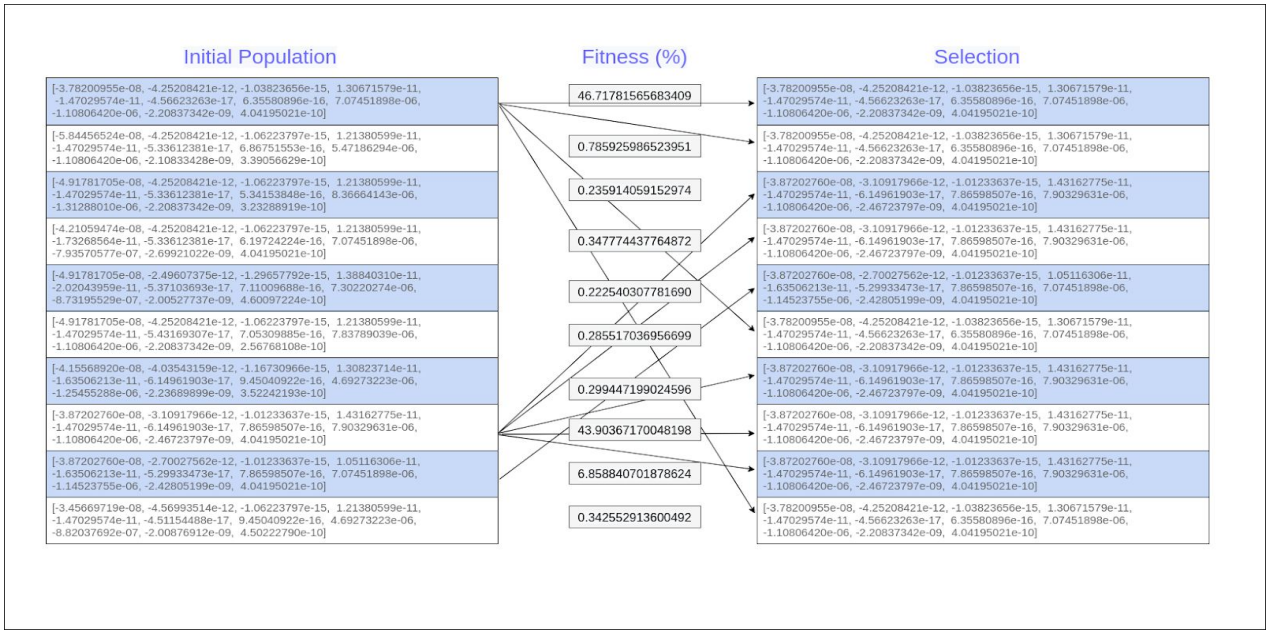


Mutation :

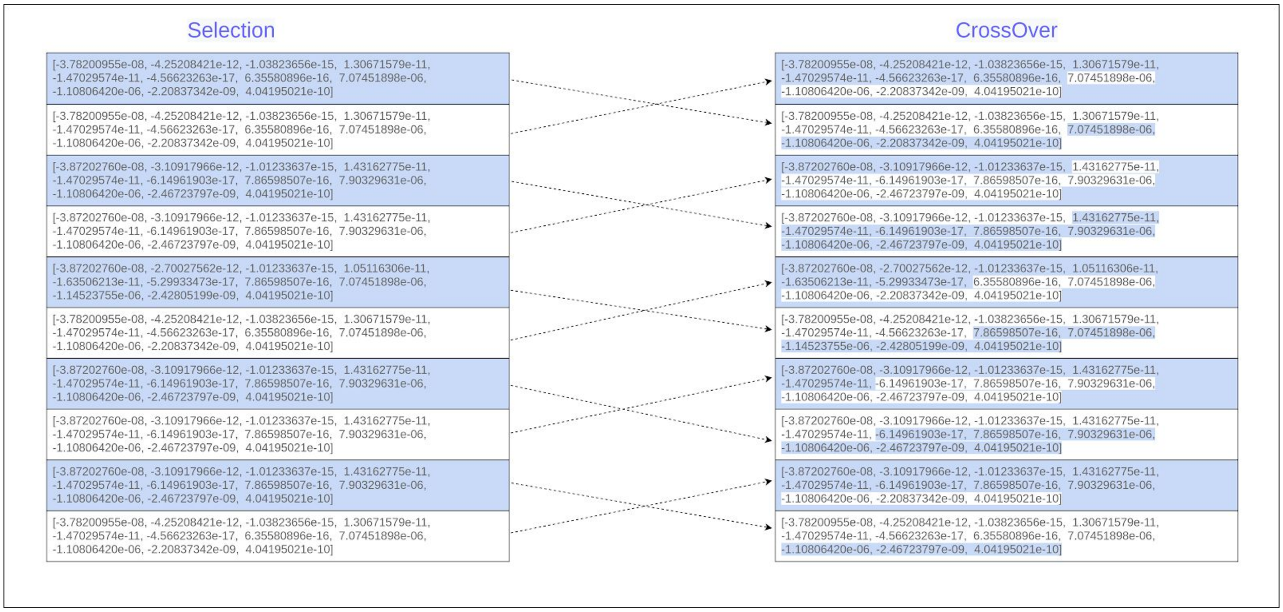


Gen # 2 :

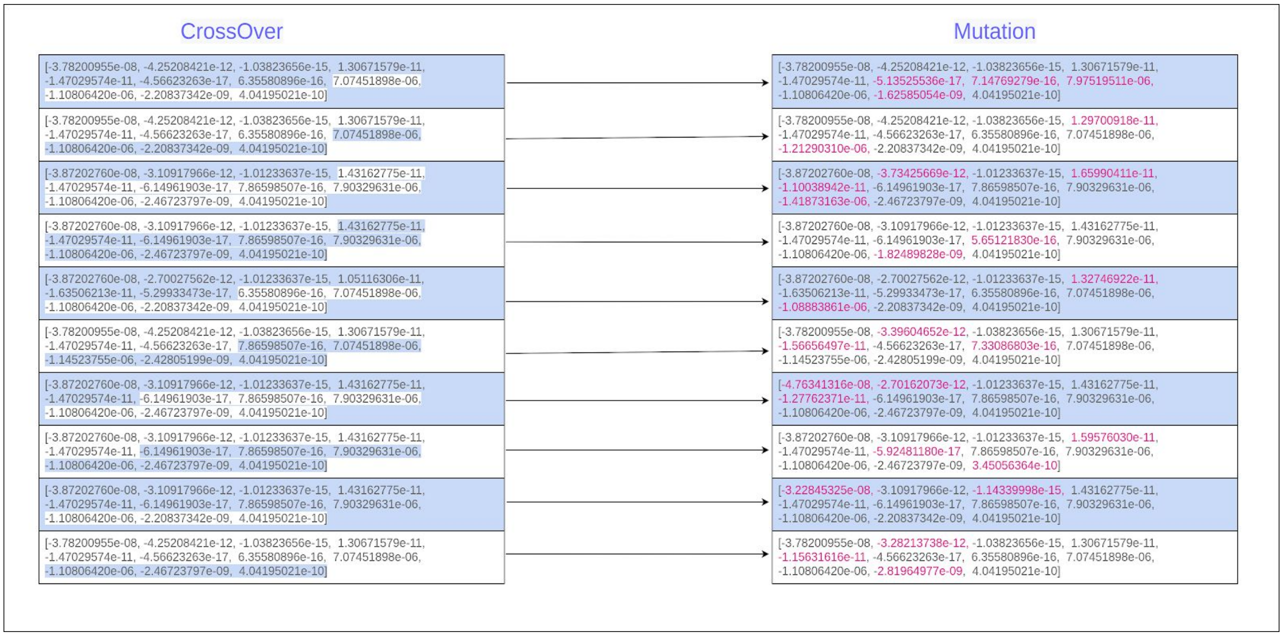
Selection :



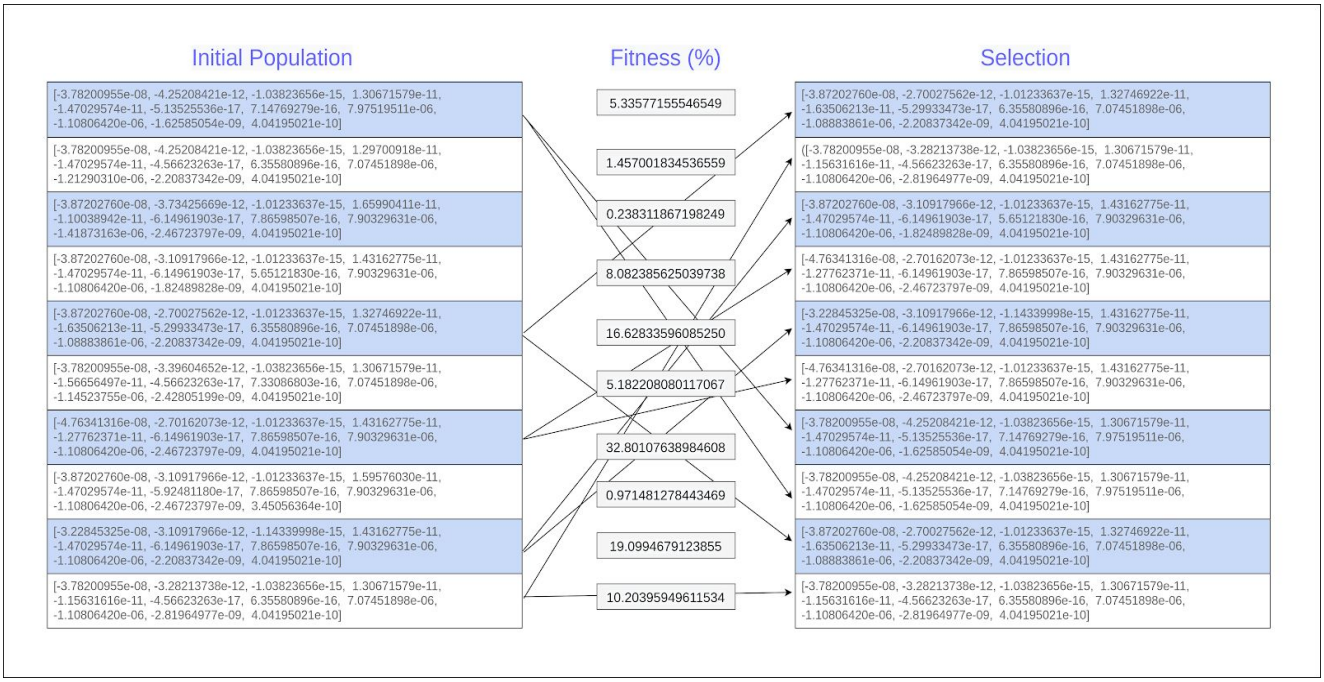
CrossOver :



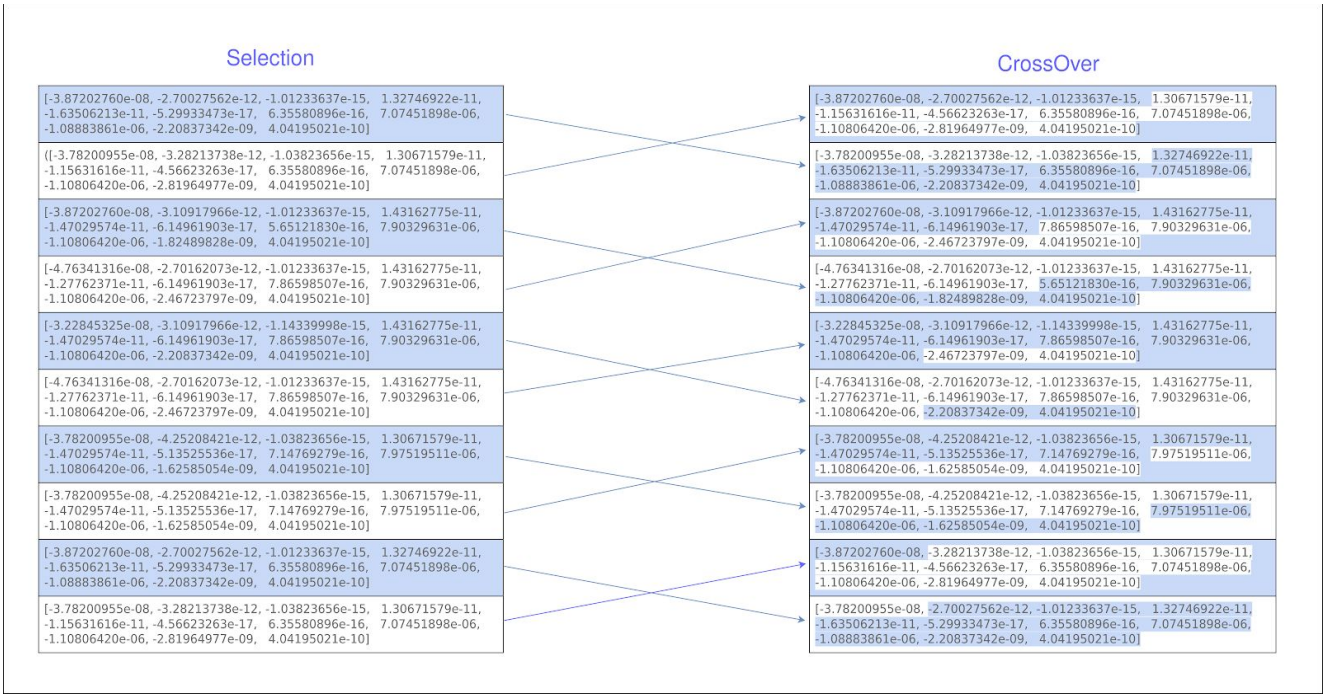
Mutation :



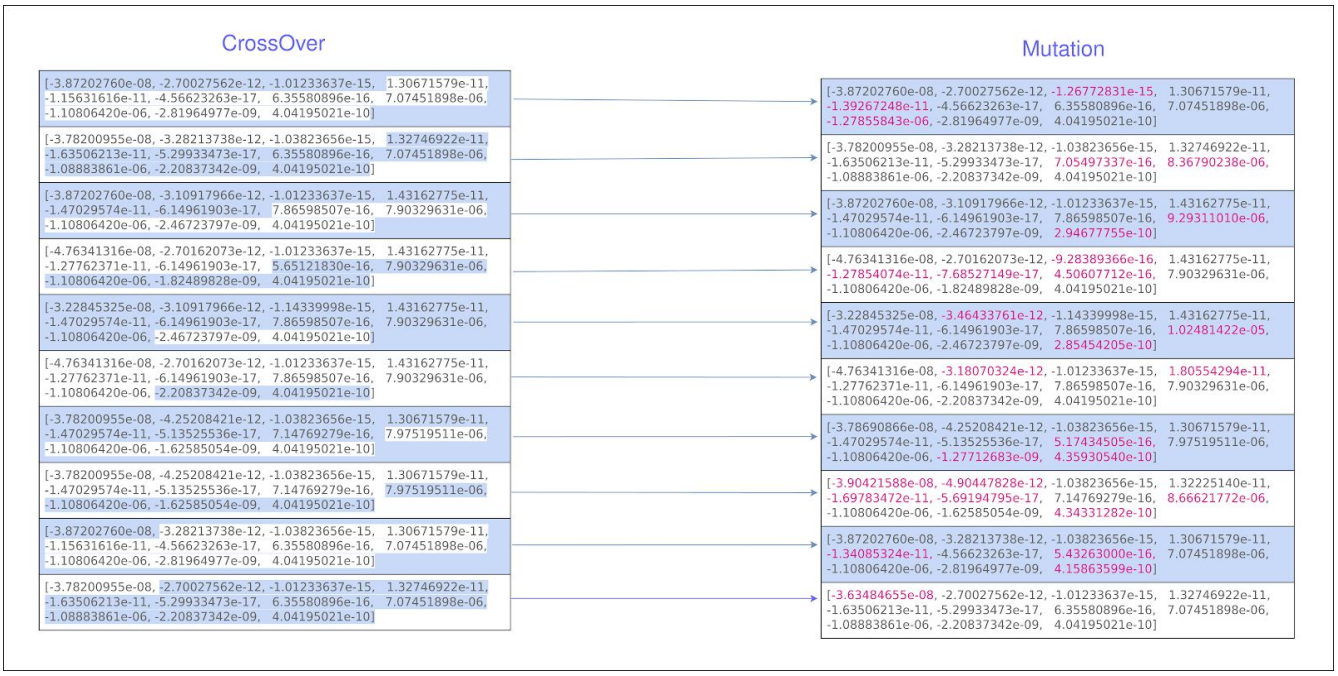
Gen # 3 :



CrossOver :



Mutation :


CrossOver
Mutation
[-3.87202760e-08, -2.70027562e-12, -1.01233637e-15, 1.30671579e-11, -1.15631616e-11, -4.56623263e-17, 6.35580896e-16, 7.07451898e-06, -1.10806420e-06, -2.81964977e-09, 4.04195021e-10]
[-3.87202760e-08, -2.70027562e-12, -1.26772831e-15, 1.30671579e-11, -1.39267248e-11, -4.56623263e-17, 6.35580896e-16, 7.07451898e-06, -1.27855843e-06, -2.81964977e-09, 4.04195021e-10]
[-3.78200955e-08, -3.28213738e-12, -1.03823656e-15, 1.32746922e-11, -1.63506213e-11, -5.29933473e-17, 6.35580896e-16, 7.07451898e-06, -1.08883861e-06, -2.20837342e-09, 4.04195021e-10]
[-3.78200955e-08, -3.28213738e-12, -1.03823656e-15, 1.32746922e-11, -1.63506213e-11, -5.29933473e-17, 7.05497337e-16, 8.36790238e-06, -1.08883861e-06, -2.20837342e-09, 4.04195021e-10]
[-3.87202760e-08, -3.10917966e-12, -1.01233637e-15, 1.43162775e-11, -1.47029574e-11, -6.14961903e-17, 7.86598507e-16, 7.90329631e-06, -1.10806420e-06, -2.46723797e-09, 4.04195021e-10]
[-3.87202760e-08, -3.10917966e-12, -1.01233637e-15, 1.43162775e-11, -1.47029574e-11, -6.14961903e-17, 7.86598507e-16, 9.29311010e-06, -1.10806420e-06, -2.46723797e-09, 2.94677755e-10]
[-4.76341316e-08, -2.70162073e-12, -1.01233637e-15, 1.43162775e-11, -1.27762371e-11, -6.14961903e-17, 5.65121830e-16, 7.90329631e-06, -1.10806420e-06, -1.82489828e-09, 4.04195021e-10]
[-4.76341316e-08, -2.70162073e-12, -9.28389366e-16, 1.43162775e-11, -1.27854074e-11, -7.68527149e-17, 4.50607712e-16, 7.90329631e-06, -1.10806420e-06, -1.82489828e-09, 4.04195021e-10]
[-3.22845325e-08, -3.10917966e-12, -1.14339998e-15, 1.43162775e-11, -1.47029574e-11, -6.14961903e-17, 7.86598507e-16, 7.90329631e-06, -1.10806420e-06, -2.46723797e-09, 4.04195021e-10]
[-3.22845325e-08, -3.46433761e-12, -1.14339998e-15, 1.43162775e-11, -1.47029574e-11, -6.14961903e-17, 7.86598507e-16, 1.02481422e-05, -1.10806420e-06, -2.46723797e-09, 2.85454205e-10]
[-4.76341316e-08, -2.70162073e-12, -1.01233637e-15, 1.43162775e-11, -1.27762371e-11, -6.14961903e-17, 7.86598507e-16, 7.90329631e-06, -1.10806420e-06, -2.20837342e-09, 4.04195021e-10]
[-4.76341316e-08, -3.18070324e-12, -1.01233637e-15, 1.80554294e-11, -1.27762371e-11, -6.14961903e-17, 7.86598507e-16, 7.90329631e-06, -1.10806420e-06, -2.20837342e-09, 4.04195021e-10]
[-3.78200955e-08, -4.25208421e-12, -1.03823656e-15, 1.30671579e-11, -1.47029574e-11, -5.13525536e-17, 7.14769279e-16, 7.97519511e-06, -1.10806420e-06, -1.62585054e-09, 4.04195021e-10]
[-3.78690866e-08, -4.25208421e-12, -1.03823656e-15, 1.30671579e-11, -1.47029574e-11, -5.13525536e-17, 5.17434505e-16, 7.97519511e-06, -1.10806420e-06, -1.27712683e-09, 4.35930540e-10]
[-3.78200955e-08, -4.25208421e-12, -1.03823656e-15, 1.30671579e-11, -1.47029574e-11, -5.13525536e-17, 7.14769279e-16, 7.97519511e-06, -1.10806420e-06, -1.62585054e-09, 4.04195021e-10]
[-3.90421588e-08, -4.90447828e-12, -1.03823656e-15, 1.32225140e-11, -1.69783472e-11, -5.69194795e-17, 7.14769279e-16, 8.66621772e-06, -1.10806420e-06, -1.62585054e-09, 4.34331282e-10]
[-3.87202760e-08, -3.28213738e-12, -1.03823656e-15, 1.30671579e-11, -1.15631616e-11, -4.56623263e-17, 6.35580896e-16, 7.07451898e-06, -1.10806420e-06, -2.81964977e-09, 4.04195021e-10]
[-3.87202760e-08, -3.28213738e-12, -1.03823656e-15, 1.30671579e-11, -1.34085324e-11, -4.56623263e-17, 5.43263000e-16, 7.07451898e-06, -1.10806420e-06, -2.81964977e-09, 4.15863599e-10]
[-3.78200955e-08, -2.70027562e-12, -1.01233637e-15, 1.32746922e-11, -1.63506213e-11, -5.29933473e-17, 6.35580896e-16, 7.07451898e-06, -1.08883861e-06, -2.20837342e-09, 4.04195021e-10]
[-3.63484655e-08, -2.70027562e-12, -1.01233637e-15, 1.32746922e-11, -1.63506213e-11, -5.29933473e-17, 6.35580896e-16, 7.07451898e-06, -1.08883861e-06, -2.20837342e-09, 4.04195021e-10]