# Report

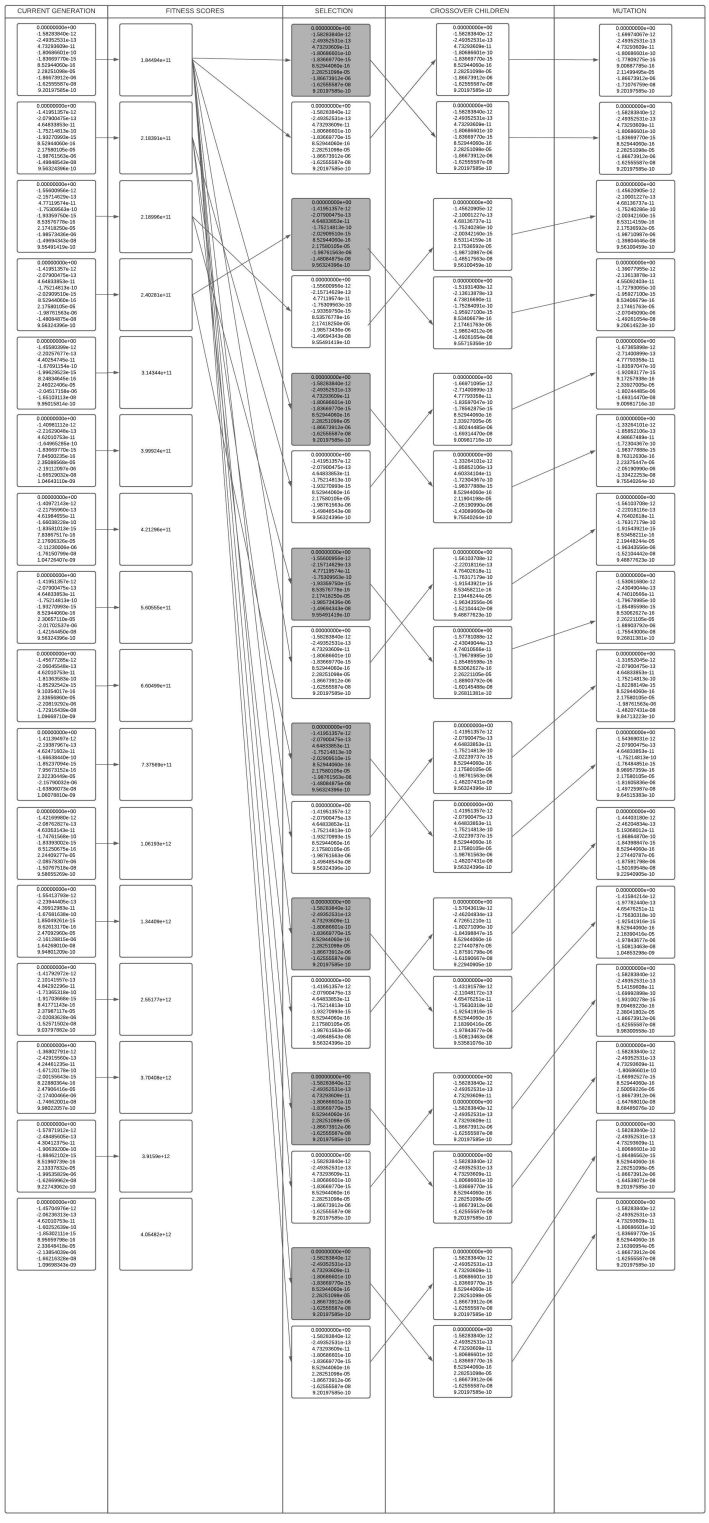
## Summary

Genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection.

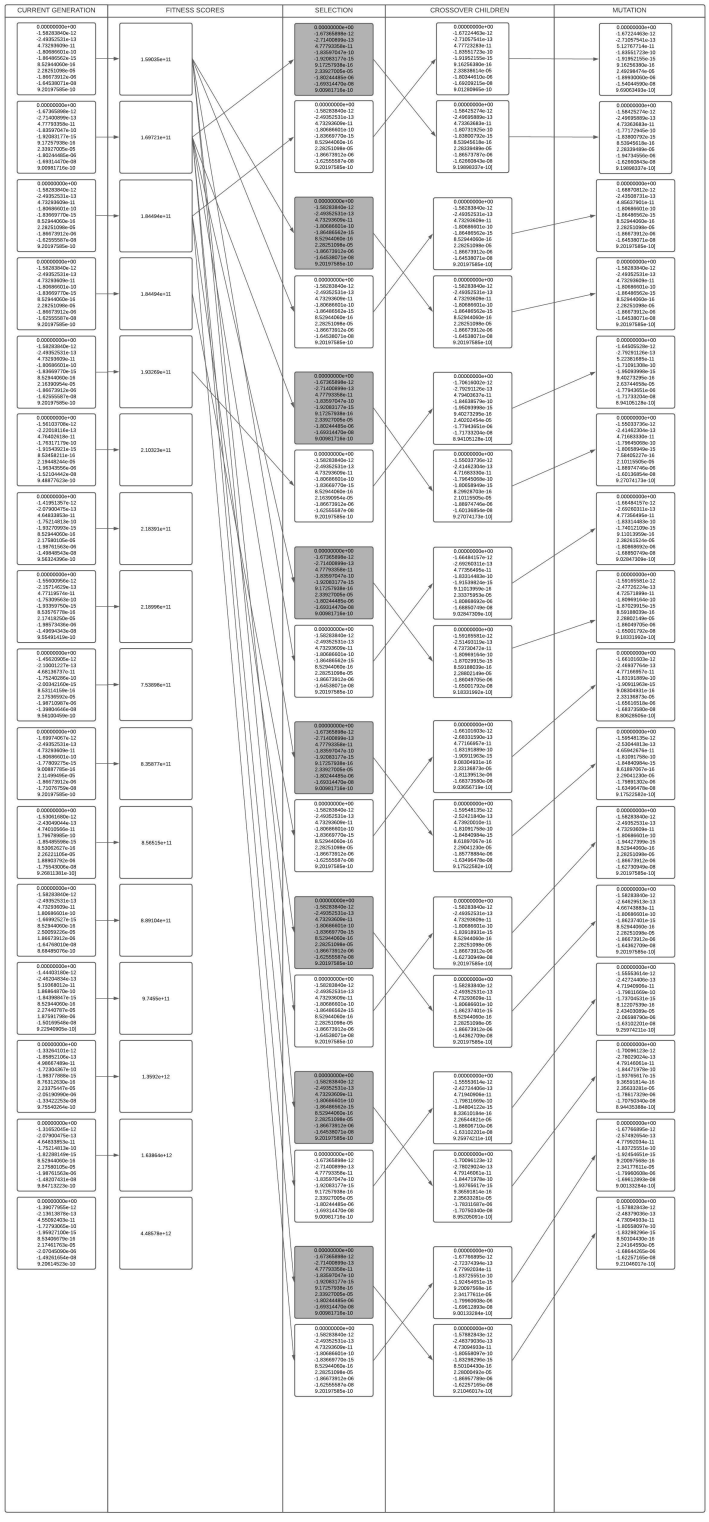
In GAs, we have a pool or a population of possible solutions to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more “fitter” individuals. This is in line with the Darwinian Theory of “Survival of the Fittest”.

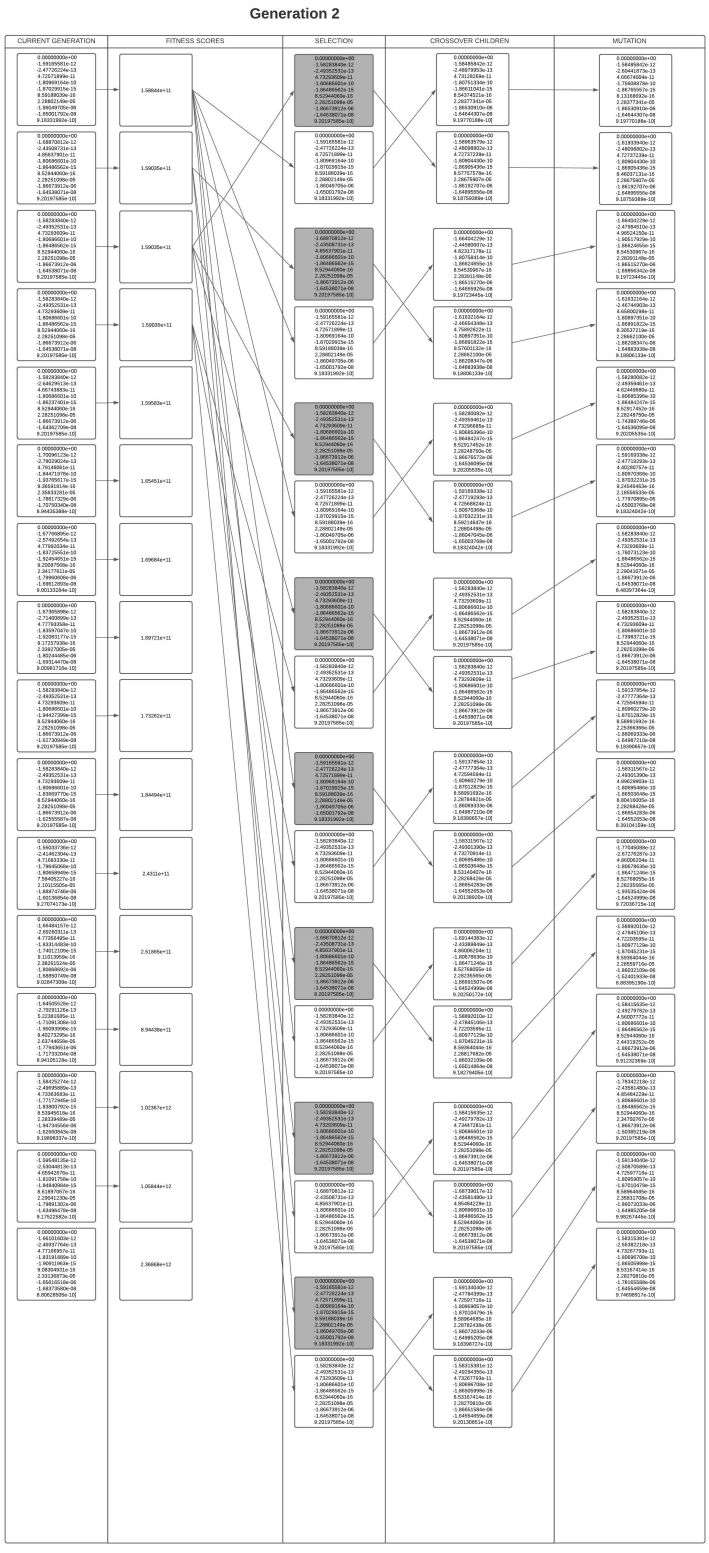
### Iteration Diagrams

## Generation 0



## Generation 1






Below is the step by step explanation of our genetic algorithm:

## Initializing Population

This step involves making the initial sets of the coefficients. These sets known are known as **chromosomes**. In our code we initialize the population by heavily **mutating** the *overfit vector* that has been provided to us and making the population from that vector. We mutate the overfit vector with a probability of 90% ***POP_SIZE(16)*** times within a range of 0.01 to get vectors close to the overfit vector as our initial generation. The code for this step is as follows:

```
def initialize_population(starting_vector):
    population = np.zeros((POP_SIZE , MAX_DEG) , dtype='float64')
    for i in range(len(population)):
        population[i] = mutate(starting_vector , 0.01 , 0.9)

    return population

def mutate(vector  ,mut_range , mut_prob):
    mutated_vect = np.copy(vector)
    for i in range(len(mutated_vect)):
        w = mutated_vect[i]
        reduction_fact = random.uniform(-mut_range,mut_range)
        reduction = w*reduction_fact
        if random.random() <= mut_prob:
            w = w + reduction
        mutated_vect[i] = w

    return mutated_vect
```

## Calculation of Fitness

Fitness function determines how fit a chromosome. The probability that an individual will be selected for reproduction is based on its fitness score. Our fitness function is based on the errors that we get from the **Training Error** and **Validation Error** which we receive after using the `get_error` function provided to us by sending a chromosome of the initial population as the argument. We have an inverse relation between the errors and the fitness score , i.e , lesser the calculated function , higher the score and hence likelihood of being carried forward. Our *fitness function* is as follows :

$$\text{Fitness} = (\text{training\_error} \times \text{training\_weight}) + \text{validation\_error}$$

The **Training Weight** was updated from time to time depending on what the **Training Error** and the **Validation Error** . The following are the values we kept for the **Training Weight.**

### 0 < Training Weight < 1

Initially we noticed that the **Training Error** considerably lower than the **Validation Error** of the overfit vector that we were given. We approached the optimization problem by

first decreasing the **Validation error**, and not allowing the **Training Error.** This was achieved by putting **Training Error** = 0.25, which in the selection ( explained later ) process gives preference to those chromosomes higher probability of selection which had lower **Validation Error** and which was less strict on the **Training error** increasing**.** Sometimes the value was even lowered down to 0.2 or 0.15 so as to make the **Validation Error** drop faster, and preventing the **Training Error** from shooting up. Conversely , we even made it 0.5 or 0.7 some times to **slow down the decrease** of training error.

### Training Weight = 1

This was done when the Training Error had become large and we wanted to bring both the validation and the training error to decrease simultaneously.

The code for above is:

```
def get_err(vector):
    train_valid_error = get_errors(SECRET_KEY , list(vector))
    return train_valid_error

def calculate_fitness(errors):
    return errors[0]*WT + errors[1]
```

The actual fitness function runs this function for every vector in the population and the **orders them in ascending order of scores** to return the actual generation

```
def fitness_function(population):
    training_errors = np.zeros(len(population))
    validation_errors = np.zeros(len(population))
    fitness_array = np.zeros(len(population))
    generation = []
    for i in range(len(population)):
        chromosome = population[i]
        error = get_err(chromosome)
        training_errors[i] = error[0]
        validation_errors[i] = error[1]
        fitness_array[i] = calculate_fitness(error)
        generation.append((chromosome , training_errors[i] , validation_errors[i] , fitness_array[i]))


    generation.sort(key=lambda x:x[3])
    generation = np.array(generation)
    return generation
```

## Selection

The idea of selection is that the *fittest* chromosomes are chosen and there genes are passed on to the next generation. Our selection process involves using both **Rank based Selection** and it goes as follow:

1. From the **current generation** if we have $n$ chromosomes we go on to select $\frac{n}{4} + 1$ number of fittest chromosomes from the generation.

   ```
   def select_parent_pool(generation):
       top_selection = int(math.ceil(len(generation) / 4)) + 1
       parent_generation = generation[:top_selection]
       return parent_generation
   ```

   This goes on to make our *parent pool*. It is from this pool we go on to make our parents. Such a pool was chosen to ensure only the best chromosomes get the chance to breed and make the selection process tighter.

2. Now from this *parent pool* we give ranks to the parents with the highest fitness being given too the fittest chromosome. For the fittest chromosome the rank it was assigned was $\frac{m}{\frac{m(m+1)}{2}}$ where $(m = \frac{n}{4} + 1)$, and for the second fittest the rank was $\frac{m-1}{\frac{m(m+1)}{2}}$ and so on till the least fit chromosome is assigned the rank $\frac{1}{\frac{m(m+1)}{2}}$. These ranks are the weights that have been given to the chromosomes, and now from the chromosomes we made weighted selections of 2 chromosomes each. This happned till *Population_Size/2* iterations so as to make Population_Size children.

   ```
   def make_parents(parent_pool):
       n = len(parent_pool)
       total = n*(n+1)/2
       weight_arr = np.array(range(1,n+1))[::-1]
       weight_arr = weight_arr/total
       print('WEIGHTS ARE')
       print(weight_arr)
       num_pars = int(POP_SIZE/2)
       parents = []
       while num_pars:
           pars = random.choices(list(enumerate(parent_pool)) , k=2 , weights=weight_arr)
           parents.append(pars)
           num_pars -= 1

       parents = np.array(parents)
       return parents
   ```

## Crossover

The crossover we used was the **Simulated Binary Crossover.**

The working behind this crossover is :

The two parents (p1 and p2) and two children should follow this equation -

$$\frac{c_2 + c_1}{2} = \frac{p_1 + p_2}{2}$$

A random number $\mu$ is chosen between 0 to 1 and according to this choice and the selection of $\eta_c$(distribution index) , $\beta$ is calculated as follows

$$\beta = \begin{cases} (2\mu)^{1/(\eta_c+1)}, if \mu \leq 0.5 \\ (\frac{1}{2(1-\mu)})^{1/(\eta_c+1)}, otherwise \end{cases}$$

Here the $\eta_c$(**Distribution Index)** determines if the children will be close to the parents or not. A higher index means closer children and vice versa. We chose a base value of 3 for $\eta_c$ with little variations. The two children are then calculated as:

$$c_1 = 0.5((1+\beta)p_1 + (1-\beta)p_2)$$

$$c_2 = 0.5((1-\beta)p_1 + (1+\beta)p_2)$$

The code for the following is:

```
def binary_crossover(father_chromosome , mother_chromosome):

    u = random.random()
    n_c = 3 #factor would be changed after some iterations

    if (u < 0.5):
        beta = (2 * u)**((n_c + 1)**-1)
    else:
        beta = ((2*(1-u))**-1)**((n_c + 1)**-1)

    child1 = 0.5*((1 + beta) * father_chromosome + (1 - beta) * mother_chromosome)
    child2 = 0.5*((1 - beta) * father_chromosome + (1 + beta) * mother_chromosome)
    child1 = np.array(child1)
    child2 = np.array(child2)

    return (child1, child2)
```

## Mutation

In our algorithm , we have kept a base mutation probability of 30% and range of 0.1.

*Mutation Probability* - The probability by which a weight of a vector can under go mutation

*Mutation Range* - The range of mutation around the weight. A range of *k* means the weight *w* can now change uniformly to any value between: *w - w*(k) and w + w*(k)*

We iterate over each weight of every vector and each weight has a this defined probability to under go the above mentioned mutation

The code is:

```
def mutate(vector  ,mut_range , mut_prob):
    mutated_vect = np.copy(vector)
    for i in range(len(mutated_vect)):
        w = mutated_vect[i]
        reduction_fact = random.uniform(-mut_range,mut_range)
        reduction = w*reduction_fact
        if random.random() <= mut_prob:
            w = w + reduction
        mutated_vect[i] = w

    return mutated_vect
```

Here *mut_range* represents the range of mutation and *mut_prob* represents the probability.

## Creating Children and Mutation

The next step is to create and store two children from each of the pairs of parents selected. Then these child vectors are then mutated with a probability of 30% and a range of 0.1 (lesser but bigger steps). The functions finally return the mutated new generation in an array.

```
def make_children(parents):
    children_population = []
    for i in range(len(parents)):
        father = parents[i][0][1]
        mother = parents[i][1][1]
        father_chrom = father[0]
        mother_chrom = mother[0]
        child_pair = binary_crossover(father_chrom , mother_chrom)
        children_population.append(child_pair[0])
        children_population.append(child_pair[1])

    children_population = np.array(children_population)
    return children_population
```

This function stores the two children after each crossover in an array

```
def make_offspring_population(children_population , mut_range , mut_prob):
    mutated_children = []
    for c in children_population:
        mutated_children.append(mutate(c , mut_range , mut_prob))

    mutated_children = np.array(mutated_children)
    return mutated_children
```

This function takes the returned children array and performs the mutation function on each of them. The final return is an array of now mutated children

### Calculating new fitness and choosing next generation

The fitness function is then run for this new mutated generation which calculated the new scores and sorts them in ascending order.

```
def get_mutated_fitness(mutated_children):
    mutated_generation = fitness_function(mutated_children)
    return mutated_generation
```

Now to choose the new generation in our algorithm , we used a strategy in which the **top 3 parents from the parent pool as well as this new mutated generation** along with all the fitness scores is combined. This is then again sorted in ascending error and then the **top *population_size*** vectors are chosen as the next generation. This was done to make sure the error doesn't drop in case the new offspring population all had worse scores. So if the top parents dominated even the newly generated vectors , **they were allowed to be carried forward**.

```
def make_next_gen(parent_pool , mutated_generation):
    new_pool = np.concatenate((parent_pool[:3] , mutated_generation))
    new_pool = new_pool[np.argsort(new_pool[:,3])]
    new_generation = new_pool[:POP_SIZE]
    return new_generation
```

## Hyperparameters

- Population Size = ***16*** - A larger population size was chosen as to make it more likely to have better children in the vast pool of children created , which was very hard to do in a smaller population

- Mutation Range = ***0.1(Base value)*** - We used a mutation range (explained above in mutation) of 0.1 for the most of the first generations to ensure healthy and impactful mutation. This value was changed according to our observations if the vectors were converging or not.
- Mutation Probability - ***0.3 (30 percent)(base value)*** - A low mutation probability was chosen so as to ensure that vectors are allowed to settle down on some values rather than changing rapidly. This value was again changed if we observed that the vectors were not mutating enough
- Distribution Index($\eta_c$) ***- 3(base value)*** *- We kept this as 3 as we initially didnt want to have children too close or children too far away from the parents.*
- Calculation of Fitness → Training Error weightage ***- 0.25(base value)*** *- A weight of 0.25 was given to the training error initially in the calculation of fitness score:* $score = \frac{1}{train\_error * weight + validation\_error}$ . This was chosen as the training error of the over fit vector started out to be very less (4*10e10) and validation error high (3*10e11). Therefore to allow training error to increase with some freedom and apply cost for validaition error for the same, this was chosen.
- Top Parent Pool Size ***- 5*** - Top 5 vectors from the generation were chosen to be considered to be made parents every time, This was to make the selection process sufficiently strict so as to ensure very good genes going forward. We found this to be an optimal value as anything less than this was resulting in less diversity as the same kind of genes were present every where , whereas anything less than this spoiled the newer generations as bad vectors were now allowed to seep in as parents.
- Pool for choosing next generation ***- 19 - Top 3 parents and 16 mutated children*** - The top 3 parents were also included as to make sure the worse children didn't increase the minimum error. This way generations wont be wasted in dealing with bad vectors

## Statistical Observations

- The main observation was that almost on all runs the errors of the vectors converged to the range - $9 * 10^{10}\ to\ 1.1 * 10^{11} (training),\ 1.1 * 10^{11}\ to\ 1.4 * 10^{11} (validation)$ , in about **30-40 generations.** Most number of times after 35 to 40 generations and these errors the errors remained settled in this minima.

GENERATION: 36

INITIAL POPULATION

| Number | Chromosome | Training Error | Validation Error | Fitness (Train + Valid) |
|---|---|---|---|---|
| 0 | [ 0.00000000e+00 -1.82064522e-12 -2.19650462e-13 5.32595559e-11<br>-1.81233256e-10 -1.62960578e-15 7.36379408e-16 2.66351554e-05<br>-1.96867869e-06 -1.74199591e-08 9.44343563e-10] | 8.86497e+10 | 1.13843e+11 | 1.27141e+11 |
| 1 | [ 0.00000000e+00 -1.71355554e-12 -2.32857064e-13 5.32935033e-11<br>-1.74652322e-10 -1.66578129e-15 7.38513668e-16 2.67000075e-05<br>-1.96867869e-06 -1.74122434e-08 9.43625933e-10] | 1.00864e+11 | 1.12242e+11 | 1.27372e+11 |
| 2 | [ 0.00000000e+00 -1.79283802e-12 -2.23997722e-13 5.32280790e-11<br>-1.79690696e-10 -1.62962184e-15 7.27038611e-16 2.63008004e-05<br>-1.96867869e-06 -1.74597386e-08 9.48043422e-10] | 8.68693e+10 | 1.18025e+11 | 1.44085e+11 |
| 3 | [ 0.00000000e+00 -1.66049852e-12 -2.42947464e-13 4.90122909e-11<br>-1.81233256e-10 -1.62960578e-15 6.77403500e-16 2.48431019e-05<br>-1.96867869e-06 -1.76331668e-08 9.64173818e-10] | 8.92314e+10 | 1.39613e+11 | 1.52997e+11 |
| 4 | [ 0.00000000e+00 -1.66116152e-12 -2.42851015e-13 4.90298745e-11<br>-1.81233256e-10 -1.62960578e-15 6.90417468e-16 2.48505210e-05<br>-1.96867869e-06 -1.76322841e-08 9.64091721e-10] | 8.91776e+10 | 1.39489e+11 | 1.66242e+11 |

- On coming out of the above mentioned converging minima (by more mutation and distant children in crossover) , in about **55-60 generations ,** both training and validation errors reached the $\leq 6.5 * 10^{10}$ range.

GENERATION: 60

INITIAL POPULATION

| Number | Chromosome | Training Error | Validation Error | Fitness (Train + Valid) |
|---|---|---|---|---|
| 0 | [ 0.00000000e+00 -1.38749322e-12 -2.15597186e-13 3.80830364e-11<br>-1.64694244e-10 -1.17919870e-15 8.53826845e-16 2.85784661e-05<br>-1.80344848e-06 -1.86831700e-08 8.76288487e-10] | 5.26464e+10 | 6.87053e+10 | 9.76607e+10 |
| 1 | [ 0.00000000e+00 -1.38748682e-12 -2.15617911e-13 3.80840453e-11<br>-1.64930071e-10 -1.17920985e-15 8.67520275e-16 2.85784661e-05<br>-1.80344848e-06 -1.86831700e-08 8.76288504e-10] | 5.26485e+10 | 6.87041e+10 | 9.76608e+10 |
| 2 | [ 0.00000000e+00 -1.38742858e-12 -2.15806427e-13 3.80932218e-11<br>-1.67075101e-10 -1.17931131e-15 8.67489529e-16 2.85784661e-05<br>-1.80344848e-06 -1.86831700e-08 8.76288658e-10] | 5.26681e+10 | 6.8694e+10 | 9.76614e+10 |
| 3 | [ 0.00000000e+00 -1.38722478e-12 -2.16466071e-13 3.81253316e-11<br>-1.74580879e-10 -1.17966633e-15 8.67381947e-16 2.85784661e-05<br>-1.80344848e-06 -1.86831700e-08 8.76289196e-10] | 5.27368e+10 | 6.86585e+10 | 9.76637e+10 |
| 4 | [ 0.00000000e+00 -1.40196704e-12 -2.16466071e-13 3.96834530e-11<br>-1.74580879e-10 -1.17966633e-15 8.67381947e-16 2.85784661e-05<br>-1.80344848e-06 -1.86831700e-08 8.76289196e-10] | 5.27368e+10 | 6.86585e+10 | 9.76637e+10 |
| 5 | [ 0.00000000e+00 -1.41671962e-12 -2.16466071e-13 3.81253316e-11<br>-1.65679502e-10 -1.17966633e-15 8.67381947e-16 2.85784661e-05<br>-1.80344848e-06 -1.86831700e-08 8.76289196e-10] | 5.27368e+10 | 6.86585e+10 | 9.76637e+10 |

- These points however **varied immensely with the way the fitness score was calculated**. The above observations were from if we kept training error weight as 0.25 initially and then increased it during the last generations.
- If the train weight was high from the beginning itself , the training and validation error would come to the ranges of 6-7*10^10 and 8-9*10^10 in only about 40 generations
- The trend that was observed was that higher the weight of training error the more tendency it had to get reduced along with validation error , which was undesirable as

training error was already pretty low and it was needed that we let it increase on only put a weight of 1 on validation error.

- We observed a lot of times that the errors were not improving for more than 10 generations sometimes. The relation of this with the mutation and crossover was as follows:
  - At very high mutation range and probability , this tendency was rare as in most cases the vectors would mutate and jump out of any minima
  - At decently high mutation and distribution index in crossover as 2 - If this was emplyed as the base heuristic then in most cases the vectors got stuck as the mutation became unnecssary and also was not sufficient enough to allow them to escape a minima after a point.
  - Low mutation and D.I as 3 - this was the base strategy and resulted in the convergence (getting stuck) at 1e11 values of errors. The above two were then used to get the vectors out. On average , if the vectors were stuck at a minima , about 5 to 10 generations were needed with the above tow strategies to get them out of the minima

## Heuristics

During the course of the whole project , a number of different strategies were tried out:

- **A lower population size (8 or 10) -** We stuck with this population for a quite a while during the project. The main drawbacks of this was that it didn't sufficient children to be created and therefore a lot of times it wasted many generation until a child vector started doing better than one of the parent vectors.
- **A higher population size (16) -** This strategy was very helpful. After deciding on this heuristic we stuck to it as it instantly started giving good results. This increased our server calls per generation but that was balance by the fact that now there was a higher chance of producing better children vectors.
- **Fitness Score Calculation:**
  - *train_error + validation_error* - This heuristic resulted in the training error getting too low and it ended up remaining overfitted.
  - *train_error*0.25 + validation_error* - This way the training error was allowed to increase and the validation error was decreased. Then at the point when both of them became equal we increased the weight to *0.6 or 0.7* which slightly decreased the rapidness of training error increase.

- *abs(training_error - validation_error)* - This was used after many generations when training error and validation error are around the same values in 1e11 range. After this way , the errors oscillated around the same points

- **Selection**
  - *Number of chromosomes in the parent pool* - We tried different pool sizes. Looking at top **3** chromosomes for parent selection proved to be ineffective as that meant a lot of chromosome were inhibited from going forward. For top **8** parents , it meat 2-3 bad vectors also came in the population hence ruining the genetic makeup. We found **top 5 to be perfect.**
  - Method of selection:
    - *Roulette Wheel Selection* - In this we gave weights to the top 5 vectors according to the score. But this was not that helpful as since the vectors had similar errors , so therefore each of them had almost equal chance of being selected , which ruined the purpose
    - *Rank Bases Selection* - This solved the above purpose and vectors were given weights according to their relative rank to the others in the top 5 pool

- **Crossover**
  - *Uniform Crossover* - This crossover was used by us during the initial days. In this we made it so that each gene had 50% chance of coming from the father chromosome or mother chromosome. Variations - (55% from the parent who has higher crossover instead of 50-50)
  - *Simulated Binary Crossover* - This is our current heuristic. This created more diversity as in simulated binary crossover both chilfren have all diffrent values from any of the parents genes(unless the parents are the same). So we didn;t have to rely on mutation a lot for diversity

- **Mutation Probability -** We initially kept this at **70%** but this resulted in the vectors jumping around too much and no convergence point was observable. A mutation probability of **30%** was found to be optimum
- **Choosing the next generation -** For this we chose the top 3 parents of the top parent pool along with the newly generated mutated children to choose the next generation.

This ensured that the very good vector genes were never completely lost due to bad children vectors

## Errors

The errors we obtained for our submitted 10 vectors are:

**Copy of Top 10 errors**

| Vector | Training Error | Validation Error |
|---|---|---|
| 1 | 2.65898e+11 | 1.2639e+11 |
| 2 | 2.28545e+11 | 1.24616e+11 |
| 3 | 1.51756e+11 | 1.07978e+11 |
| 4 | 1.46984e+11 | 1.16446e+11 |
| 5 | 1.6754e+11 | 1.14091e+11 |
| 6 | 1.05163e+11 | 1.06552e+11 |
| 7 | 1.94785e+11 | 9.93249e+10 |
| 8 | 1.4856e+11 | 1.27517e+11 |
| 9 | 1.5438e+11 | 8.04571e+10 |
| 10 | 8.4742e+10 | 6.07518e+10 |

Here are our observations/inferences:

- For the best vectors out of these , we submitted the errors where the training error was on the higher side and validation error was in the 1e11 to 1.2e11 range.
    - This ensures there is no chance of overfitting on the training data set as the training error is sufficiently high. The validation error is low but not lower than 1e11 which ensures that there is no chance of overfitting on the validation dataset too.
    - The high training error and sufficiently but not too low of a validation error makes us infer - The model does not try to over fit the training data and generalizes well on the validation data (without over fitting there as well)
    - Therefore we believe it will generalize well on the test set as it does not over fit and also performs and generalizes well on the practice unseen data (validation set)
- Few vectors in the middle have both validation and training error in the 1e11 to 1.5e11 range. This is to test if the limits of the over fitting boundary of the training data as we

were not sure if the high training errors in the first few vectors were too high and needed to be a bit lower

- The last few vectors submitted are extreme cases where the validation error is very low in the 1e10 range so as to check if such a low validation error is needed to perform well on unseen data
- If the generalization of validation set holds till unseen set , then the vectors with validation errors ~ 1e11 will do good and assuming a very high training error is what is needed to completely avoid over fit and still be low enough to perform well on validation , our first 2-3 vectors should perform very well