Fitness Function

Initially we saw that (Validation Error) = 2*(Training Error) approximately, on the overfit vector provided to us. And both errors are of order around ~15. So we decided on using fitness function such that it reduces sum of errors and also don't get overfit on the respective hidden dataset. Hence we first used fitness function to be F(te, ve) = 1 / ((te + ve) + 2(abs(te - ve))), where te & ve are train error and validation error respectively. This function seems to account for reducing sum of both error and also at the same time keeping small difference between them which will lead to decrease the overfit nature of model.

```
def get_fitness(te, ve):
    '''
    This function calculates the fitness
    for given list of errors. Higher the
    fitness more fit/perfect the vector.
    Returns the list of fitness for
    corresponding errors.
    '''
    fitness = []
    for i in range(len(te)):
        sum_err = te[i] + ve[i]
        abs_diff_err = abs(te[i] - ve[i])
        fit = ( sum_err) + ( abs_diff_err)
        fit = 1/fit
        fitness.append(fit)
    return fitness
```

Cross-Over Function

We followed the Singe-Point-Cross-Over method to achieve the crossover of two individual. We ran a loop of `POPULATION_SIZE//2`rounds and each will generate two children by crossing two parents. The parents will be selected randomly among the parent population. Logic is generate a random integer from 1 to 10 say X. Then two children will be defined by

```
    c1 = list(p1[:crossover_point]) + list(p2[crossover_point:])
    c2 = list(p2[:crossover_point]) + list(p1[crossover_point:])
```

Basically, the first parent was copied till a certain index, and the remaining was copied from the second parent. Similarly, for second child.

```
def cross_over(p1,p2):
    '''
```

```
    This function simply does the cross-over
    on two individual p1,p2 and returns c1,c2
    i.e. crossed-child.
    '''
    crossover_point = random.randint(1, 10)
    c1 = list(p1[:crossover_point]) + list(p2[crossover_point:])
    c2 = list(p2[:crossover_point]) + list(p1[crossover_point:])
    return c1, c2
```

Mutation Function

Mutation is a genetic operator that changes one or more gene values in a chromosome. Therefore, we implemented it in a way that it will mutate every gene of an individual which is 11-D vector here with a probability of `(MUTATION_PROBABILITY)/2`. We also counted for the fact that gene might be zero. Hence, to change it we replace gene with value `0` by a random number between `-0.01 to 0.01`. This will prevent that gene from being zero all the time during training of our model and also bring a very small variation. Next we kept our gene value between `MIN_GENE_VAL to MAX_GENE_VAL` while multiply it with random number between `0 to 1` for higher order gene/coefficients and with `1 + random.uniform(-0.03, 0.03)` for lower order gene/coefficients. The following code is for the same:

```
def mutation(crossed_popuplation):
    ...
        curr_vec = crossed_pop[i]
        for j in range(len(curr_vec)):
            if random.uniform(1,10)<=(10*(MUTATION_PROBABILITY)/2):
                if curr_vec[j]==0:
                    curr_vec[j] = random.uniform(-0.01,0.01)
                else:
                    if j > 4:
                        fac = random.uniform(0, 1)
                    else:
                        fac = 1 + random.uniform(-0.03, 0.03)
                    new_gene = fac*curr_vec[j]
                    if abs(new_gene)<10:
                        curr_vec[j]=new_gene
    ...
```

Hyperparameters

Population Size

As we were supposed to use limited API calls during the working span of assignment. It was very important to choose a proper population size of a generation. We initially kept it at 6. It was limiting the search space due to less randomness in population. Later when API calls increased we shifted to 15 but finally kept it to a satisfying level of 10.

Mating Pool

Initially we kept mating pool size to `6` and logic was such that next generation contains 4 parents and 7 crossed-mutated children from the mating pool. But that lead to overfit condition. We relied it as our vectors error got optimized to order ~11 but rank on the leader board fall down to 105. That obviously represent the case of overfit as our model performed well on train & validation data but not on test data. Finally we kept mating pool size to same as `POPULATION_SIZE` and logic was such that next generation contains only crossed-mutated children from the mating pool. We got an error of order 12 which was not as previous one but our rank improved to 50 that meant it is not overfit now and will perform well on testing data.

Cross-Over Point

We wanted to randomness during cross-over as much as possible so instead of choosing a fixed index as a cross-over we kept it a random number between 1 to 10 both including.

Mutation Factor

Initially we kept our mutation multiplication factor to be in range (0.95,1.05) such that we don't lost current generation fitness much and at same point bring some slight variation in the population, but only for lower order coefficients such that we can explore much deviation for lower order at time of convergence and kept long range(0,1) for higher order for making start with greater variation for more randomness in generation's vectors. Later we changed (0.95, 1.05) to (0.27, 0.33) and (0,1) to (0,0.3) while doing hit & trial we found errors to increased to order 15. hence discarded later changes.

Statistical Parameters

We made our code run for fixed number of generations i.e. `GENERATION_LOOP`. This was because of limited daily API calls. But after running till 70th generation

we got stuck with fixed error(both train & validation) of order 12. We reasoned for that to be stuck in local minima. Even after doing some changes in mutation factor as described in Hyperparameters section we didn't get much improvement. Therefore we can say running a loop till `~100` generation will give the optimized vectors with best errors globally.

Heuristics

- As mentioned in Fitness Function Section we implemented function F(te, ve) = 1 / ((te + ve) + 2(abs(te - ve))). Later, we tried other fitness functions but those didn't gave better result. One function i.e. F(te, ve) = 1 / ((te + ve) + abs(te - ve)) lead to reduction of errors to order around 12 but difference between error was poor as compared to previous fitness function. Some other functions that we tried:
  - 1 / (3* (te + ve) + 7* abs(te - ve))
  - 1 / (te + ve)
  - 1 / (4* (te + ve) + abs(te - ve))
- We didn't implemented probability during mutation which won;t worked well leading to much loss of parents gene.
- We tried mating pool size to 6 which won't worked well.(Explained in Heuristics section)

Final Vector & Error

- Generation : 118
- Vector : [9.071930543742546, 7.228528053270771e-07, 4.701488747229637e-11, 2.400690310127252e-10, -0.009633764416961064, -1.4263021690267203e-21, 1.6057832858201011e-21, 5.619827635464067e-15, -1.4868804786503433e-07, -2.7788980743367736e-13, 1.0093819206985834e-19]
- Train Error : 907006137307.031
- Validation Error : 543051366048.3366
- Theoretical validation argument : This vector has error in the order of ~11, which is the best error we saw and giving rank ~43. That means it is not overfit also and generaizes well as validation error is really very low. Hence, this could be vector on the server.

Trace File

`trace.json` contains all the information related to each and every vector of all generations for our final working code. It contains data in json format. We recommend to install json viewer extension for browser to view the file. Format of the data stored:

```
{
    "trace": [
    {
        "reproduction": {
            "parent": [...],
            "parent_fitness": [...],
            "selected": [...],
            "cross": [...],
            "cross_detail": [...],
            "mutated": [...],
            "child": [...],
            "child_fitness":[...]
        }
    },
    {
        "reproduction": {
            "parent": [...],
            "parent_fitness": [...],
            "selected": [...],
            "cross": [...],
            "cross_detail": [...],
            "mutated": [...],
            "child": [...],
            "child_fitness":[...]
        }
    },...
    ]
}
```

- `parent`is list of 10 11-D vectors i.e. our coefficient vectors
- `parent_fitness`is list of 10 15-D vectors. A vector has first 10 value as normal parent vector has and later 4 values are train error, validation error, fitness and selection percentage for the same corresponding coefficients vector.
- `selected`is 15-D vectors selected for crossing over from `parent_fitness` list.
- `cross` is a list of 11-D vectors obtained from crossing `selected` vectors
- `cross_detail` stores the children and corresponding parents
- `mutated` store vector after mutation
- `child` same as a mutated
- `child_fitness` same as `parent_fitness` but stores fitness and errors for `child`vectors