MDL ASSIGNMENT 3

Genetic Algorithms

Team No -8
Kanish Anand 2018101025
Akshat Goyal 2018101072

Overview:

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

In this assignment we were given a weight vector of 11 weights which was overfit on a training dataset we need to generalize that vector with use of Genetic Algorithm.

Steps of Genetic Algorithm :

- *Initial Population*
- *Fitness Function*
- *Selection*
- *Crossover*
- *Mutation*

Summary :

Initially an initial population of size 10 using the given overfit weight vector was created. To create this population of size 10 from one vector we added random noise in the given vector using python's *numpy.random.uniform()* function and created 10 different weight vectors from one vector.

```
# Given weight vector
 wghts = [ 0.0, 0.1240317450077846, -6.211941063144333, 0.04933903144709126,
0.03810848157715883,   8.132366097133624e-05, -6.018769160916912e-05,
-1.251585565299179e-07, 3.484096383229681e-08,   4.1614924993407104e-11,
-6.732420176902565e-12 ]

   population = []

   # Appending same population to list 10 times
   for i in range(POPULATION_SIZE):
       population.append(wghts.copy())

   # adding noise to make all 10 vectors different
   for i in range(POPULATION_SIZE):
       for j in range(len(population[i])):
           population[i][j] = population[i][j] + \
               np.random.uniform(-1*1e-13, 1*1e-13)
           population[i][j] = min(10, population[i][j])
           population[i][j] = max(-10, population[i][j])
```

After that fitness value for each vector was calculated by calling fitness function.

```
   for i in range(POPULATION_SIZE):
       er = fit(population[i])
       val = get_error(er)
       store_population.append((val, population[i], er[0],
er[1]))
```

In this ***store_population*** is a list of tuples where each weight vector is stored in form of tuple of 4 elements :

a. val: fitness value
b. population[i]: weight vector of 11 elements
c. er[0]: training error of current vector
d. er[1]: validation error of current vector

Then comes part of selection where two parents are selected for crossover. Here to calculate the probability value of each vector i.e the probability that this vector is selected when a vector is selected at random from a given population from crossover, we used formula probability = (1/er)/total . Here ***er*** is fitness value of that vector and reverse of that is taken because more the error value of a vector , less good it is therefore less chances it should have for getting selected as parent for crossover and total is sum of ***1/er*** over all vectors in this population . For this selection process python's *numpy.random.choice()* function was used in which we can pass the probability values and it selects the desired number of values from given values according to their probability values.

```
  for er in error:
          prob = 1/(er*total)
          probability.append(prob)
      ind = np.random.choice(POPULATION_SIZE, 2, replace=False,
p=probability)
```

After the two parents got selected we called crossover function to create a child from crossover of the two parents. Finally we got a child created from two parents selected based upon their fitness values upon which we apply mutation and get the final child.

```
    while i < POPULATION_SIZE:
          i += 1
          # choose two parents according to their probability
values
          ind = np.random.choice(
              POPULATION_SIZE, 2, replace=False, p=probability)
          # crossover of parents to make child
          children = crossover(ind, population)


          # mutation of children
          children = mutation(children)
          # store this children
          er = fit(children)
          val = get_error(er)
          store_population.append((val, children, er[0],
er[1]))
```

Now in this way we created 10 childs per iteration from an initial population of size 10 and stored then in ***store_population*** list in form of tuple and after this for the next iteration we selected the best 10 vectors among these all vectors of current iteration (calculated using fitness function) and these best 10 vectors from this iteration are used as initial population for the next iteration.

```
  store_population.sort(key=lambda x: x[0])
```

In this way we run several iterations in order to improve both training and validation accuracy. Also along with this in order to save population every time at end of iterations ran we saved best 10 vectors created till now in a json file and next time whenever we ran the algo we read those 10 vectors from that json file and considered that as our

initial population instead of creating initial population every time from given overfit vector.

```
    with open('population.json', 'w') as f:
       f.write(json.dumps(store_population[:POPULATION_SIZE],
indent=4))
    with open('population.json','r') as f:
        store_population = json.loads(f.read())
```

Diagram for Three Iterations Link :

- Iteration Number1 : https://www.draw.io/#G18YQKthRMl8HdmM_bW5YKz3x1vZSOKZ6i
- Iteration Number2 : https://www.draw.io/#G13rW6umo1tcW1wD6Pw8_G606oFL6RYHnE
- Iteration Number3 : https://www.draw.io/#G1bhWUUiRLxDlbuZmEG6GB5yHdUd_pOw-v

Also all 3 diagrams are attached as png images in zip file

Fitness Function :

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score. More the fitness value of a vector less is its probability of getting selected. Because more fitness score here means more total error. So we calculated probability accordingly by taking ***1/score*** value and based upon that we selected parents.

```
    def fit(vector):
            # returns fitness score for given vector
            err = get_errors(TEAM_ID, vector)
```

```
      # err : value returned by server upon calling with given
vector
      return TRAIN_RATIO*er[0] + VAL_RATIO*er[1]
```

Here TRAIN_RATIO , VAL_RATIO are adjusted to get better accuracy. Like in start the given vector was overfit so the train error was too less as compared to validation error. So initially we used TRAIN_RATIO as 0.25 and VAL_RATIO as 0.75 to balance both errors.

Crossover Function :

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.

```
      def crossover(ind, population):
                crossover_point = np.random.randint(1, len(population)-1)
                first_parent = population[ind[0]][:crossover_point]
                second_parent = population[ind[1]][crossover_point:]
                children = np.concatenate((first_parent, second_parent))
                return children
```

To apply crossover with two parents we selected a random integer ***'crossover_point'*** from 1 to 9 index which represents the crossover point i.e the point before which we will take all weights from first parent and after which all weights from second parent. Therefore in such a way we applied crossover on two parents and created a new child.

Mutation :

In certain new offspring formed, some of their genes can be subjected to mutation with a low random probability. This implies that some of the elements of the vector get changed by a small value. This is done in order to bring some randomness in vectors.

```
        def mutation(children):
            # adding some random number to elements of children with some
probability
       for i in range(0, 11):
          if np.random.uniform(-1, 1) < 0:
                continue
             children[i] = children[i] + np.random.uniform(-1e-13, 1e-13)
             children[i] = min(10, children[i])
             children[i] = max(-10, children[i])
             return list(children)
```

In this mutation function we iterated from 0 to 10 which represents index of elements in a vector and we applied mutation to any index with some probability for which we have used ***np.random.uniform*** function.
In order to apply mutation to some index we have added a random number in some range to that index using ***np.random.uniform(a,b)*** function where [a,b] represents range in which we want to select the number. This range is varied according to the training results. At the end min, max with value 10 is taken so as to make the range of elements from ***-10 to 10***.

Hyperparameters :

- Our ***Population size was 10***. Initially we prepared 10 vectors from given one vector by adding some noise in them so that all 10 become different.
    - The diversity of the population should be maintained otherwise it might lead to premature convergence.
    - Basically we need to select population size such that it is not too less that it can not accommodate both the changes made in the current generation (due to crossover and mutations) and the best population of the previous generation nor should it be too large that randomness to select best

vectors increases too much because due to that it will take too much time to converge.
    - Considering these points we tried with various population sizes like 4, 10 and 20 .
    - The problem with the population of size 4 was that we were not able to see the combined result of previous all generations in the new generations else only the result of previous one generation was reflected in the new generation. Due to this we were not able to achieve a good population for crossover.
    - With the population of size 20 we faced a problem that due to selection based on probabilities it was taking randomness too much i.e chances of selection of best vectors were decreased a lot. Due to which it was taking too much time to converge.
    - Therefore we select population size 10 which didn't suffer from any of the above problems.

- To Select Splitting point we used random.randint() function to select one index from 1 to 9 randomly which represents our crossover point.
- Number of iterations for convergence is not confirmed because initially it got converged in about 50-60 iterations and after that error stops decreasing but after that we changed parameters according to the current situation of errors and upon running again it again starts converging. Therefore its not confirmed but with particular hyperparameters no of iterations for convergence are near 50-60.
- TRAIN_RATIO and VAL_RATIO was selected based on current population. Like initial population had very less train error and very high val error. So initially we chose TRAIN_RATIO as 0.25 and VAL_RATIO as 0.75. So we kept on changing these ratios based on the current population. If both were almost equal we kept both as 0.5.
- Also while using various fitness functions we changed their parameters a lot throughout the training process.

Techniques :

- We tried with many different fitness functions to calculate the fitness scores :
    er[0] : train error

er[1] : validation error

- **Score = TRAIN_RATIO*er[0] + VAL_RATIO*er[1] :** In this score function we adjusted train and val ratio in order to decrease both the errors and bring them close.. Because if we want to generalise our weight vectors then it should perform better on both the train and val error. Like if in our population the train error was too less and val error was too high like in the initial one we made the parameters as 0.25 , 0.75 respectively whereas on training if both come close then we want to decrease both erros equally in that case we made both parameters 0.5, 0.5. Thus we tried with various parameters in this score function to reduce the error value.

- **Score = er[0]*er[1]** : This will focus on reducing both train error and validation error as when both errors are less then only their product will be less. But the problem we faced with this function was that sometimes it bends the population to such a side where one of train or validation error was too less as compared to another which is not good as we want to generalise the weights.

- **Score = er[0]*(er[1]**val) :** Here we have taken **er[1]**val** in order to increase power of validation error. Eg. If we are training and always validation error is not decreasing much whereas train error is decreasing too much in such cases we tried this fitness function where we gave more focus on decreasing validation error by raising its power so that whenever validation error decreases it will cause much good effect to fitness score as compared to decrease of training error. This faced a problem sometimes that validation error decreased too much as compared to train error.

- **Score = abs(er[0]-er[1])*(er[0]+er[1]) :** In this function we included two factors, one is difference of train and val error and the other one is their addition. We tried this because in the score function of multiplication of both the errors we were facing a problem that one was decreasing too much whereas the other was large because that too decreased the overall product. So in order to remove that problem we used this function which focused on making both train and val error close too i.e there is not much

gap in both which is required for generalisation of vectors and also it focussed on decreasing their values.

- We tried to make some changes in crossover function eg :
    - Instead of taking initial some part of first parent and second half of other parent we iterated for each index and randomly selected one of both parents and took its weight at that place. But using this function we got stuck at some local minima always i.e train and val error decreased till some point only which was not a good error and after that it was not getting better. Therefore we decided to use the previous function of crossover where we selected the first part from one parent and second part from some other parent.

- One technique which worked very well was that while taking the initial population instead of taking all 10 vectors same as the given overfit vector I added some random noise to the vectors in range(-1e-13,1e-13). This worked very well as in the start only we were able to reduce the error from starting error. Reason for this is that this added randomness to the given overfit vector which helped us to get good results.

- For the mutation part we played a lot with the value to be added.
    - Initially we tried with a random number in range (-5,5) in mutation but it resulted in drastically bad results as this was increasing error too much.
    - Then we tried with a random number in range(-1e5,1e5) this also suffered from the same problem.
    - So then we tried with range(-1e-13,1e-13) this worked very well at initial stages and helped us a lot to reduce the error.
    - But this also had a major problem : what it was doing is that to the weights were in range near this it was performing very well with those numbers but with weights were of order like say 1e-01 this was leading to almost negligible change . So for that we added to each weight a number which was weight*1e-03 and this worked very well to remove previous issue
    - We experimented a lot with probability values for mutation also throughout the training process and noticed the fact that probability value should be too less for good errors

- One major thing which we noticed in the weights was that among the 11 weights if we changed initial weight values by very less amount they were not giving any significant changes whereas changing last weight values gave very significant changes with just small changes in their weights. Reason for that was that the order of initial weights was in the range of 1-e01 - 1e-02 whereas that of the last ones was in range 1e05 - 1e013. But we took care of this problem with our mutation function where we added a number*1e-03 to that number. Also due to noticing this thing we tried with one more change where we added a divide number to be added by (i+1) where i is its index. It helped us to add less more values to starting numbers and less value to ending numbers while mutation.
- Also in order to restore best vectors of previous population too along with best vectors obtained in the current iterations we saved the best 10 vectors always to a file instead of just saving the vectors created in the current population.

Finally vector which I think will perform best on unseen dataset is :

```
1.0495532567575542e-12,
9.996846796957856,
-6.545830077493285,
0.058182145026023105,
0.038183172909594734,
8.189843231811995e-5,
-5.9997913628066724e-5,
-1.234489279710759e-7,
3.4782006019793494e-8,
3.8235399631976284e-11,
-6.710205279851606e-12
```

Although we were able to get vectors which had train and val error less than one given by this one but we think this will perform best on unseen datasets because in this vector both training and validation errors are almost near equal and good enough because in order to generalise the weights both the training and validation error should be near

equal. If consider like initial population where training error was too low but validation was too high that type of population is an overfit population and not a generalised one. So in order to perform well on an unseen dataset we need to have a generalised population which may give more error than overfit one but should give almost equal errors on both training and validation error.