Genetic Algorithms

Definition

Genetic Algorithms are optimisation algorithms that imitate Charles Darwin's description of the natural evolution. We use the theory of *Survival of the Fittest* to calculate the chromosome (vector) with the best genes (weights). This algorithm follows a certain flowchart as given:

> Initialisation of Population Evaluate Fitness Select best chromosomes Crossover the chosen chromosomes Mutate randomly (if needed) Repeat the above steps for N generations Return best chromosomes

Code Explanation

Initially

We started off by understanding the `get_errors` function provided in the `client.py` file. We used it to obtain the Train error and Validation error of the provided overfit vector. On obtaining a simple understanding, we tried to calculate fitness using `fitness = -(train_err + valid_err)`. We realised that the fitness was coming out negative. So we changed the function as we'll explain later.

The first crossover function we tried was the single point crossover. We realised the similarity of the given problem with the *unbounded knapsack problem* except we were't given a total weight.

On reading more about genetic algorithms and implementations, we came accross the Russian Roulette system of breeding children. We tried this out to achieve a slightly better result. This was the progress till the intermediate evaluation.

We saw that the leaderboard put us at the lower end of the list. Russian roulette was bringing in too much randomness and we started searching for more optimised methods. We read research papers and more methods such as the Ant Colony optimisation and more.

Eventually we tried the stimulated binary crossover function and random mutations to achieve a slightly better result.

After that progress seemed to increase rapidly. We decided to start from scratch. We set the population per generation to 30, and number of generations to be 10, this seemed to optimise the function but we started running out of server calls pretty quickly, as each run used about 350 server calls.

We realised that starting with the overfit vector to generate the initial population left very little randomness. So we started with a vector including only 0 as weights. Every time a vector was mutated, it did this according to the overffit vector, to reach the exponent.

We decided to only include the best of each population to generate the next generation. Which again greatly optimised our result. We soon realised that this again was not a very good way, as if the previous generation had a better chromosome, it would get left behind, as the new population will only include the children. This lead to a new idea to integrate both populations and select only the chromosomes with the best fitness.

We changed the fitness function as `abs(train_err*TRAIN_FACTOR + validation_err)`. We added the train factor due to the difference between the magnitudes of the train error and the validation error. On hit and trial, we figured that the best value to reach an optimised result, was 0.7. At a train factor equal to 0.5 or 0.6 the fitness was rapid and would often produce a slightly overfit vector.

Walkthrough

Mutation function:

```
    for i in range(CHROMOSOME_SIZE):
        mutation_prob = random.randint(0, 10)
        if mutation_prob < 3:
            if i <= 4:
                vary = 1 + random.uniform(-0.05, 0.05)
            else:
                vary = random.uniform(0, 1)
            rem = overfit_vector[i]*vary
            if abs(rem) <= 10:
                child[i] = rem
    return child
```

Since the initial population was a vector of zeros, we needed mutation to bring the vector closer to the optimised result and would do so with a random probability of multiplying the overfit vector with a variation.

Crossver function:

```
 child1 = np.empty(11)
child2 = np.empty(11)
u = random.random()
n_c = 3
if (u < 0.5):
    beta = (2 * u)**((n_c + 1)**-1)
else:
    beta = ((2*(1-u))**-1)**((n_c + 1)**-1)
parent1 = np.array(parent1)
parent2 = np.array(parent2)
child1 = 0.5*((1 + beta) * parent1 + (1 - beta) * parent2)
child2 = 0.5*((1 - beta) * parent1 + (1 + beta) * parent2)
return child1, child2
```

This is the single point crossover function. We break of two parents at a random point and interchange their chromosomes.

Mating pool generation:

```
 population_fitness = sorted(population_fitness, key=lambda x:x[-1])
mating_pool = population_fitness[:MATING_POOL_SIZE]
MATING_POOL.append(mating_pool)
return mating_pool
```

As seen here, we take only the top 10 vectors to generate the children.

New generation

```
 generation = np.concatenate((parents_fitness, children_fitness))
generation = sorted(generation, key=lambda x:x[-1])
generation = generation[:POPULATION_SIZE]
return generation
```

We combine both populations, the best 10 of the previous one, and the new children, sort and cut them off at the population size. Thus, retaining the best chromosomes.

Dictionaries

The `UNIVERSAL_DICT` contains all generations and all vectors of all generations. The `CHILD_DICT` contains all vectors which belong to all generations after the initial population with information about their parents.