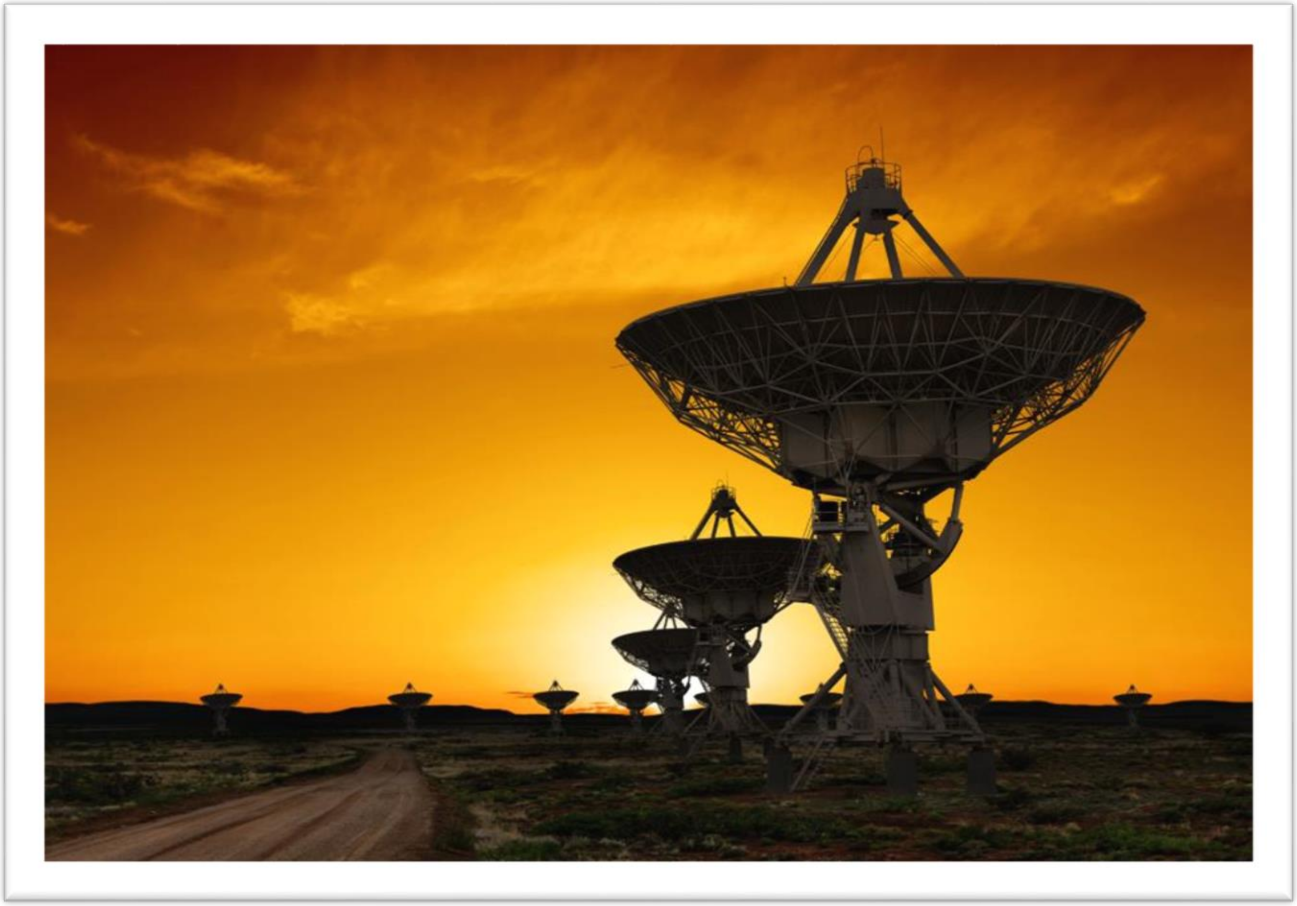


MDL Assignment 3 [TEAM 38]

Genetic Algorithms

Abinash Maharana: 2018111033

Shivansh Anand Srivastava: 2018101086

SIMULATING 3 ITERATIONS LINK:

https://anandshivansh.github.io/team38.github.io/

Summary

Our genetic algorithm tested error against limits and the ratio of validation to train error and progressed based on that. Basic steps

```
    #     print(agent.ar)

    for generation in range(generations):

        print('Generation: ' + str(generation))

        agents = fitness(agents)
        # for agent in agents:
        #     print(agent.fitness)
        agents = selection(agents)
        agents = crossover(agents)
        agents = mutation(agents)
        # agents = mutation(agents)
        # agents = mutation(agents)

        # gen after gen
        f2 = open("gendata.txt", "a")
        for agent in agents:
            f2.write(str(agent.ar) + '\n fitness for #' +
                     str(agents.index(agent)) + 'is ' + str(agent.fitness))
            f2.write('\n')

        f2.write(
            '\n---------------------------------------------------------------\n' + str(generation))

        f2.close()
        f = open("responses.txt", "a")
        f.write(
            '\niter is:---------------------------------------------------------------\n' + str(generation))
        f.close()

    # Cutoff
        if any(agent.fitness > 90 for agent in agents):
            print('Threshold reached!')
            exit(0)
```

Initialization;

Loop:

Fitness

Selection

Crossover

Mutation

Check threshold

Those file writes are for debugging output data and so that we do not lose our work.

Strategy

Bringing the validation error down to a reasonable value by coarse tuning, in which scenario train error also went up, so this tackled overfitting.

Adjusting the ratio to get it to almost 1:1 (MSE is roughly 10^7 now)

Decreasing train error after constraining validation error to a max value (10^7) (after the ratio became stable)

Decreasing validation error after getting a reasonable train error.

Mutation was applied randomly to all parameters initially (coarse tuning), after hitting the limit of error, only few parameters were mutated

After that fine tuning was applied by changing fitness function and erval

Heuristics

In the fitness function, changed the base_val_err and ratios once test:validation ratio got stabilized to put more emphasis on minimizing errors then.

Made the base_val_err adaptive to the errors (tighten the bound regularly, if errors are getting lesser).

Introduced a base_sum_error to decrease error and thresholds for each error to keep them bound

Removed those thresholds after reaching around total error = 1400000

Initially used a larger population size (90). Then changed it to 10. Then to 20.

Parameters

- Population size : 20
- Cutoff : scores > 900/1000 in fitness function (during coarse validation only)
- Generations: 50 (ideally till convergence but had to change because of debugging and constraints

Ideally pool size and generations should be as high as possible but due to a limit on requests these were chosen like this.

Chromosomes = Agents
Genes = Agent.ar values

Statistics

The coarse algorithm converged roughly in about a day and we got errors as ~600000 and ~900000 respectively.

Explanation of the functions

Note: Removed code for trace while taking screenshots

Initialization:

Each agent begins with some parameter slightly mutated to create diversity in the initial group.

```
def init_agents(population, ar):

    ret = [Agent(ar) for _ in range(population)]
    for a in ret:
        k = random.randint(1, 10)
        l = random.randint(1, 10)
        a.ar[k] += random.uniform(-a.ar[k] *
                                   0.00000001, a.ar[k]*0.00000001)
        a.ar[l] += random.uniform(-a.ar[l] *
                                   0.00000001, a.ar[l]*0.00000001)

    return ret
```

Selection:

Selects the top 80% agents, ordered by fitness (runs each round)

```
def selection(agents):

    agents = sorted(agents, key=lambda agent: agent.fitness, reverse=True)

    agents = agents[:int(0.8*len(agents))]
    # print('from sel' + str(len(agents)))
    return agents

```

During fine tuning directly sum of values for errors was taken so “reverse” became False.

Crossover:

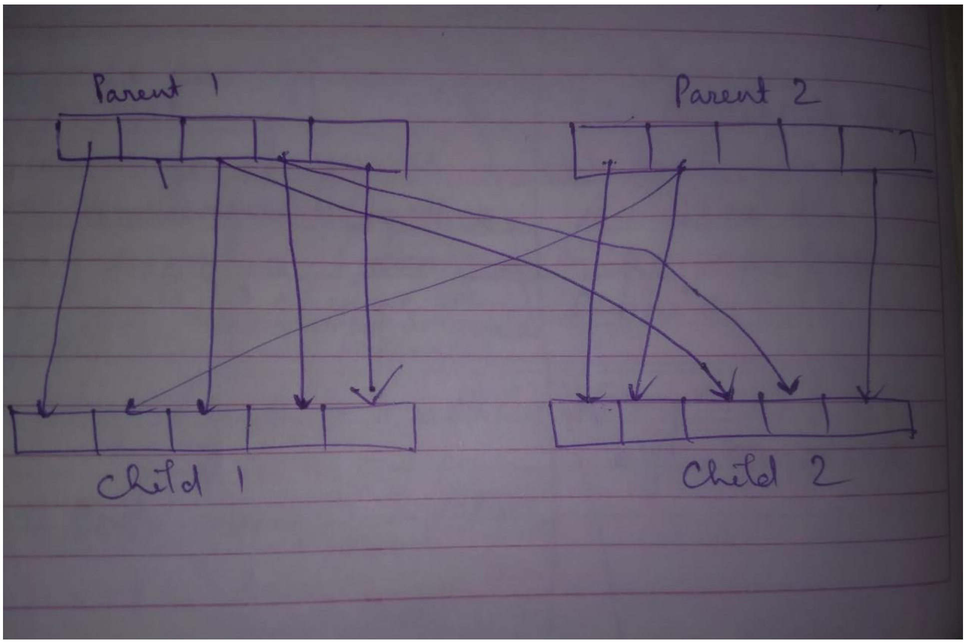
Two agents are randomly selected and then randomly mixed up.

So the deficit made in selection is fixed now

```python
def crossover(agents):
    offspring = []

    for _ in range(0, population/10):
        parent1 = Agent(ar)
        parent2 = Agent(ar)

        yy = random.choice(
            list(filter(lambda agent: agent.fitness > 0, agents)))
        # yy = agents[0]
        parent1.ar = yy.ar.copy()
        allowed_values = agents.copy()
        allowed_values.remove(yy)
        parent2.ar = random.choice(
            list(filter(lambda agent: agent.fitness > 0, allowed_values))).ar.copy()

        child1 = Agent(ar)
        child2 = Agent(ar)

        # Can/May change this
        child1.ar = parent1.ar.copy()
        for idx in range(0, len(parent2.ar)):
            if random.uniform(0.0, 1.0) < 0.6:
                child1.ar[idx] = parent2.ar[idx]

        child2.ar = parent2.ar.copy()
        for idx in range(0, len(parent1.ar)):
            if random.uniform(0.0, 1.0) < 0.6:
                child2.ar[idx] = parent1.ar[idx]

        offspring.append(child1)
        offspring.append(child2)

    agents.extend(offspring)

    return agents
```

Crossover randomly cross-mixes distinct parents genes to produce children.

Crossover diagram:



Mutation:

Note: the erval is error % value, which was varied from time to time.

In mutation, for each agent, with random probability, a random value from within a percent of the parameter (given by erval) is added to or subtracted from it.

```python
def mutation(agents):
    # modify this
    erval = 0.000000000000001  # 1e-17
    for agent in agents:

        for idx in range(0, len(agent.ar)):

            if random.uniform(0.0, 1.0) <= 0.8:
                new_ar = []

                for idd in range(0, idx):
                    new_ar.append(agent.ar[idd])

                new_ar.append(agent.ar[idx] +
                              random.uniform(-erval, erval))
                for idd in range(idx+1, len(agent.ar)):
                    new_ar.append(agent.ar[idd])

                agent.ar = new_ar.copy()
                # print('mutationsuccess')

    return agents


# Running the code
Genetic_Algorithm()
```

Fitness:

This function was the most difficult to optimize.

First *get_errors()* is called for all agents.

Then their fitness (out of 100) is calculated part by part according to individual errors, sum of errors and difference of errors.

The ratio of these has been changed from time to time to optimize the algorithm.

Parameters:

```
population = 20
generations = 100
base_val_thr = 1000000.1623710159
base_train_thr = 1000000.0000000000
base_val_err = base_val_thr * 2
base_train_err = base_train_thr * 2
base_sum_thr = base_val_thr + base_train_thr
base_sum_err = base_sum_thr * 2
```

```
def fitness(agents):
    # maxer = 0
    for agent in agents:
        cap = foo(agent.ar)
        train = cap[0]
        val = cap[1]

        # bve = 40*(1-(max(val, train)/base_val_err))
        bve = 10 * (1-(val/base_val_err))
        bte = 30 * (1-(train/base_train_err))
        thr = 40 * (1-((val+train)/base_sum_err))

        agent.fitness = 20 * \
            (1-(abs(val-train)/max(val, train))) + bve + bte + thr
        # if(agent.fitness < -1):
        #     agent.fitness = -1
        # if(train > base_train_thr or val > base_val_thr or thr > base_sum_thr):
        # agent.fitness = -2
        # if(thr > base_sum_thr):
        #     agent.fitness = -2

    # if(base_val_err > maxer):
    #     base_val_err = maxer + maxer*0.01

    return agents
```

Fine tuning fitness

```python
def fitness(agents):
    # maxer = 0
    for agent in agents:
        cap = foo(agent.ar)
        train = cap[0]
        val = cap[1]

        bve = 0 * val**4
        bte = 0 * train**4
        thr = 1000 * (val+train)**5

        agent.fitness = 0 * \
            (1-(abs(val-train)/max(val, train))) + bve + bte + thr
        # if(agent.fitness < -1):
        #     agent.fitness = -1
        # if(train > base_train_thr or val > base_val_thr or thr > base_sum_thr):
        # agent.fitness = -2
        # if(thr > base_sum_thr):
        #     agent.fitness = -2

    return agents
```

Most probably the reason we got stuck at our best value for a long time is because of getting a local maxima ☹ .

Although very slow, fine tuning is working right now.