1. Introduction

Genetic Algorithm (GA), developed by John Holland and his collaborators in the 1960s and 1970s (Holland, 1975; De Jong, 1975), is a search-based optimization algorithm based on biological evolutionary principles (*Genetic algorithm* 2021). These algorithms are powerful optimization tools since they don't need any extra information about a function's features, and thus can and have been used in a wide range of scenarios.

In this project, I aim to explore the use of GA with Neural Networks (NN) to see if they could provide better optimization of weights between the network layers leading to better performance. I first use GA to solve a simple optimization problem to familiarize myself with its workings and fundamental principles, then I use a more complex GA implementation to optimize the weights in a 9x15x1 classification NN. The NN is an open-source project and can be found here. I compare the performance of the GA with a standard Gradient Descent (GD) approach for optimizing the neuron's weights.

The first optimization problem was solved quite satisfactorily by the GA, despite the GA itself being very simplistic. Optimizing the weights in the NN proved to be a much more difficult problem for a variety of reasons I will be discussing later.

In this report, I will be discussing my target problems, my implementations of some key functions and algorithms and the subsequent results. I will provide a more in-depth look into the implementation of GA in this project, and also elaborate on some of the key challenges I faced along the way. I have also tried to suggest reasonable improvements that could be made to improve the project.

2. Problem Definition and Algorithm

2.1 Task Definition

- Task 1

  For task 1, I have a function called *foo* in three variables. The function is arbitrarily defined, and the only essential information for GA to function is that foo is a function of 3 variables—x, y and z. The task is to find values for the three variables such that the function returns a value as close to 0 as possible, or as close to 0 as I would like. As such, a solution is simply a set of three values.

- Task 2

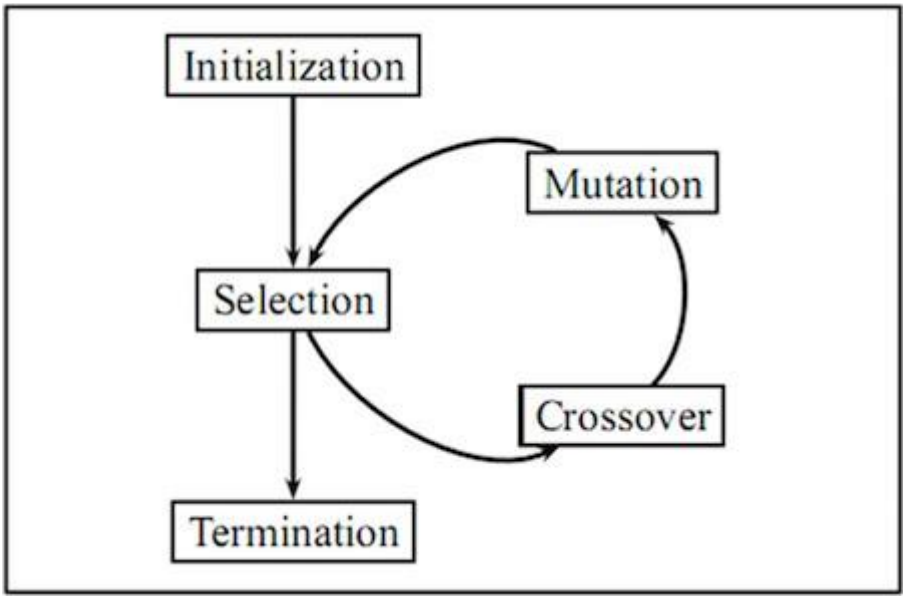
  Task 2 is more complex and much more testing. For this task, I have a NN consisting of three layers i.e., one input layer, one output layer and one hidden layer. The NN is a classification NN trained on The Wisconsin Cancer Data-set, and uses a cross-entropy loss function. The input layer consists of 9 neurons, the hidden layer has 15 and the output layer only has 1 since the output is always binary.

  The task for the GA is to calculate weights for the connections between the neurons in consecutive layers. As such the solution for this task is a set of two weight matrices representing the weights of the neurons in the input and the hidden layer. I also then train the NN with GD to be able to compare the results and see if GA provides any distinct advantages. The neurons' biases remain constant so that both GA and GD work only to optimize the weights between network layers.

2.2 Algorithm Definition

- Overview

For both the tasks, GA is being used to find a combination of elements such that they maximise some fitness function, and a final solution is accepted once the algorithm has run a certain number of times or if a certain pre-determined threshold fitness value is achieved.



GA has 4 important steps: initialization, selection, crossover and mutation, and finally termination. These steps roughly correspond to the process of natural selection, and fundamentally it is the same principle: fitter members survive and proliferate while the unfit members die off and don't contribute to the gene pool of future generations (*Introduction to evolutionary algorithms* 2021).

- Task 1

  Step 0. Fitness function

```
def fitness(x,y,z):
    ans = foo(x,y,z)

    if ans == 0:
        return 99999
    else:
        return abs(1/ans)
```

The fitness function evaluates the fitness score for a particular solution. The higher the fitness score, the better the solution. Since the perfect solution would make "ans" in above code 0, the fitness of such a solution should ideally be infinity since I have then solved the problem, but here I've just used a very big number to indicate success. In the case where "ans" is not zero, I return a score that is inversely proportional to how far from the ideal answer the current answer is i.e., the farther the answer is from 0, the lower its fitness score.

  Step 1. Generating initial population

```
solutions = []
for s in range(1000):
    solutions.append( (random.uniform(0,10000),
                       random.uniform(0,10000),
                       random.uniform(0,10000)))
```

This step creates an array of 1000 random sets of values for (x,y,z). This is the initial population of solutions that I would progressively “evolve” to find our optimum solution. The array “solutions” stores the population of solutions for every generation.

Step 2. Finding the best performers in a population

```
rankedsolutions = []
for s in solutions:
    rankedsolutions.append( (fitness(s[0],s[1],s[2]),s) )
rankedsolutions.sort()
rankedsolutions.reverse()

print(f"=== Gen {i} best slutions === ")
print(rankedsolutions[0])

if rankedsolutions[0][0] > 999:
    break

bestsolutions = rankedsolutions[:100]
```

For every generation of solutions, I calculate the fitness score for every member of the population. Then I pick out the 100 best performers—the solutions with the highest fitness scores—of a generation into the array “bestsolutions”.

Step 3. Crossover and mutation

```
elements_x = []
elements_y = []
elements_z = []
for s in bestsolutions:
    elements_x.append(s[1][0])
    elements_y.append(s[1][1])
    elements_z.append(s[1][2])

newGen = []
for _ in range(1000):
    e1 = random.choice(elements_x) * random.uniform(0.99, 1.01)
    e2 = random.choice(elements_y) * random.uniform(0.99, 1.01)
    e3 = random.choice(elements_z) * random.uniform(0.99, 1.01)

    newGen.append((e1,e2,e3))

solutions = newGen
```

I create a new generation of solutions using the best performers from the previous generation. The arrays “elements_x”, “elements_y” and “elements_z” contain the values

for x, y and z from the array “bestsolutions”. The new generation is formed by randomly picking a value of x, y and z from the three arrays and then adding a random variation of 2% to the values.

Step 4. Termination

```
if rankedsolutions[0][0] > 999:
    break
```

Finally, in every generation I check the fitness score of the top performer and if the fitness score is above a certain threshold, I accept the solution. In this case, the threshold value is 999 i.e., if the output of the function is in the range [-0.001, 0.001] I would accept the inputs as acceptable solutions. It’s important to allow a margin of error since I really don’t know anything about the function, which could very well be discontinuous. Also, this allows us to save time and resources since for most practical optimization problems, optimizing beyond a certain point would become redundant.

- Task 2

The actual implementation of GA for task 2 would take up too much space here, and probably wouldn’t add much value to this report since the steps carried out are very much similar to task 1 but with two key differences. The first one obviously is the actual implementation of the fitness, crossover and mutation functions is way more complex now. And the second difference is that for this task I have to very frequently convert between 2-D and 1-D so of the weights of the NN. This is because while GA works with vectors, the weights are better represented in the NN as matrices to facilitate forward propagation operations such as dot product.

With these two key differences in mind, I will be discussing the implementations of the key functions of the GA without going into too much detail about how GA is actually implemented.

1. Fitness function

```
def fitness(self, weights_mat):
    fitness = np.empty(shape=(weights_mat.shape[0]))
    for sol_idx in range(weights_mat.shape[0]):
        curr_sol_mat = weights_mat[sol_idx, :]
        self.ginit(curr_sol_mat[0], curr_sol_mat[1])
        Yh, loss=self.forward()
        if(loss == 0):
            fitness[sol_idx] = 99999
        else:
            fitness[sol_idx] = abs(1/loss)
    return fitness
```

For this task the fitness of a solution is how close the produced outputs of the NN are to the expected outputs provided in the training data. This closeness is represented as the variable “loss” and is calculated at the end of every forward propagation through the NN. The lower the loss value, the higher the fitness score.

2. Initialization

```
initial_pop_weights = []
for _ in np.arange(0, sol_per_pop):
    input_HL1_weights = np.random.randn(self.dims[1], self.dims[0]) / np.sqrt(self.dims[0])
    HL1_output_weights = np.random.randn(self.dims[2], self.dims[1]) / np.sqrt(self.dims[1])

    initial_pop_weights.append(np.array([input_HL1_weights,
                                         HL1_output_weights]))

pop_weights_mat = np.array(initial_pop_weights)
pop_weights_vector = ga.mat_to_vector(pop_weights_mat)
```

The initial population of weights is calculated using python's random functions. The weights are created as a list of random values first, and subsequently transformed into a matrix and a vector.

3. Selection

```
def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents
```

The "select_mating_pool" function selects the top performers of a generation to be the parents for the next generation. It receives the "fitness" scores for the entire generation as a parameter, and then iteratively picks the solution with the highest fitness score and copies it into "parents".

4. Crossover

```
def distinct_crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    # The point at which crossover takes place between two parents. Usually, it is at the center.
    crossover_point = numpy.uint32(offspring_size[1]/2)
    #print(offspring_size)
    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken from the first parent.
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        # The new offspring will have its second half of its genes taken from the second parent.
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring
```

The crossover function uses the selected parents to generate the next generation of solutions. The "parents" vector represents the parents whose values would be passed on to the next generation, and the "offspring_size" represents the size of the next generation vector. The next generation consists of all the parents chosen from the current generation, and their offspring. To generate an offspring, the function picks two distinct parents from the "parents" vector and then copies the first half of parent at "parent1_idx" into the first half of the offspring and the second half of parent at "parent2_idx" into the second half of the offspring. This is repeated for as many offspring as the next generation needs.

5. Mutation

```
def mutation(offspring_crossover, mutation_percent):
    num_mutations = numpy.uint32((mutation_percent*offspring_crossover.shape[1])/100)
    mutation_indices = numpy.array(random.sample(range(0, offspring_crossover.shape[1]), num_mutations))
    # Mutation changes a single gene in each offspring randomly.
    for idx in range(offspring_crossover.shape[0]):
        # The random value to be added to the gene.
        random_value = numpy.random.uniform(-1.0, 1.0, 1)
        offspring_crossover[idx, mutation_indices] = offspring_crossover[idx, mutation_indices] + random_value
    return offspring_crossover
```

The mutation function's objective is to introduce variation in solutions. The function receives "mutation_percent" as a parameter, the "mutation_percent" determines how many indices of an offspring are altered and the number of mutations to be made is stored in "num_mutations". Then I randomly pick the indices to be mutated in every offspring, and add a random value in the range [-1, 1] to the value at that index.

3. Experimental Evaluation

3.1 Methodology

- Task 1

For task 1 my objective was simply to simply observe the efficacy of GA, and thus the only performance parameter worth measuring is the increase in fitness of the population per generation.

- Task 2

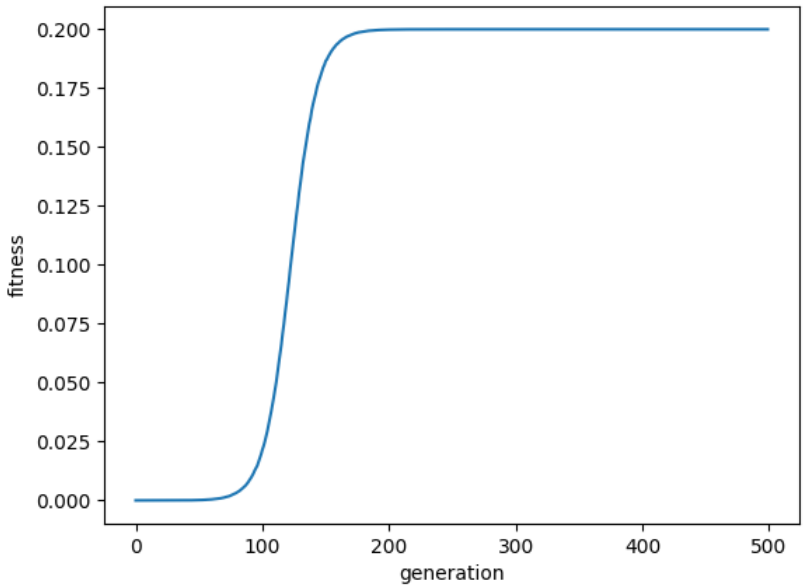
For task 2 the objective was not only to measure the performance of GA, but also to compare its performance with GD as a tool for training the NN.

I measure the performance of GA in terms of loss reduction per generation, where loss is an averaged difference between the outputs produced by the NN and the expected outputs for a particular set of weights for the neurons. The performance of GD is measured in loss reduction per iteration, where an iteration is simply one call of GD.

To compare the two performances however is a bit tricky since one generation of GA runs the NN many more times than one iteration of GD since a generation of GA calculates the fitness of every member. Thus, I control for the number of times the NN runs when comparing the performances of GA and GD.

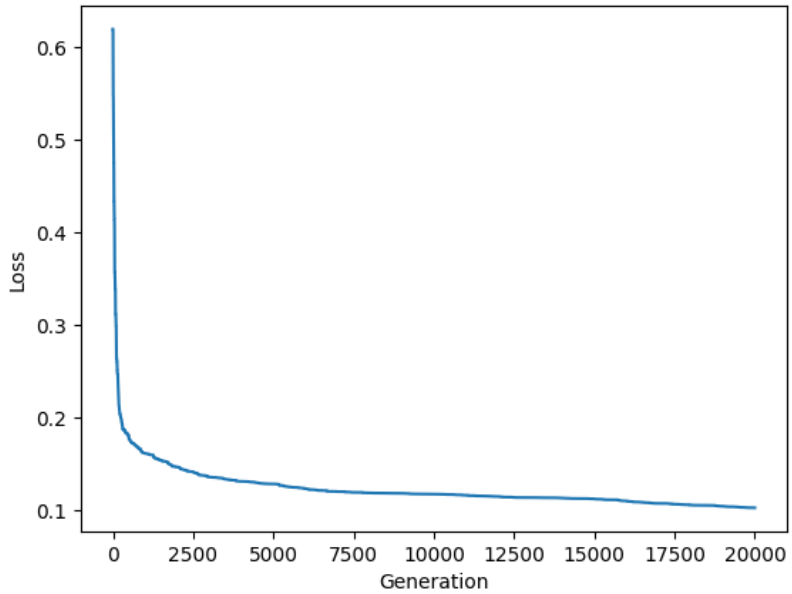
3.2 Results

- Task 1

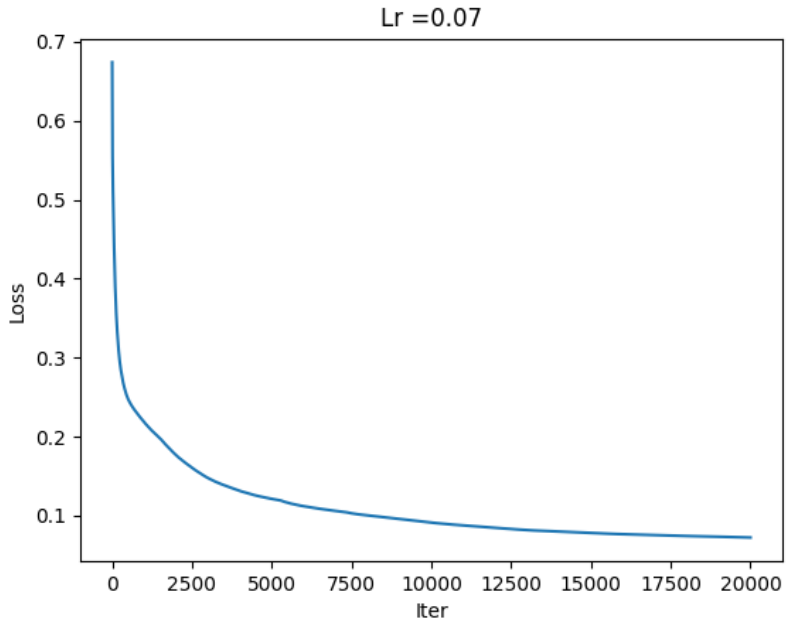


The above graph represents how the generations keep getting fitter. However, the value stagnates at about 0.2

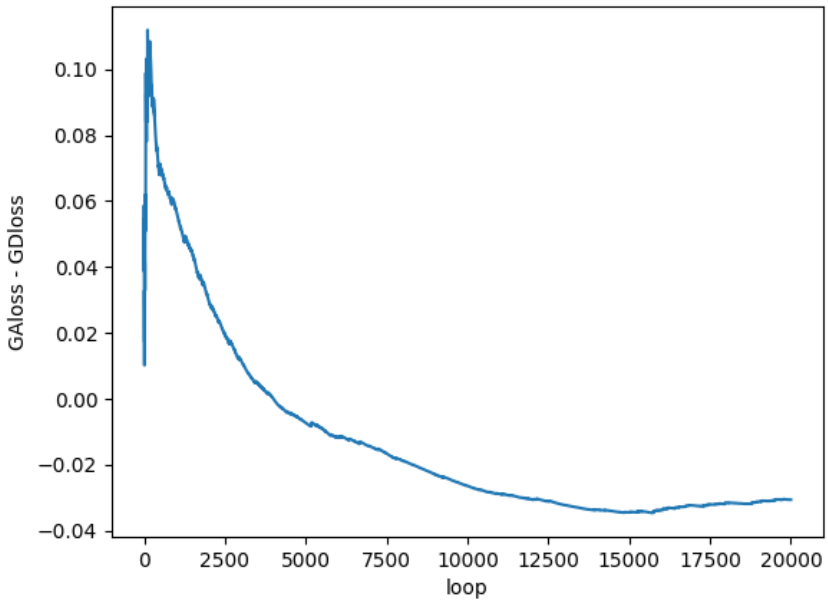
- Task 2



The above graph shows how “loss” decreases as the number of generations increases when using GA to find the weights for the NN.



The above graph shows how “loss” decreases as the number of iterations increases when using GD to find the weights for the NN.



The above graph shows the difference between the performance of GA and GD. The difference in the loss value, GAloss - GDloss, is calculated for both after both have run the NN the same number of times.

3.3 Discussion

- Task 1

The function that the GA was trying to reduce to 0 is a fairly complex and discontinuous function. That's the reason I believe that even the GA's best solutions the function evaluates to 0. Support for that sentiment comes from the fact that the program I've written faces no problem at all in solving simpler polynomial problems, and that too with a very high fitness score.

- Task 2

If I compare the performance of GA and GD by equalizing the number of generations with the number of iterations, then both have very similar performance. Over 20,000 generations, GA could reduce loss to 0.0986 and over 20,000 iterations GD reduced loss to 0.0975

I would consider this impressive result considering that GD works with intrinsic knowledge of how the NN works, whereas GA works without any prior knowledge of how the weights affect loss.

But I see that if I control the number of times the two methods get to run the NN to evaluate a set of weights then GD seems to have a slight edge. This could be due to a range of factors, but perhaps one of the most obvious ones is that I couldn't figure out the optimal values for the parameters of GA—number of solutions in a generation, number of parents, mutation rate.

It is still possible that even after optimizing the parameters it's not possible to make the GA as efficient as GD, but the sheer fact that GA could optimize the NN's weights without knowing anything about the actual connections between the weights and the loss function is quite impressive, and satisfactory.

I think the most immediate improvement which could be made is to track the performance of GA over different values of its parameters to determine their optimum values for the job. Next steps could include trying to optimize the parameters of other AIs using GA to see if I can further optimize AIs.

4. Conclusion

In this project, I've explored Genetic Algorithms and their application in optimizing the weights of a Neural Network. I've shown that GA can perform just as well as some of the more standard training techniques like GD, however parameters of the GA play an essential role in how good the results are.

Perhaps the most exciting aspect of the project has been the feeling of using a learning model to optimize another learning model. And in doing reducing the number of parameters that need to be optimized from hundreds to 3 (since the GA was optimizing a vector of 135 elements, and itself needed only 3 parameters). This suggests that I could perhaps use other AI/ML heuristics together too to improve their performance.

Bibliography

D. S. (2021, June 23). *Introduction to evolutionary algorithms*. Medium. Retrieved from https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac.

Genetic Algorithm adapted from: Gad, A. (2019, August 1). *Artificial neural networks optimization using genetic algorithm with python*. Medium. Retrieved from https://towardsdatascience.com/artificial-neural-networks-optimization-using-genetic-algorithm-with-python-1fe8ed17733e.

Neural Network adapted from: Javismiles. (n.d.). *Deep-learning-predicting-breast-cancer-tumor-malignancy/NN-2L-raw.ipynb at master · javismiles/deep-learning-predicting-breast-cancer-tumor-malignancy*. GitHub. Retrieved from https://github.com/javismiles/Deep-Learning-predicting-breast-cancer-tumor-malignancy/blob/master/nn-2l-raw.ipynb.

Wikimedia Foundation. (2021, October 22). *Genetic algorithm*. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Genetic_algorithm.