APPENDIX A: GENETIC ALGORITHM CODE

```
import model_train
import random as rand
import numpy as np
import csv
import time

Filename = "GA_Results_Revised_AA_10Nm_400rpm.csv"


activation_list = ['tanh', 'linear', 'sigmoid']
initial_Mu_list = [0.00001, 0.0001, 0.001, 0.01, 0.1]
mu_decrease_list = [0.1, 0.15, 0.01, 0.001, 0.0001]
mu_increase_list = [10, 50, 100, 1000, 10000]

def genesis(n_population):
# The genesis function generates the initial population of the genetic algorithm
    population = []
    for i in range(n_population):

        individual = []
        individual.append(rand.choice(activation_list))
        individual.append(rand.randint(1,120))
        individual.append(rand.choice(initial_Mu_list))
        individual.append(rand.choice(mu_decrease_list))
        individual.append(rand.choice(mu_increase_list))
        # Randomized # of neurons for Hidden Layer 1
        individual.append(rand.randint(1, 100))
        # Randomized # of neurons for Hidden Layer 2
        prev = rand.randint(0,100)
        individual.append(prev)
        # Randomized # of neurons for Hidden Layer 3, valid only if layer 2 neuron
count is != 0.
        if(prev == 0):
            individual.append(0)
        else:
            individual.append(rand.randint(0, 100))

        population.append(individual)

    return population
```

```
def fitness_evaluation(chromosome):
# The fitness evaluation function takes an individual, parses through its genes for the hyperparameters then trains and evaluates the NN
# based on the individual's genes.
    act_func =              chromosome[0]
    batch_size =            chromosome[1]
    initial_mu =            chromosome[2]
    mu_decrease_factor =    chromosome[3]
    mu_increase_factor =    chromosome[4]
    layer_1 =               chromosome[5]
    layer_2 =               chromosome[6]
    layer_3 =               chromosome[7]

    accuracy = model_train.evaluate_fitness(act_func, batch_size, initial_mu, mu_decrease_factor, mu_increase_factor, layer_1, layer_2, layer_3)

    return accuracy

def get_fitness_sum(fitness_list):
    sum = 0
    for i in fitness_list:
        sum += i

    return sum

def parent_selection(population_set, fitness_list, k=3):
# Parent selection is a tournament style process. A parent candidate is chosen first, and its performance is compared with another parent candidate.
# The parent candidate with better performance in 2 levels is promoted to the population set of parent candidates.
    parent_candidate = rand.randint(0, (len(population_set)-1))
    for ix in [rand.randint(0, (len(population_set)-1)) for x in range(k-1)]:
        if fitness_list[ix] > fitness_list[parent_candidate]:
            parent_candidate = ix

    return population_set[parent_candidate]

def mate_parents(parent_a, parent_b):
# In mating two individuals, a randomized value determines if a gene would be inherited from Parent A or Parent B
```

```
    offspring = []
    prob = np.random.rand(8)
    for i in range(8):
        if(prob[i] < 0.5):
            offspring.append(parent_a[i])
        else:
            offspring.append(parent_b[i])

    return offspring

def mate_population(parent_population):
# Population Mating iterates over the population for mating
    next_generation = []

    for i in range(0, len(parent_population), 2):
        parent_1, parent_2 = parent_population[i], parent_population[i+1]
        offspring = mate_parents(parent_1, parent_2)
        next_generation.append(offspring)

    return next_generation

def mutate_population(population):
# Population Mutation is set at a 40% mutation rate. If an individual mutates, a
single gene is randomized depending on a randomized setting
    mutated_population = population
    for i in range(len(population)):
        if(rand.random() < 0.4):
            gene_number = rand.randint(0,7)
            if(gene_number == 0):
                mutated_population[i][0] = rand.choice(activation_list)
            elif(gene_number == 1):
                mutated_population[i][1] = rand.randint(40,120)
            elif(gene_number == 2):
                mutated_population[i][2] = rand.choice(initial_Mu_list)
            elif(gene_number == 3):
                mutated_population[i][3] = rand.choice(mu_decrease_list)
            elif(gene_number == 4):
                mutated_population[i][4] = rand.choice(mu_increase_list)
            elif(gene_number == 5):
                mutated_population[i][5] = rand.randint(1,100)
            elif(gene_number == 6):
```

```
                if(mutated_population[i][7] == 0):
                    mutated_population[i][6] = rand.randint(0,100)
                else:
                    mutated_population[i][6] = rand.randint(1,100)
            elif(gene_number == 7):
                if(mutated_population[i][6] == 0):
                    mutated_population[i][7] = 0
                else:
                    mutated_population[i][7] = rand.randint(0,100)

    return mutated_population


f = open(Filename, "w")
writer = csv.writer(f)
start_time = time.time()
# Generate initial Population
population = genesis(100)
try:
    alpha, alpha_score = 0, fitness_evaluation(population[0])
except:
    alpha, alpha_score = 0, 0
data = []
#Run Algorithm for 30 generations
for i in range(30):
    data = []
    print("Generation number: %i" %i)
    scores = []
    for individual in range(len(population)):
        data = []
        data.append(str(i))
        print("Generation number : %i" %i)
        print("Individual number: %i" %individual)

        data.append(str(individual))

        for x in population[individual]:
            data.append(str(x))
        try:
            print(population[individual])
            #Determine Fitness of an individual
            scores.append(fitness_evaluation(population[individual]))
```

```
        except:
            #Error exception for hyperparameters that doesn't allow a network
training to converge
            print("\n")
            print("Error Found")
            scores.append(0)

        data.append(str(scores[-1]))
        writer.writerow(data)
    #Determine if the generation produced a better performing individual
    for scores_index in range(len(population)):
        if(scores[scores_index] > alpha_score):
            alpha, alpha_score = population[scores_index], scores[scores_index]
            print(alpha, alpha_score)
    #Generate the parent population
    parent_population = [parent_selection(population, scores) for _ in
range(len(population)*2)]
    #Run population through Mutation Process
    next_gen = mutate_population(mate_population(parent_population))
    population = next_gen

print("Best config: ", alpha)

writer.writerow(["Best Config " , str(alpha)])

print("Best Accuracy: ", alpha_score)

writer.writerow(["Best ACC: " , str(alpha_score)])

total_time = time.time() - start_time
print("Total Time: %f" %total_time)
writer.writerow(["Time: ", str(total_time)])
f.close()
```