# Applications of Genetic Algorithms And Reinforcement Learning in Neural Networks

https://github.com/michaeldlee23/cs390-project

Shubham Jain, Michael Lee, Jeremy Meyer

December 2, 2020

## 1 Project Overview

## 2 Genetic Algorithms

Genetic Algorithms (GA) are are type of learning algorithm founded on the idea that crossing over the features of two models should result in a better model. In essence, GAs aim to mimic evolution through natural selection — given a population, determine which individuals are the "fittest," then allow them to "mate" and produce hopefully fitter offspring. After many generations, we should end up with a well-trained and competent model.

### 2.1 Procedure for Neural Networks

GAs are applicable to a large variety of problems, typically for search and optimization. They can be used to learn the parameters of a regression model, the structure of a decision tree, and even the architecture of a neural network. In the first phase of our project, we use a GA in place of typical weight optimizers like SGD or Adam to tune the weights of each layer of a neural network used for MNIST digit classification. While GA proved to be suboptimal when compared to the standard methods of backpropagation, we were nonetheless able to obtain somewhat satisfactory results, achieving around 70% accuracy.

A general overview of our pipeline is as follows:

1. Construct an initial population of networks with randomly initialized weights and biases.
2. Use the training data to immediately make predictions using each network in the population.
3. Rate each population's fitness using its training accuracy.
4. Select the top individuals of this population to create a mating pool.
5. Randomly select parents from this mating pool to perform crossover and produce sets of weights and biases for the next generation of networks.
6. For each new network, randomly select weights and biases to mutate.
7. Repeat steps 2-6 for a large number of generations.

## 2.2 Hyperparameters

We have the following hyperparameters:

1. `NUM_GENERATIONS`: the number of generations to continue this process for
2. `POPULATION_SIZE`: the number of networks in each population
3. `MUTATION_RATE`: the probability that a weight is mutated during crossover
4. `MUTATION_WEIGHT_RANGE`: the range of values that a weight could be modified by
5. `MUTATION_BIAS_RANGE`: the range of values that a bias could be modified by.

We use uniform crossover to produce offspring. That is, we randomly select weights from each parent to use in the offspring with equal probability. We also use uniform additive mutation, so each weight has a chance to mutate with uniform probability, and each mutated weight changes by adding some random value to it.

We performed this process on both a single-layer feed-forward artificial neural network as well as a convolutional neural network.

**ANN Network Architecture**

```
Model: "sequential_ANN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_0 (Dense)              (None, 784)               615440
_________________________________________________________________
dense_1 (Dense)              (None, 32)                25120
_________________________________________________________________
dense_2 (Dense)              (None, 10)                330
=================================================================
Total params: 640,890
Trainable params: 640,890
Non-trainable params: 0
_________________________________________________________________
```

**CNN Network Architecture**

```
Model: "sequential_CNN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 28, 28, 8)         40
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 16)        528
_________________________________________________________________
flatten_1 (Flatten)          (None, 12544)             0
_________________________________________________________________
dense (Dense)                (None, 10)                125450
=================================================================
Total params: 126,018
Trainable params: 126,018
Non-trainable params: 0
_________________________________________________________________
```

## 2.3 Results

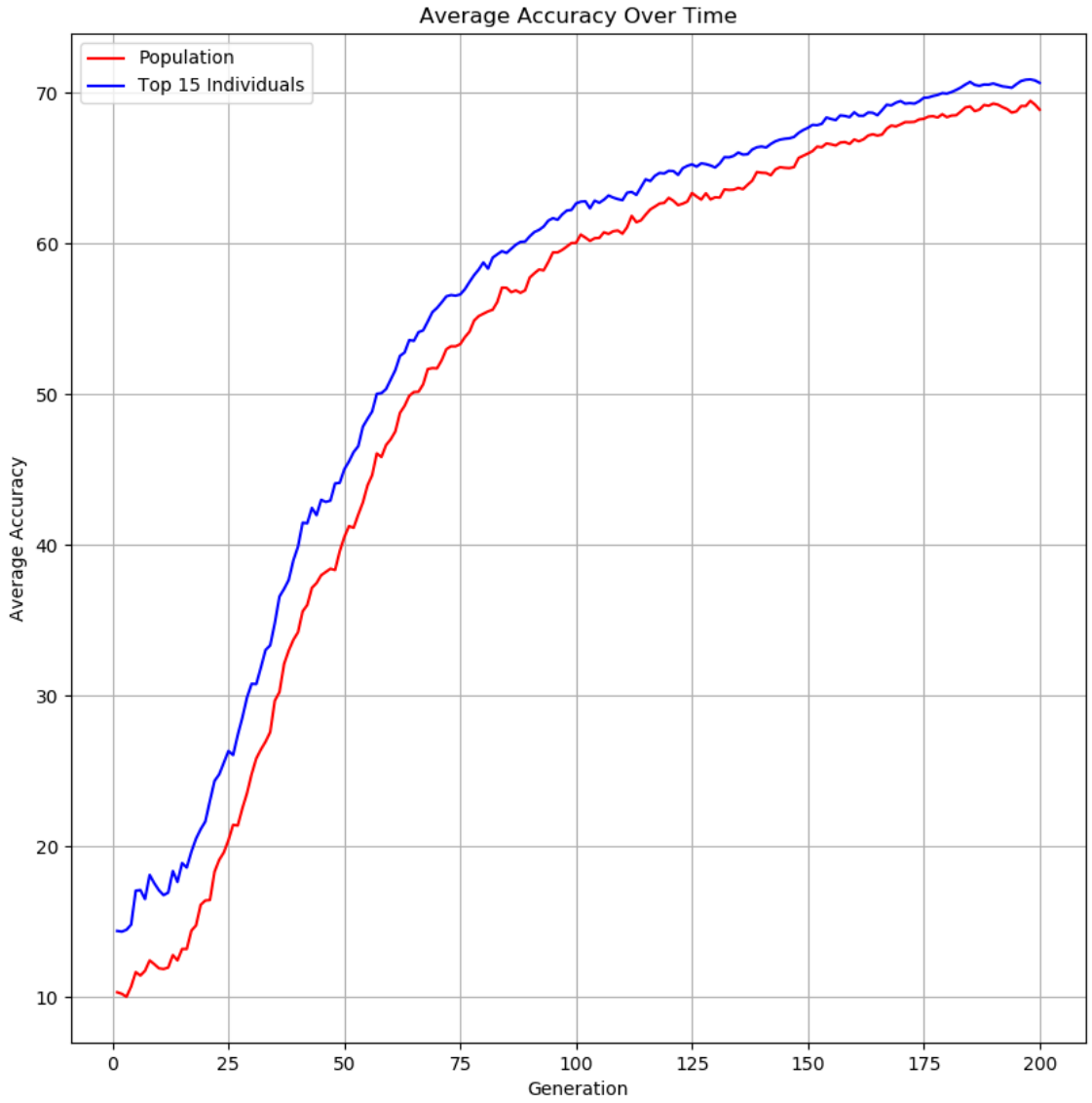



Figure 1: Learning Curve for ANN over 200 Generations

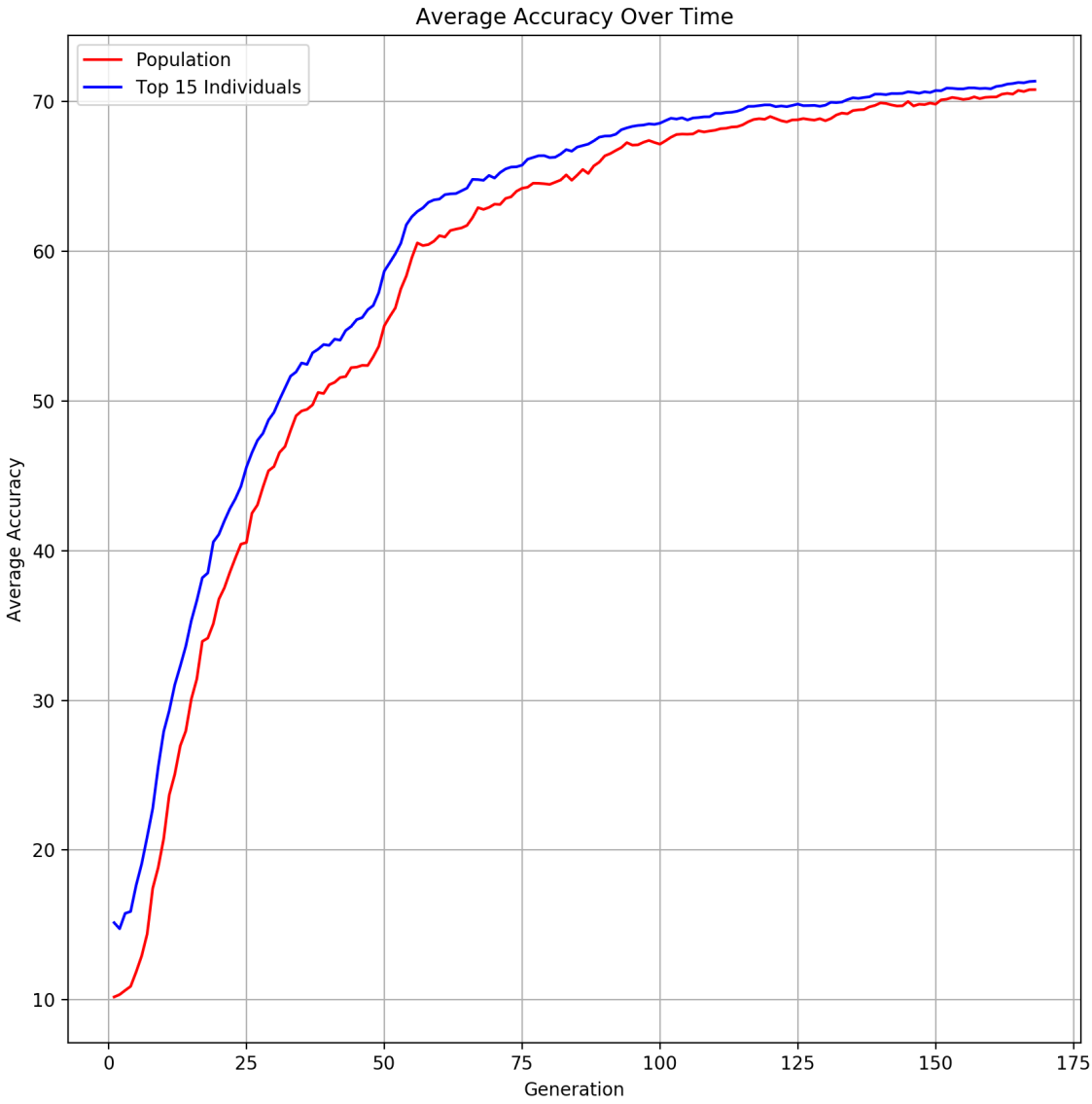



Figure 2: Learning Curve for CNN over 171 Generations*

*Due to connection issues, the training of the CNN was ended prematurely and did not reach the full 200 generations. However, it appears to have been converging anyways.

As we can see, both classifiers reached around 70% accuracy by the end of the training process. The CNN converged much faster than the ANN.

# 3 Deep Q Networks

Deep Q Networks (henceforth referred to as DQNs) are a type of reinforcement learning algorithm that try to encode and loosely approximate the data that would be stores in a Q-learning table as a neural network. In depth information about DQNs can be found at https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4#:~:text=DQN%20updates%20the%20Q%2Dvalue,to%20have%20a%20larger%20impact.

## 3.1 OpenAI Gym - Testing Environment

OpenAI's Gym package provides an ideal testing environment for any kind of reinforcement learning application. For this project, we chose Gym's Pole Cart v0 environment to train our DQN against. Gym describes the problem as follows:

"A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center."

Each iteration of the pole-cart environment (called a step) adds to the total score of that run, which means that the score an agent acquires on a run is directly proportional to how long it can balance the pole. Gym considers the pole-cart environment as "solved" if an agent can achieve an average score of 195 or higher over 100 consecutive runs. In both reinforcement learning algorithms demonstrated in this paper, this requirement is used as the metric for determining success.

## 3.2 Hyperparameters and Network Architecture

Our DQN implementation has the following hyperparameters:

1. `LEARNING_RATE`: the rate at which the algorithm improves every step
2. `EXPLORATION_RATE`: the rate at which the algorithm explores alternates for the next action
3. `EXPLORATION_DECAY`: the rate at which the exploration rate is stepped down as the algorithm runs
4. `MEMORY_SIZE`: the size of the memory buffers used to keep track of previous actions and rewards

The following neural network was used to solve this instance of OpenAI Gym's Pole Cart v0 environment:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 24)                72
_________________________________________________________________
dense_1 (Dense)              (None, 24)                600
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 75
=================================================================
Total params: 747
Trainable params: 747
Non-trainable params: 0
_________________________________________________________________
```

### 3.3 Results

The DQN trained for 500 games, with each game ending if either the environment returned a fail or if the network managed a score of 500 (i.e. the max score).

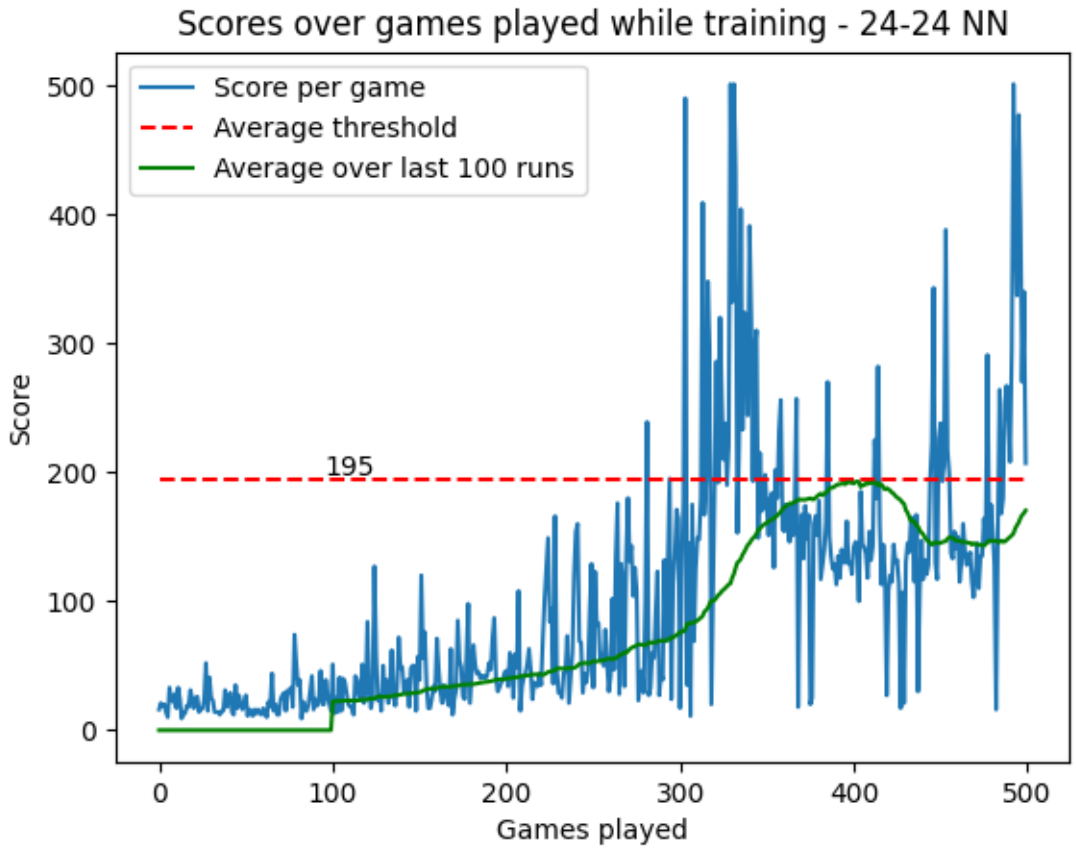



Figure 3: Scores per game played during training over 500 games

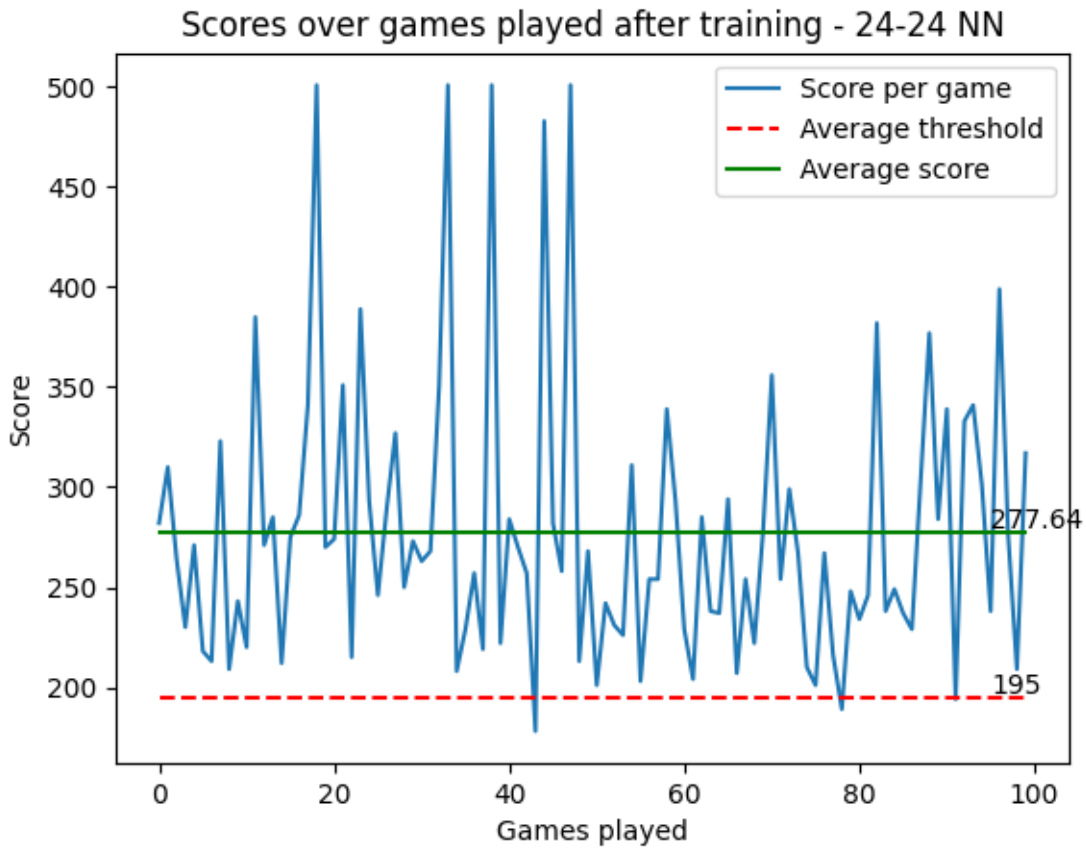



Figure 4: Scores per game played during testing over 100 runs

# 4 Policy Gradient

Policy gradient algorithms are reinforcement learning algorithms that directly optimize the policy, compared to Deep Q-Learning which first approximates the value function which then informs the policy. Instead, in the stochastic policy gradient model in this project a distribution of how likely the policy is to take a certain action is output, which is then used to select the action to take.

## 4.1 Testing Environment

All testing was done in OpenAI's PoleCart-v0 environment, as described in section 3.1

## 4.2 Hyperparameters and Network Architecture

The network architecture are the same between DQN iterations and policy gradient iterations as described in 3.2. Training was done for 500 epochs with a maximum of 200 steps before exiting. The ADAM optimizer was used, with a learning rate of 0.003 and initial gamma value of 0.99.

## 4.3 Results

The policy gradient implementation proved to be much more robust in terms of convergence than the DQN version. A model that converges was easy to find given a pool of ten trained algorithms. Due to the probabilistic nature of the algorithm (the policy only outputs a distribution of how likely an action is to be the better action) it was entirely possible for the algorithm to search in the completely wrong direction for the optimal policy, or during a testing run choose multiple actions in a row that would lead to failure. This chance would likely be reduced as the algorithm trains for longer and fully learns that the policy should not consider those actions at all.

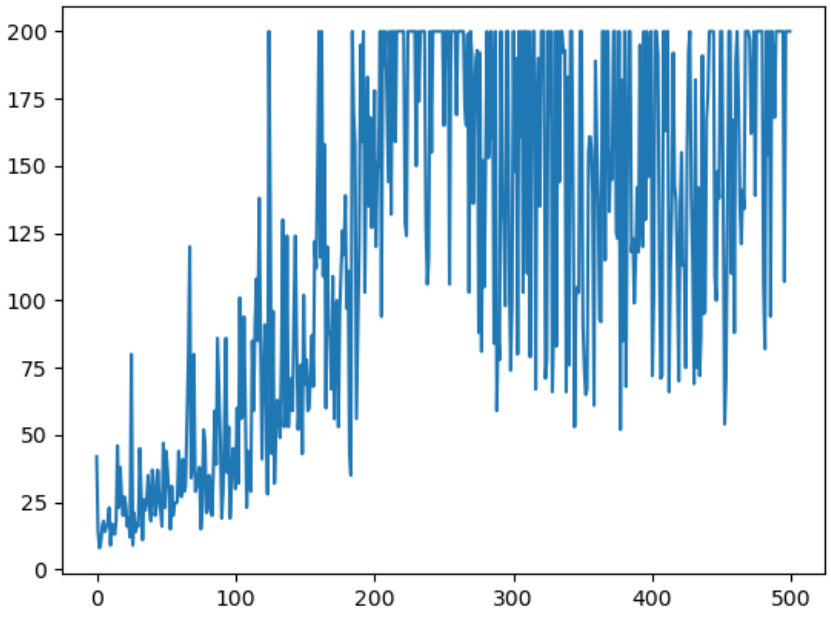
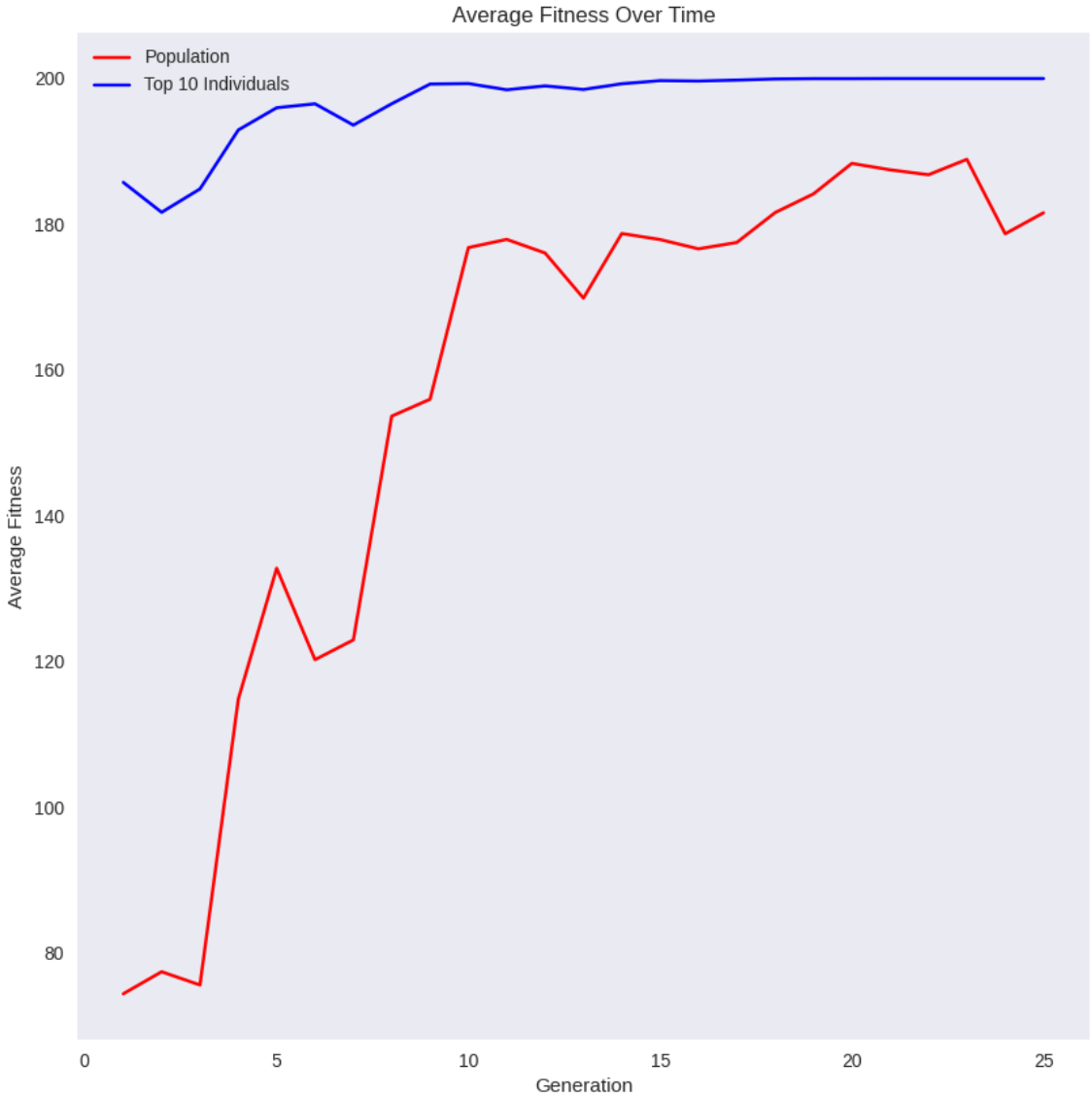



Figure 5: Example training run capped at 500 timesteps


Figure 6: score of example policy over 20 runs

# 5 Genetic Algorithm with Policy Gradient




At the end of the project we tried to use the genetic algorithm to improve the convergence consistency of the policy gradient RL networks. Instead of randomly seeding the networks for each run, we use a GA to determine what a better initial seed weight would be for the network. We can see that the average score for the networks was much more consistent as the GA progressed through the generations.

# 6 Resources

- Introduction to Genetic Algorithms and General Intuition
  - https://towardsdatascience.com/using-genetic-algorithms-to-train-neural-networks-b5ffe0d51321
  - https://theailearner.com/2018/11/08/genetic-algorithm-and-its-usage-in-neural-network/
  - https://www.youtube.com/watch?v=XP8R0yzAbdo
- Crossover Methods
  - https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/
- Deep Q Learning Networks
  - https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4#:~:text=DQN%20updates%20the%20Q%2Dvalue,to%20have%20a%20larger%20impact.
  - https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c
- Policy Gradient algorithms
  - https://homes.cs.washington.edu/~todorov/courses/amath579/reading/PolicyGradient.pdf
  - https://medium.com/@ts1829/policy-gradient-reinforcement-learning-in-pytorch-df1383ea0baf