CS 169
Final Project Report

Luke Keating, An Le, Son Le, David Legg

UCI FALL 2018

Problem Statement

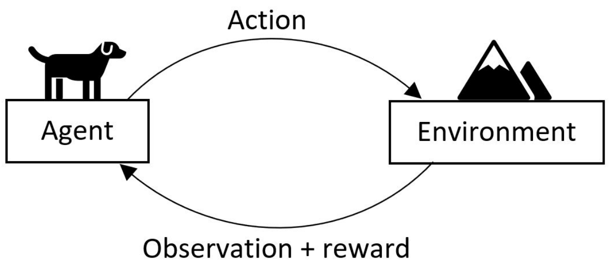
When we first formed groups, we decided that we were interested in using a genetic algorithm in training a population. We decided to focus on optimizing the performance of something that we could watch (we wanted to have fun watching our algorithm fail). We were inspired by Google's deepmind algorithm and wanted to take on a similar task. After further discussion (and inspiration from deepmind's training of a bipedal model to walk), we chose to take on a similar problem and train a half-cheetah model to run. We decided on the half-cheetah to both have fun but to also get a deeper understanding of the genetic algorithm.

Approach

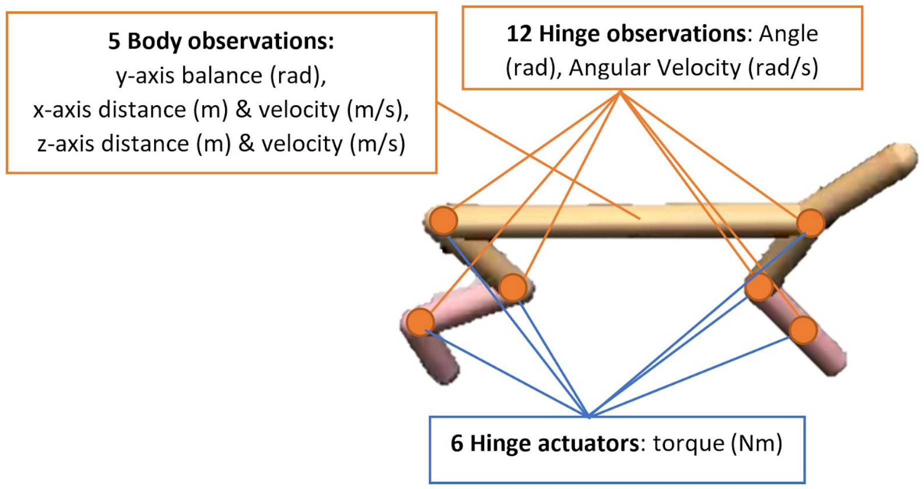
a. OpenAI Gym with Half-Cheetah model

OpenAI Gym is a python library which includes several environments to use in the training of a learning algorithm. These environments includes several tasks such as balancing a pole on a moving cart and several atari games along with several tasks requiring the physics engine mujoco, including the half-cheetah model. We agreed upon using the gym library in order to focus more resources on coding the genetic algorithm and training a model rather than coding of an environment.

The first step was to build an understanding on how the gym library worked. We tested out different parts of the code any realized that it was broken down into a few key parts. Gym first loads an environment and provides an observation of that environment as a vector of data. An actor must then take a step in that environment and a new observation is provided as well as a score associated with the step that the actor took. This process is repeated until either the actor fails or the maximum number of steps is reached.



We then focused specifically on the half-cheetah model. The half-cheetah environment produces an observation space vector with a shape of (17,1) and takes an action input as a vector with the shape (6,1). The breakdown of these vectors is shown in the figure below. The half-cheetah is rewarded for how fast it travels in its environment. We initialized the cheetah using a population of random actors which took an observation and performed a random step based on that observation. These actors tended to only last for a couple of steps as they did not have a good understanding of the environment yet.


5 Body observations:
y-axis balance (rad),
x-axis distance (m) & velocity (m/s),
z-axis distance (m) & velocity (m/s)
12 Hinge observations: Angle (rad), Angular Velocity (rad/s)
6 Hinge actuators: torque (Nm)

b. Neural Network

Given the cheetah's observations, our problem is to build a function to produce actions. We decided upon using a simple feed-forward neural network in order to receive 17 inputs and make 6 regression outputs. Our network takes observations as the input layer, feeds it to various hidden layers, and produces a final action as an output. Each hidden layer has some neurons. Each neuron is associated with a bias and a set of weights.

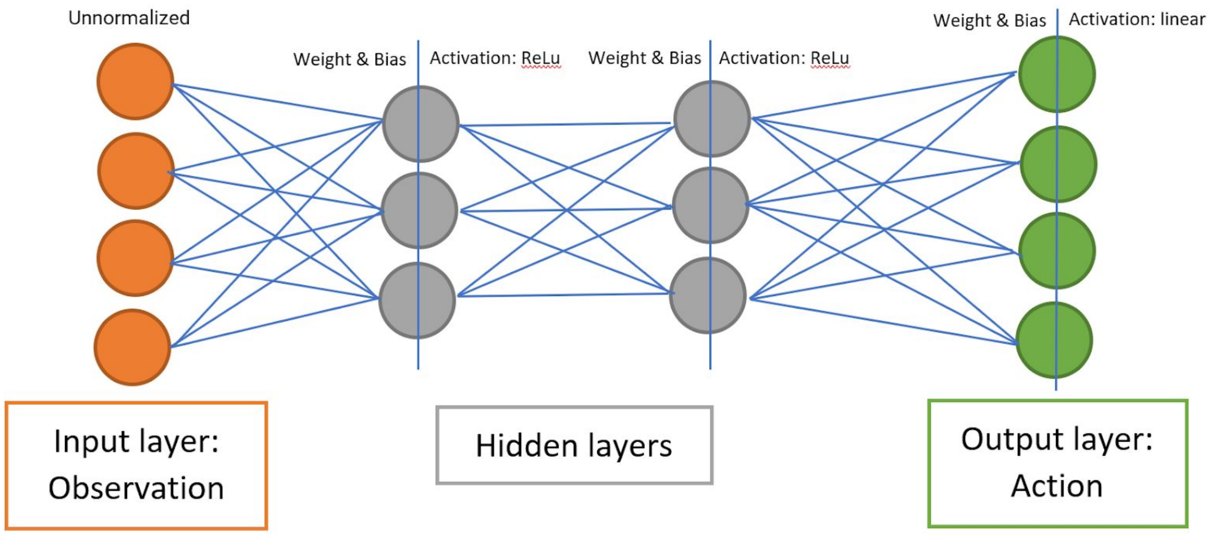
Layer H is calculated using the below formula, where W is the set of weights, b is the bias, and x is activated values of neurons in the previous layer.

$$H = (w_1 \cdot x \ldots w_n \cdot x)^T + b = Wx + b$$

Afterward, the neuron value is activated using reLu function. We chose reLu function mainly because it runs fast and thus, reduces our testing time. The activated value ha_i will then be used to compute the next layer.

$$ha_i = reLu(h_i) = \begin{cases} 0 \; if \; h_i < 0 \\ h_i \; if \; h_i \geq 0 \end{cases}$$

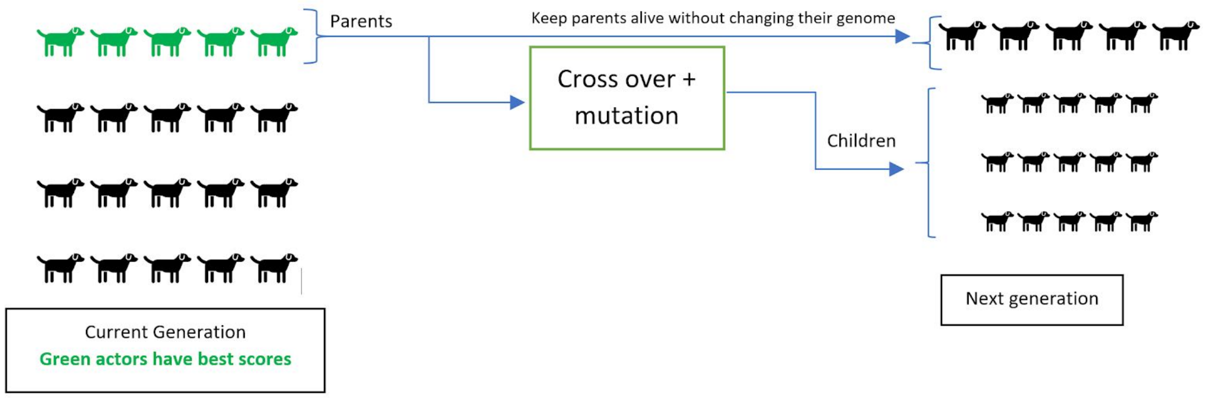
The input layer and output layer is not normalized because the observation space and action space of the half-cheetah model are unbounded.



c. Genetics Algorithm

In order to implement the genetic algorithms, we defined a "genome" for each actor as a list of floating-point numbers which represents that actor's neural network. If the weight matrices are $W_1 \ldots W_N$, with sizes $n_j \times m_j$ for $j = 1, \ldots, N$, and the bias vectors are $B_1 \ldots B_N$, we convert each one to row-major order, and concatenate the results, like this:

$$[W_1]_{1,1} \ [W_1]_{1,2} \cdots [W_1]_{1,m_1} \ [W_1]_{2,1} \cdots [W_N]_{n_N,m_N} \ [B_1]_1 \cdots [B_N]_{m_j}$$


Parents
Keep parents alive without changing their genome
Cross over + mutation
Children
Next generation
Current Generation
Green actors have best scores

We start out by generating random genomes for each member of the initial population, and reconstructing the neural networks from these genomes. We "test" these actors by simulating them in the environment (running the observation/action loop) for 1000 iterations, accumulating the reward at each step. Each actor is associated with the total score it earned in this simulation. Then, the actors which scored in the top 20% (or other value, if specified) of the population are preserved as the mating pool, and the rest are discarded. 80% of the new population is created by performing cross-over and mutation on randomly selected pairs from the mating pool, while the other 20% is the mating pool, unmodified. Cross-over is performed at a point selected uniformly at random in the genome. Each element of the child genome is then mutated with independent probability, usually between 5% and 20%. Elements are mutated by choosing a new floating point value from a normal distribution centered on its current value.

We found that using this method for creating the mating pool, which we call “Top Selection”, allowed us to mutate the offspring more aggressively while minimizing the risk of losing good configurations to bad mutations, since the best configurations were always preserved. We also tried a roulette selection procedure, as described by Belegundu and Chandrupatla in *Optimization Concepts and Applications in Engineering*, in which members of the population were selected preferentially according to their score, but had little success with it.

Responsibilities

We delegated tasks as follows. Each team member also contributed to the report, primarily in those sections most directly related to their tasks.

1. Luke Keating: Selection of environment, brainstorming actor design
2. David Legg: Initial coding of actors, genetics, and testing harness.
3. An Le: Improvements to actors, selection routines; execution of tests.
4. Son Le: Improvements to actors; graphics, video and presentation design.

Coding Experience

During the implementation, we have faced and solved several problems. First, we solve slow training time by using parallel processing provided via joblib’s parallel computing tools in Python. We used the multiprocessing library to detect how many CPU’s were available and then ran parallel processing on each of those CPU’s. This allowed our algorithm to run approximately *n* times as fast, with *n* being the amount of CPU’s on a given computer.

When we first started training, we did not have a lot of changeable parameters in order to make our code work smoothly. For example, when we started we used a single layer perceptron in order to perform regression on the observation data. This did not provide us with the accuracy which we needed so we decided to create a neural network with the ability to change the number of hidden layers and the nodes in each layer which we were working with. We also encountered several issues with the genetic algorithm, so we added controls for several key parameters in the evolution and simulation procedures, including the number of generations, simulation repetitions, probability of mutation, and whether to keep parents. The generations parameter controlled how many generations of the genetic algorithm we would run. We added this in order to monitor our algorithm and verify that it was running correctly. The simulation repetitions controlled how many times each actor would be run in a generation. We did this in order to verify that an actor was actually good by making it run multiple times, since the environments involve a small amount of inherent randomness. This prevented a performance anomaly from infiltrating the gene pool. We added a parameter to change the probability of mutation so that we could make adjustments if we were having trouble training a population or just wanted to try to optimize the time it took to train a population to an acceptable level. We added the option to keep parents alive to avoid a complete regression of the following generations due to a bad mutation or crossover. We also began to save our best actors and their scores to a file so that we could go back and create a new population based upon our best actors from past populations. As we added more parameters of control for both our neural network and our genetic algorithm, we had an easier time getting better results.

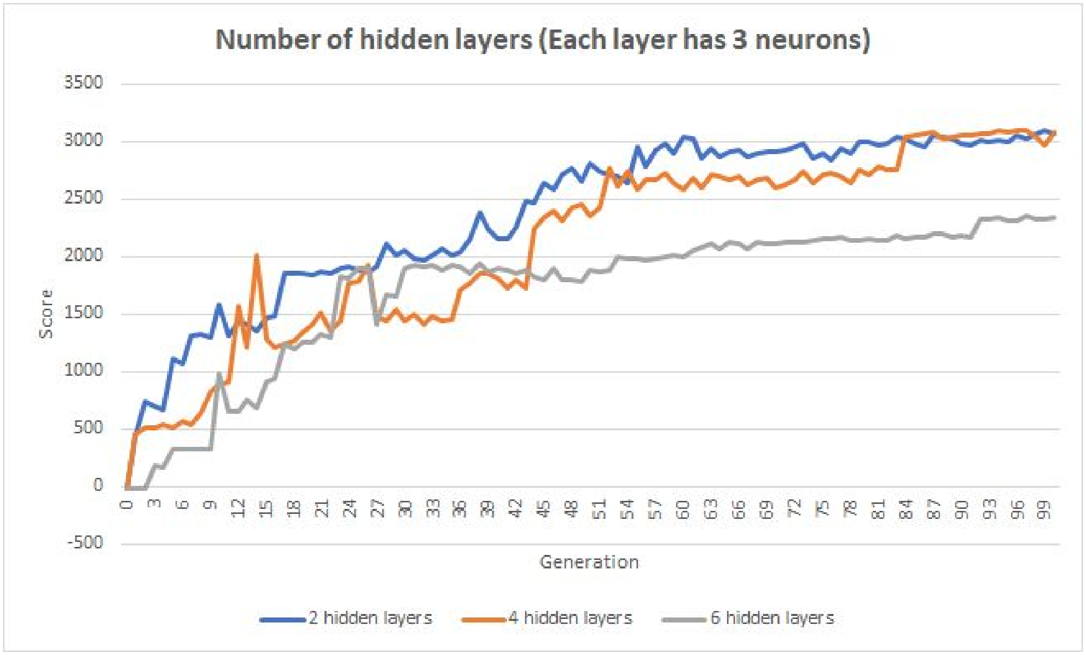
Lastly, we provided parameters in order to make the evolution process a more entertaining experience. The reason why we approached this problem in the first place was to have an enjoyable experience in teaching the half-cheetah to run. To do this we added a parameter so we could render the best actor from every *n*th generation. We also added a feature which allowed us to go back into our files and render the best actor from any of our past generations. These parameters allowed us to watch our cheetah learn and grow with the genetic algorithm.

Results

a. Parameter Comparison

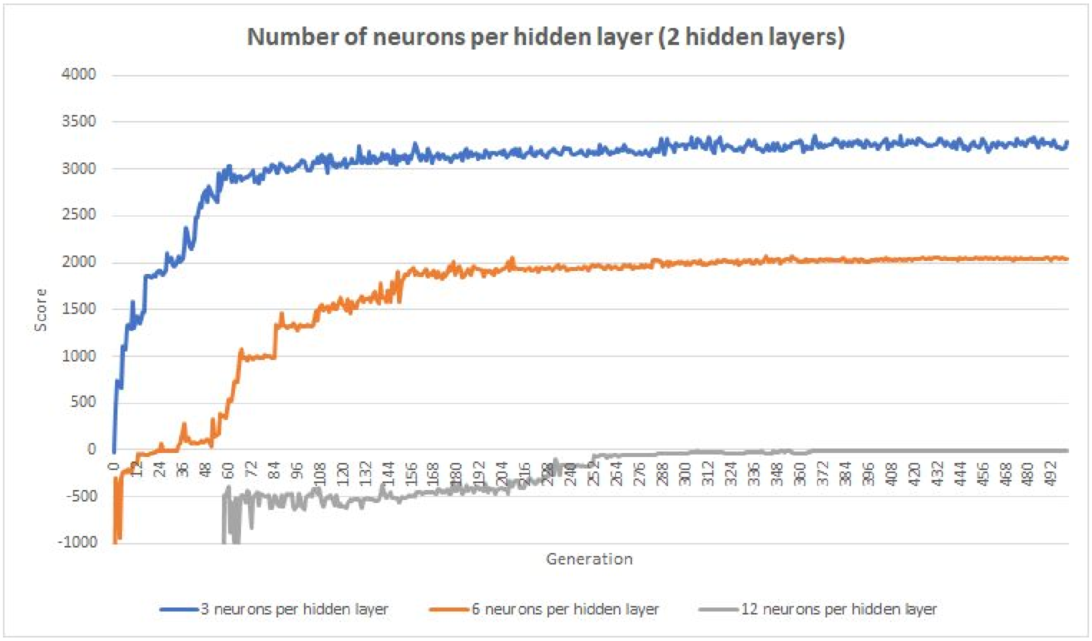
To determine which combination of parameters works best for our neural network input and output, we tried different configurations and obtained the following comparisons.

1. Number of hidden layers



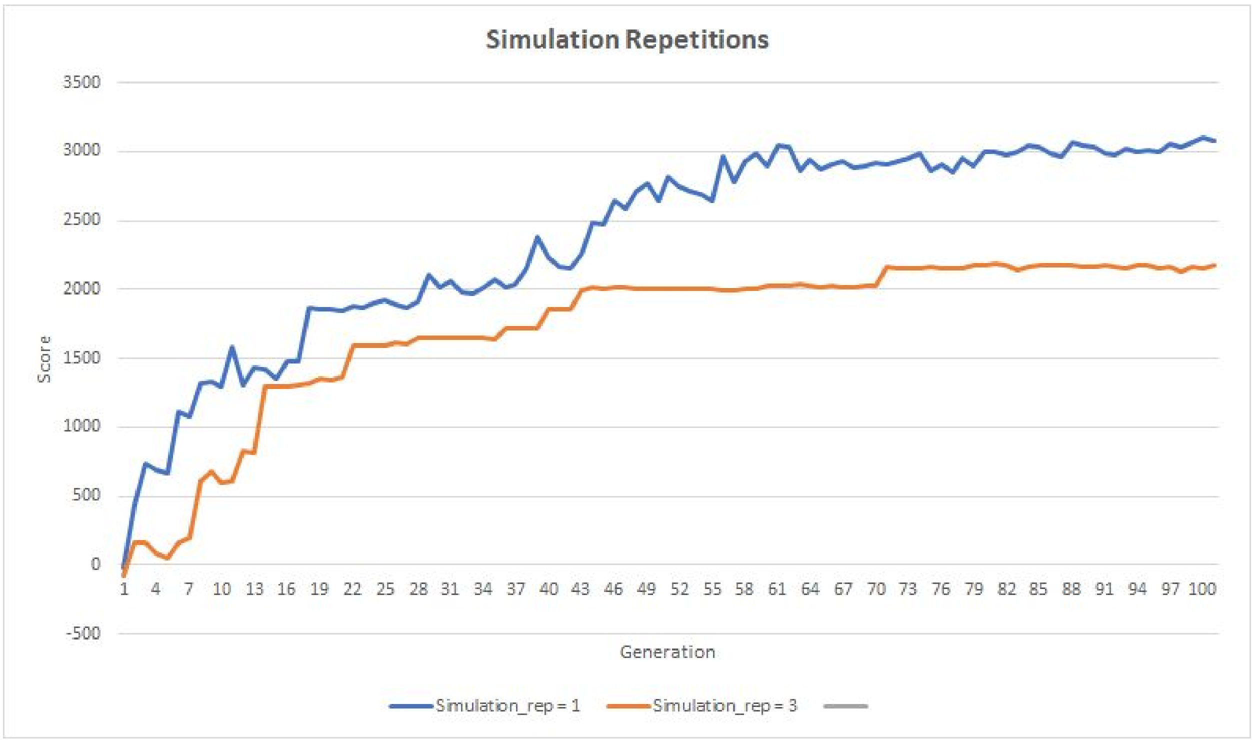
The 2x3 neural network provided us with the most stable performance. The 4x3 neural network gave us a high score but was prone to drastic changes though the genetic process and was more computationally expensive.

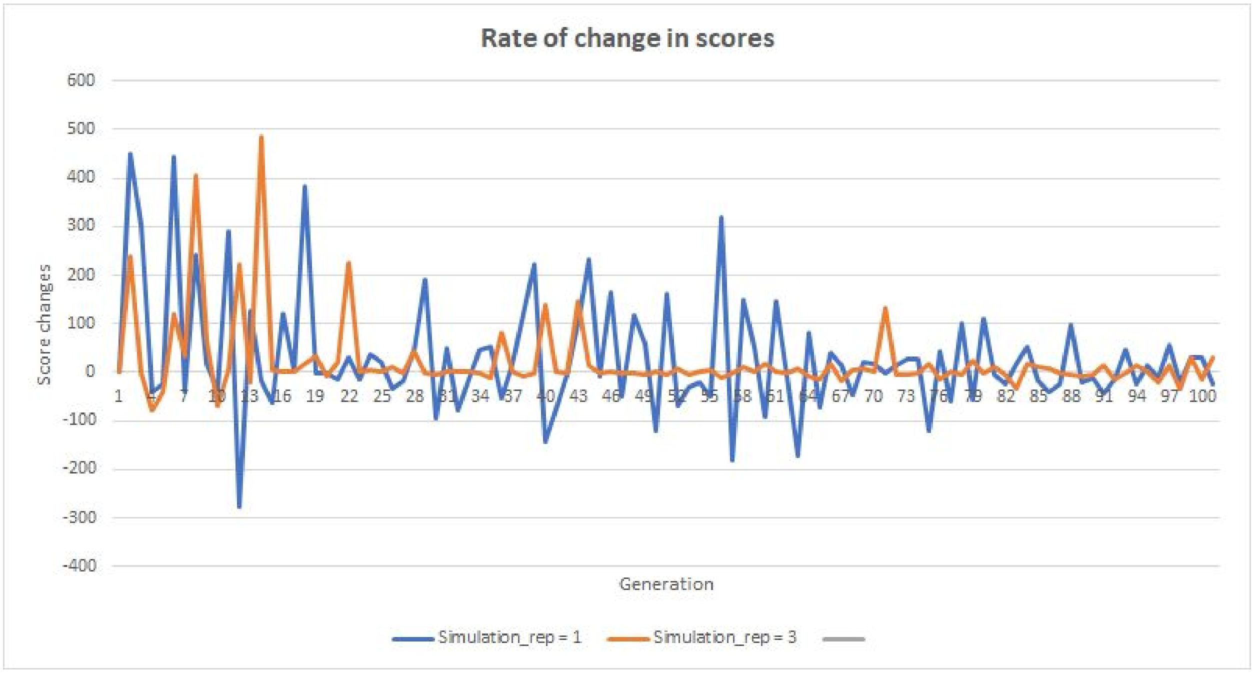
2. Number of hidden nodes per hidden layer



Having just 3 neurons per hidden layer gave us the best performance on the data and also ran faster than having more neurons in each hidden layer.

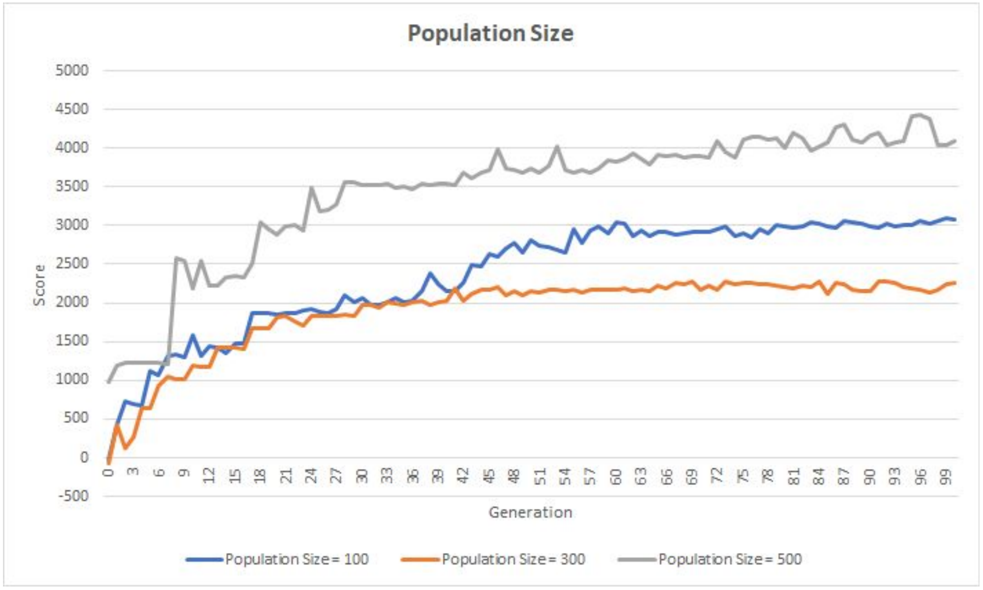
3. Simulation Repetitions





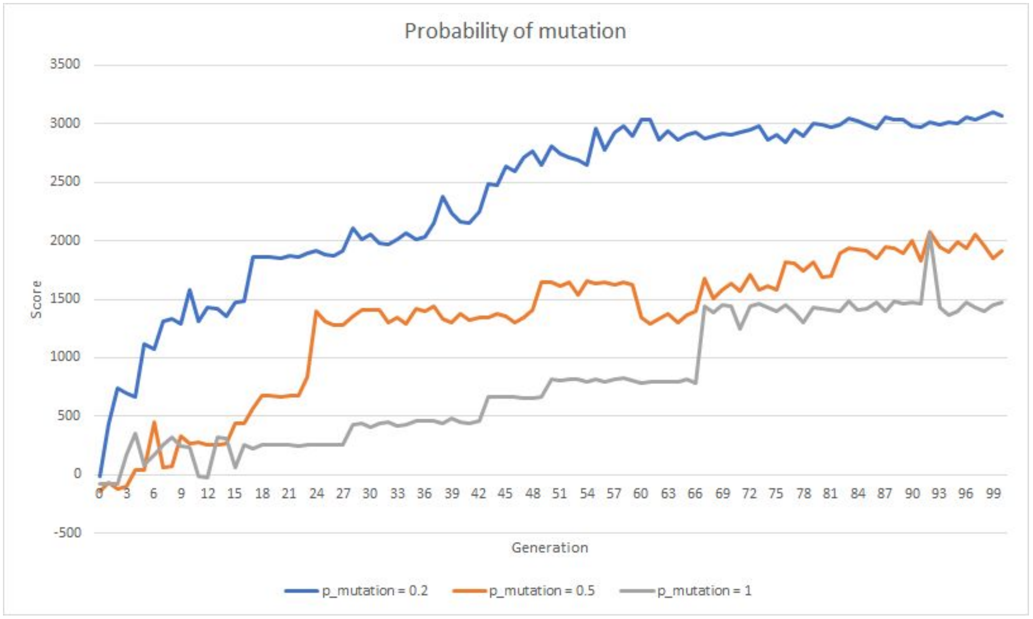
Having just 1 simutation rep gave us the best performance and was reduced the amount of time needed in training.

4. Population



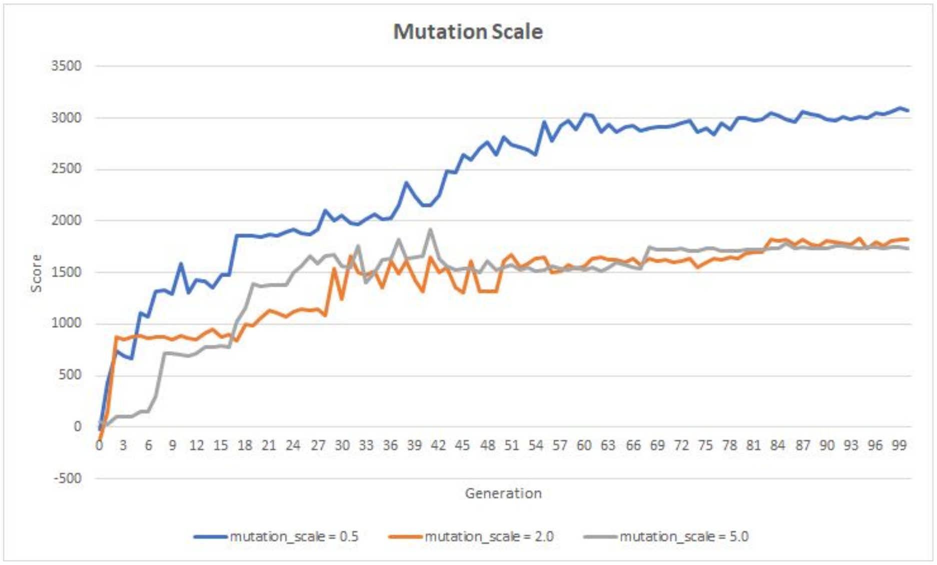
Having a population size of 500 instead of just 100 resulted in quicker evolution and higher scores by generation 100, most likely due to a larger gene pool to influence.

5. P_mutation



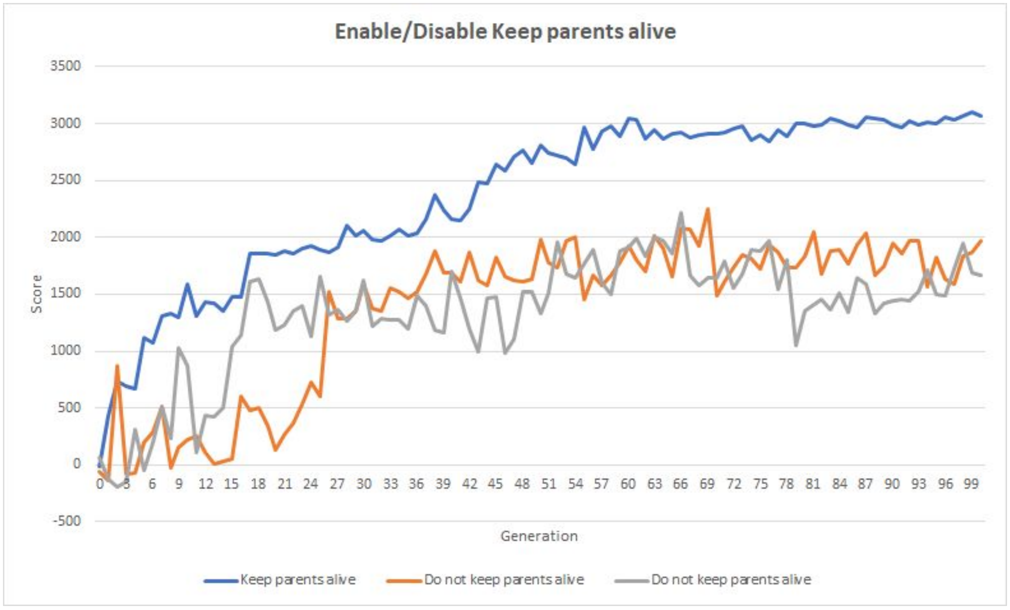
Having too large of a mutation rate caused too many changes in each generation which reduced the overall performance of future generations.

6. Mutation_scale



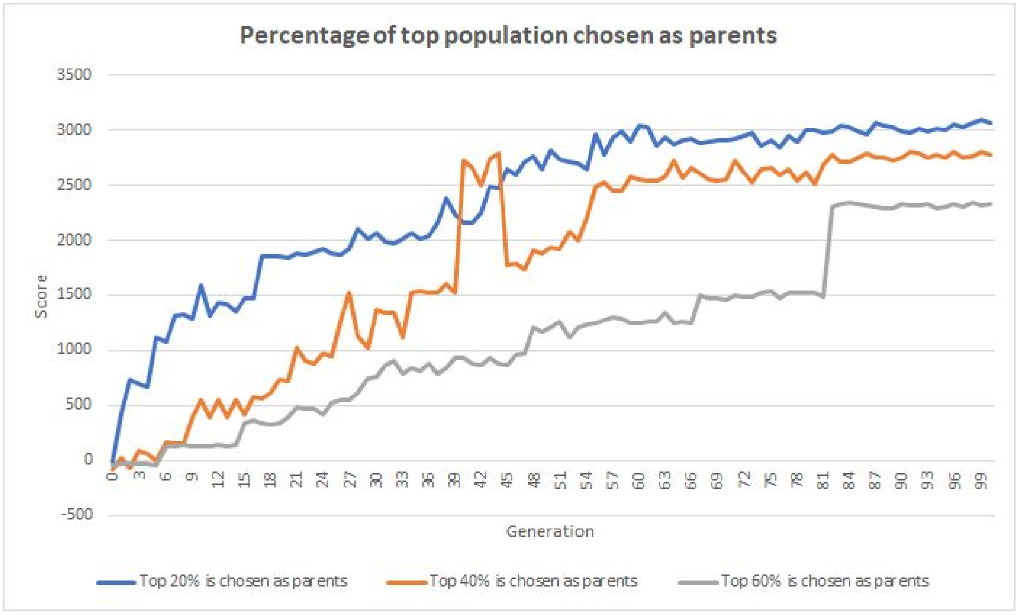
Mutation_scale controls the width of the normal distribution that mutations are chosen from. Having larger values caused more variance in the population, as expected. However, even mild values for mutation_scale allowed for fast training.

7. Keep_parents_alive



Keeping the parents alive in the next generation forced the algorithm to keep the best performers until an improvement in performance was gained.

8. Cutoff



Keeping only the top 20% of parents alive helped ensure that the mating pool was not watered down by some less than ideal actors.

b. Final Result Analysis

After testing out each parameter, we decided that our best combination of parameters to be population size 500, choose top 20% of parents to create a new population, keep parents alive in future generations, 20% probability of mutation, .5 mutation scale, 1 simulation rep, 2 hidden layers, and 3 neurons per hidden layer. Having a large population gives you a higher probability of finding a good actor. Having a 20% mutation rate is a good middle ground for both fast mutation as well not making too large of a change to each genome. Keeping the parents alive helped ensure that the model did not regress due to an less than ideal offspring generation. Keeping a lower mutation scale made sure that the offspring genomes were different but not too different from the parent genomes. Actors did not seem to have to varying of scores when running multiple simulation reps so having just 1 simulation rep helped increase the speed of the evolution process and did not affect the performances of the actors. Through testing we found that a 2x3 neural network performed the best regression on the data.

We found the results of a behavioral cloning and DAgger algorithms run by a graduate student at Berkeley (Holly Grimm) and decided to compare our results to theirs. With our genetic algorithm, we were able to score almost 1,000 points higher than what their algorithms were able to produce. Our peers in the class also ran PPO on the half-cheetah environment and were able to reach an average score of around 5,000 within 30 minutes (Lanier and Nagata). Our genetic algorithm was able to receive a similar average score within a similar amount of time. It should be noted that we streamlined our algorithm by running 12 tests in parallel, effectively cutting down the time required to run the algorithm and receive acceptable results.

Conclusions

In comparison to outside sources, it seems as though our genetic algorithm was able to get good results. With our model, we were able to get a good performing cheetah by 100 generations and an elite performing cheetah by 175 generations. One of the advantages of our approach was that the genetic algorithm has many hyperparameters that can be changed in order to improve the optimization process. One of the biggest disadvantages is the training time, though this can be managed with parallel execution. Each generation of our optimal algorithm required 500 cheetahs to be run over several hundred generations, which takes considerable computational time and resources. An interesting follow up to this project would be to compare this approach to more traditional neural network training methods.

Through this process we learned about the robustness and versatility of the genetic learning algorithm. We found that genetic algorithm optimization produced at least passable results under a wide variety of hyper-parameter choices. The genetic algorithm can be altered in many ways to fit a broad set of problems. This was especially obvious when walking through the poster session for our final. The genetic algorithm was implemented in many different problems with many varying hyperparameters, with great success. In addition to following up this project with a comparison against other optimization algorithms, it would be interesting to compare the breadth of problems that genetic algorithms can solve compared to other general-purpose optimization methods. We look forward to potentially doing such comparisons ourselves.

Instructions to test (to include in code submission)

First, install Gym and MuJoCo according to their installation instructions, found at https://github.com/openai/gym#installation and https://github.com/openai/mujoco-py#install-mujoco.

Next, optionally install joblib using "pip install joblib==0.12". **Do not install joblib 0.13, as it causes a memory leak which can crash the computer.** Installing joblib lets the algorithm train in parallel, dramatically accelerating evolution, however, it is not necessary to running the algorithm.

To run the basic example, run the script "example_mujo_nn.py". There are also changeable parameters:
actor_args: takes a list of integers with each integer representing the nodes in each hidden layer
evolve_args: allows changes to the genetic algorithm parameters
render_args: control how fast and how long a simulation will be rendered to view.

Bibliography

Activation function examples, https://en.wikipedia.org/wiki/Activation_function accessed on 11/22/18

Belegundu and Chandrupatla, *Optimization Concepts and Applications in Engineering*, 2nd ed., Cambridge University Press, 1999

Gym, by OpenAI, https://gym.openai.com accessed on 11/22/18

Hollygrimm, Week 3 - Imitiation Learning and Mujoco, https://www.hollygrimm.com/rl_bc accessed on 12/13/18

Ihler, Neural Networks (Presentation slides), https://instructure-uploads.s3.amazonaws.com/account_44070000000000001/attachments/4570924/07-mlpercept.pdf?response-content-disposition=attachment%3B%20filename%3D%2207-mlpercept.pdf%22%3B%20filename%2A%3DUTF-8%27%2707%252Dmlpercept.pdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAJDW777BLV26JM2MQ%2F20181213%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20181213T064815Z&X-Amz-Expires=86400&X-Amz-SignedHeaders=host&X-Amz-Signature=712c2f6939ed19b1e289c12a697171add609530f97e0cf470870ff169903da7e accessed on 11/22/18

Joblib, https://joblib.readthedocs.io/en/latest/ accessed on 11/22/18

Lanier and Nagata, Analysis of Proximal Policy Optimization in Deep Reinforcement Learning

(Poster Submitted on Canvas) viewed 12/13/18
MuJoCo, http://www.mujoco.org/ accessed on 11/27/18