Reinforcement Learning vs Genetic Algorithm Optimization for Robot Walking

Noah Driker, Jason Tran, Luke Ren, Farbod Ghiasi

December 2021

1 Problem Statement

THE goal of this project is to compare the quantitative results of the performance of reinforcement learning and genetic algorithm optimization techniques on the problem of teaching a robot to walk. The performance of the robot walking will be simulated using the MuJoCo physics [1] engine library. These methods will be used to maximize the reward function defined by the MuJoCo physics engine library, which rewards the robot for walking farther distances, with the constraints defined by the MuJoCo physics engine.

2 Previous Work

In order to evaluate the performance of both a reinforcement learning algorithm and genetic algorithm, research was conducted on both fields of algorithms in order to determine the optimal implementations to test and compare with. However, these algorithms required a simulated environment, ideally one based on physical constraints, to be tested on. After conducting research on various physics engines, it was decided that MuJoCo, the 3D physics environment developed by OpenAI as a part of the OpenAI Gym [2] toolkit, would be used to compare the performance of the different algorithms. The MuJoCo physics engine defined the constraints on the input variables of the optimization problem, as well as the reward function whose value would be maximized as the overarching optimization problem. With the physics environment decided and optimization problem defined, it was required to look into the optimal approach for the reinforcement learning algorithm and genetic algorithm.

The approach to reinforcement learning that was taken to maximize the distance travelled by the robots defined in MuJoCo was Proximal Policy Optimization. The different forms of this algorithm are described in the paper, *Proximal Policy Optimization Algorithms* by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov [3]. The paper details the algorithm's use of a clipped surrogate objective function and an adaptive penalty

coefficient to obtain strong performance on continuous domain optimization problems. This algorithm was chosen because the environments provided by MuJoCo were continuous domain environments, and there were many available implementations and resources to use as a foundation for our implementation. For this project, an implementation for PPO from the open-source library, RLlib [4], was integrated and tuned to maximize the reward function defined in the MuJoCo physics environment according the the constraints defined. Likewise, a similar approach to deciding the genetic algorithm was taken.

The genetic algorithm that was implemented in this project was written without a predefined library or framework, but took influence from the book, *Algorithms for Optimization* [5] by Mykel J. Kochenderfer and Tim A. Wheeler, which defined four phases: 1- Initial Population, 2- Selection, 3- Crossover, and 4- Mutation. A predefined framework was not used in order to give the genetic algorithm the most flexibility in terms of configuration and design. Because the algorithm was not extracted from a third-party library, we were able to make alterations and tune it to the specifics MuJoCo environments, as well as include custom population reduction and selection methods. In order to improve the performance of the genetic algorithm, a version of the algorithm applied a technique similar to gradient accelerating momentum, and acted as a feedback control loop for the convergence parameters. This momentum allowed the algorithm to widen its search for the optimum, and also allowed it to more easily find minima by introducing a momentum factor that decays over time to prevent overshooting. With the background of these algorithms laid out, the project was then divided into portions for each group member to work on.

3 Decomposition

The project was divided into three portions, split among the group members Noah Driker, Farbod Ghiasi, Luke Ren, and Jason Tran [6]. Jason Tran worked on integrating the PPO algorithm defined in the open-source library, RLlib, with the physics environments defined in MuJoCo, which included defining an objective function and logging quantitative metrics. Logging and tuning for the reinforcement learning algorithm was also performed using NXDO [7]. Noah Driker and Farbod Ghiasi worked on designing and tuning the custom genetic algorithm to optimize the same environments in MuJoCo. Luke Ren worked on writing and preparing the report, as well as organizing meetings and synthesizing discussions on the implementations of the algorithms. The Python packages, gym and mujoco-py, contained all of the definitions for the physics environments used to test the optimization methods, which were developed by OpenAI. The Python package, ray, contained the code for the open-source library RLlib, used to implement the proximal policy optimization algorithm. Logging and graphing was performed using the matplotlib [8] and tensorboard packages.

The project began with the whole team conducting research on possible Reinforcement Learning and Genetic Algorithms to compare, which resulted in the use of PPO for Reinforcement Learning and a natural selection based

algorithm for the genetic algorithm. Jason Tran, then, developed the program to apply RLlib for PPO on the MuJoCo project, which also included generating plots for performance statistics on the algorithm. Noah Driker and Farbod Ghiasi designed and developed the genetic algorithm, which involved deciding a population reduction heuristic and a population selection decision. They, then, applied the algorithm to the MuJoCo physics environments and measured the same performance statistics as for the reinforcement learning algorithm. Luke Ren prepared the report, including outlining the problem statement, previous work, and analyzing the results in the paper, as well as the poster.

4 Coding Work

Genetic Algorithm

The Genetic Algorithm, inspired by biological evolution, uses an iterative process to produce fitter candidates (aka cands). The process begins by initializing the genesis, a population of randomly generated candidates that will begin the search towards the optimum. Every candidate for this simulation is an action, i.e. an ordered sequence of moves that is fed to the physics environment to control the agent. The algorithm, at every iteration, evaluates every candidate in the generation for fitness, and stores the result (reward). Then, at the end of the evaluation period for each generation, the population is sorted in increasing order of fitness, and the algorithm removes the unfit candidates; this mimics the process of natural selection in biological evolution. The number of candidates that perish is decided by a hyperparameter (see Decimal Perish in the table below). Following the selection of candidates, the algorithm augments the population to include more of the fitter candidates. This is done to increase the likelihood that the fitter candidates will mate. Finally, candidates are mated together randomly, until the number of new candidates equals the next generation size, a parameter that is controlled by a few update functions described below.

Our group made a few modifications to the Genetic Algorithm to improve the rate of learning and to increase the max achieved reward. These include adaptive mutation rates and an oscillating next generation size. There are two functions in our genetic algorithm implementation that are responsible for these adjusting parameters: updateParamsAfterGen() and updateAfterStagnation(). The updateParamsAfterGen() function is called after every generation, and gradually decreases the mutation rate and next generation size to urge the algorithm towards convergence. The updateAfterStagnation() function is called after the maximum achieved reward has stayed the same for 3 or more iterations. Since the agent has not improved for multiple consecutive iterations, the function increases the Mutation Rate and the Next Generation Size to attempt at leaving the local minimum. Mutation Rate is increased to introduce possibly new genes into the pool, and Next Generation Size is increased to promote higher genetic diversity among the offspring. These functions work together to create a feedback control loop, as they automatically adjust the convergence

parameters for the simulation.

Lastly, we entered the best candidate of the current generation into the next population, to guarantee that the best candidate among all, is present in the final generation.

In the table below, we listed the hyperparameters used to control the simulation, and the values that we chose to use.

Hyperparameters		
Parameter	Description	Value
Action Len	Number of moves per action	1000
First Gen Size	Number of candidates in first gen	1000
Default Next Gen Size	Maximum Next Gen Size	500
Num Generations	Length of simulation in generations	2000
Decimal Perish	Decimal part of population that will perish	0.6
Default Chance of Mutation	Initial Chance of Mutation	0.25

The Action Length was chosen to be 1000, as this is the maximum number of moves that the agent can perform in the environment before it is reset. We chose a large number for the First Generation Size (1000), in order to increase genetic diversity in the initial population, and to increase the probability of starting our search in the right direction. The Default Next Generation Size was chosen to be 500, because our algorithm needs enough room for growth when it becomes stagnant. We chose Number of Generations to be 2000, although we've never ran our algorithm with this many generations; the algorithm usually achieves its maximum before this, and is no longer able to improve. The Decimal Perish parameter is selected to be 0.6 (meaning 60% of the population will perish). This was chosen in order to remove the majority of poor performers, but also to keep enough of the population to promote genetic growth. Finally, the Default Chance of Mutation was chosen to be 0.25. This initially forces the mutation rate to be high, so that the algorithm is able to explore many different possible candidates, but the mutation rate is decreased at each iteration with the updateParamsAfterGen() function, to allow the algorithm to converge.

Below are some of the essential pieces used in our implementation of the Genetic Algorithm.

Algorithm 4.1: PERFORMNATURALSELECTION(*Population*)

comment: Remove unfit cands, create next set of parents

$numPerish \leftarrow percentPerish * len(Population)$
$sorted \leftarrow sort(Population)$
$pruned \leftarrow sorted[numPerish : End]$
$nextParents \leftarrow []$

comment: Increase the likelihood of more fit candidates mating

for $i \leftarrow 0$ **to** $len(pruned)$
do { **for** $j \leftarrow 0$ **to** $i + 1$
  do $nextParents$.ADD($pruned[i]$)
return ($nextParents$)

Algorithm 4.1. performs mimics natural selection by removing the poor performing candidates from the population. This Algorithm takes a Population, sorts it in order of increasing candidate reward, and returns the pruned population. This pruned population will be the parents of the next generation.

Algorithm 4.2: CROSSOVER($Parent1, Parent2, p1First$)

comment: See code for implementation

comment: If p1First, Parent1 contributes to first half of genome

return (*combinedGenome*)

Algorithm 4.2. implements genetic crossover. The third parameter, p1First, is the result of a coin-flip done in Algorithm 4.3. This coin-flip decides which Parent will contribute the first half of their genome to the offspring. This algorithm achieves genetic crossover by combining the first half of one parent's genome with the second half of the other parent's genome.

Algorithm 4.3: MATE($Parent1, Parent2$)

comment: Crossover and Mutate

$parent1First = \text{RANDOMCHOICE}(True, False)$
$newGenome \leftarrow \text{CROSSOVER}(Parent1, Parent2, parent1First)$
$geneIndex \leftarrow 0$
for each $gene \in newGenome$
do { **if** (RANDOMBETWEEN$(0, 1) <= chanceOfMutation$)
then { $newGene \leftarrow \text{CREATERANDOMGENE}()$
$newGenome.\text{REPLACE}(gene, newGene)$
return ($newGenome$)

Algorithm 4.3. implements mating among candidates. Mating involves performing genetic crossover with the two given parents, and deciding by chance whether to mutate a gene in the genome. This algorithm returns the new genome.

Algorithm 4.4: MATEPARENTS($Parents$)

comment: Randomly mate parents to create next generation

$nextGeneration \leftarrow []$
$candidatesCreated \leftarrow 0$
while $candidatesCreated < nextGenerationSize$
do { $p1, p2 \leftarrow \text{RANDOMLYSELECTTWO}(Parents)$
$newCandidate \leftarrow \text{MATE}(p1, p2)$
$nextGeneration.\text{ADD}(newCandidate)$
$candidatesGenerated \mathrel{+}= 1$
return ($nextGeneration$)

Algorithm 4.4. randomly selects pairs of parents to mate, and returns the next generation of candidates.

Algorithm 4.5: CREATENEWGENERATION($Population$)

$newParents \leftarrow \text{PERFORMNATURALSELECTION}(Population)$
$nextGeneration \leftarrow \text{MATEPARENTS}(newParents)$
return ($nextGeneration$)

Algorithm 4.5. uses Algorithm 4.1 and Algorithm 4.4 to create a new generation from the current population.

Algorithm 4.6: EVOLVE()

```
generationIndex ← 0
while (generationIndex < numGenerations)
  do { population ← GENERATEGENESIS()
       for each candidate ∈ population
         do {candidate.reward = EVALUATECANDIDATE()
       population = CREATENEWGENERATION(population)
       UPDATEPARAMETERS()
       UPDATEAFTERSTAGNATION()
```

Algorithm 4.6. implements the core of the Genetic Algorithm. Evolve generates the genesis, and iteratively steps through each generation, evaluating each candidate, creates new generations, and updates the convergence parameters.

Reinforcement Learning

Reinforcement Learning is a form of machine learning that trains an agent to learn and perform tasks in an environment with a reward and observation in mind. For this project we have an agent learning to either walk as a Half Cheetah or as a Humanoid inside of the MuJoCo environment. In order for an agent to learn, it needs a policy[1] which maximizes the reward obtained for an action. To do so, policy gradient methods are often used to optimize policies with respect to the reward it generates by using gradient descent. Proximal Policy Optimization (PPO), a simpler and more effective variant of Trust Region Policy Optimization (TRPO)[2], is used in this project to train an agent to walk in different MuJoCo environments. The implementation of PPO used to train agents is from Ray RLlib[3] in Python.

A baseline of the reinforcement learning section of the project was initially created to ensure that the MuJoCo OpenAI environment worked. The agent of this version only had random actions and did not learn. Afterwards, a basic implementation using Ray RLlib was created and tested. The reward per iteration of the this version was logged into a text file and graphed via python's matplotlib library as it was simply to test if the agent was learning. The last and final implementation used a different logger and simplified the environment class. This version viewed graphs through tensorboard and its hyperparameters were tuned via the Ray tune library in-order to optimize the trainer for better rewards. The logger and tuning implementation used for the final reinforcement learning version are from nxdo[7].

[1] An approach or strategy an agent utilizes to accomplish a task.

[2] Another policy gradient method often used in reinforcement learning which uses trust regions to optimize policies.

[3] An open source library for reinforcement learning.

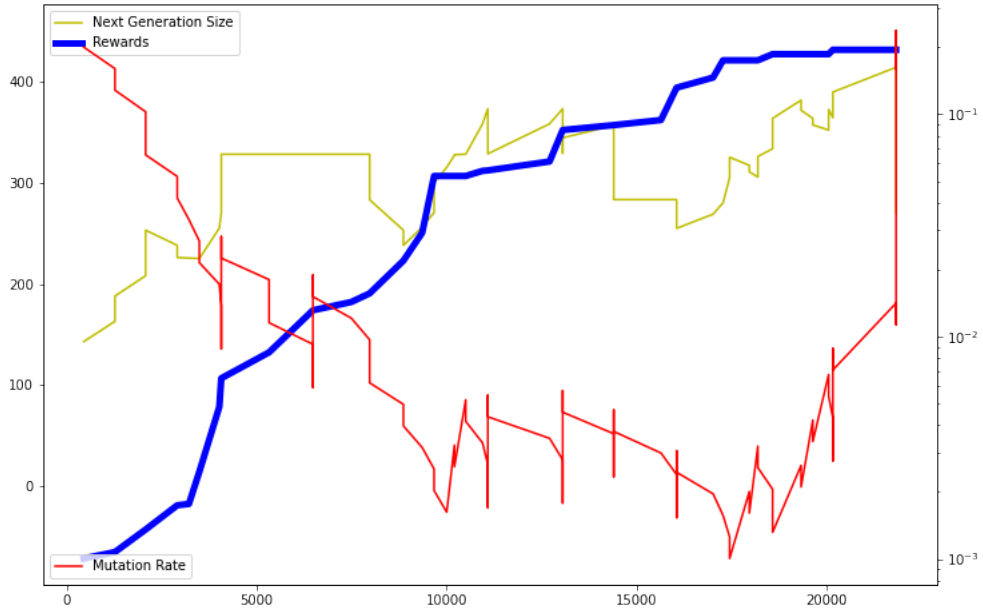

Figure 1: Reward, Mutation Rate, and Next-Generation Size for Half Cheetah

5 Quantitative Results

The results for the performance of the genetic algorithm on the Half Cheetah environment in MuJoCo are given in Figure 1. The results for the performance of the genetic algorithm on the Humanoid environment are given in Figure 2.

In the performance graphs, the total rewards yielded at each iteration of the algorithm are graphed against both the mutation rate and next generation size values, which are also adapted at each iteration. For both environments after 15 hours of training, the total rewards value has a strong positive correlation with the number of iterations, but the performance improved more slowly over time, with the total maximum reward reaching around 3300 for the Humanoid environment and 450 for the Half Cheetah environment. For the Half Cheetah graph, the next generation size maintains a weak positive correlation with the number of iterations, identifying that the algorithm is slowly increasing the generation size as the number of iterations increases. This correlation is less clear for the Humanoid graph, but the positive correlation is still identifiable. This correlation can be attributed to the population allowing a larger search space for the algorithm to improve. The sharp increases in mutation rate are also cor-

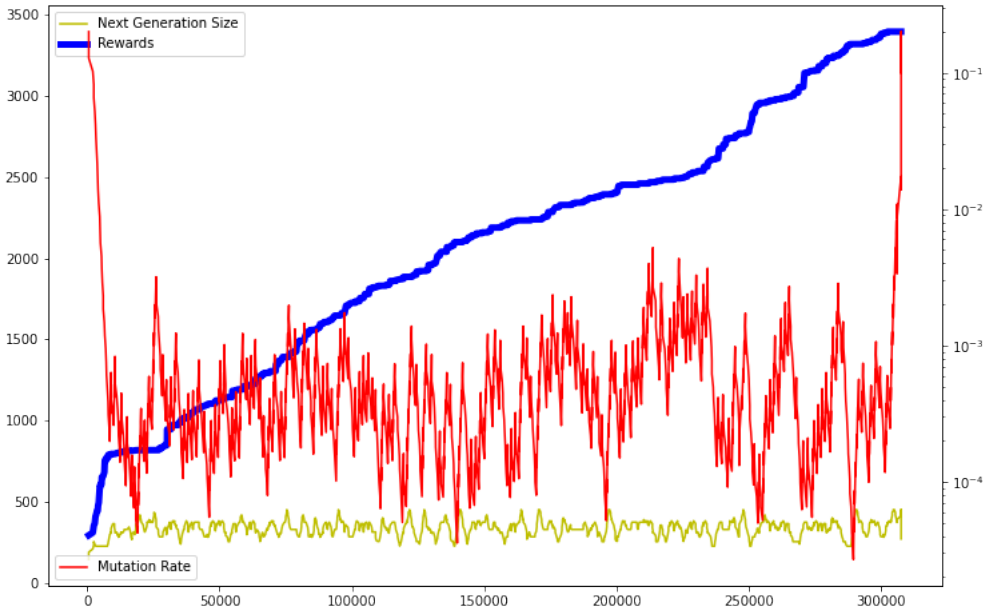

Figure 2: Reward, Mutation Rate, and Next-Generation Size for Humanoid

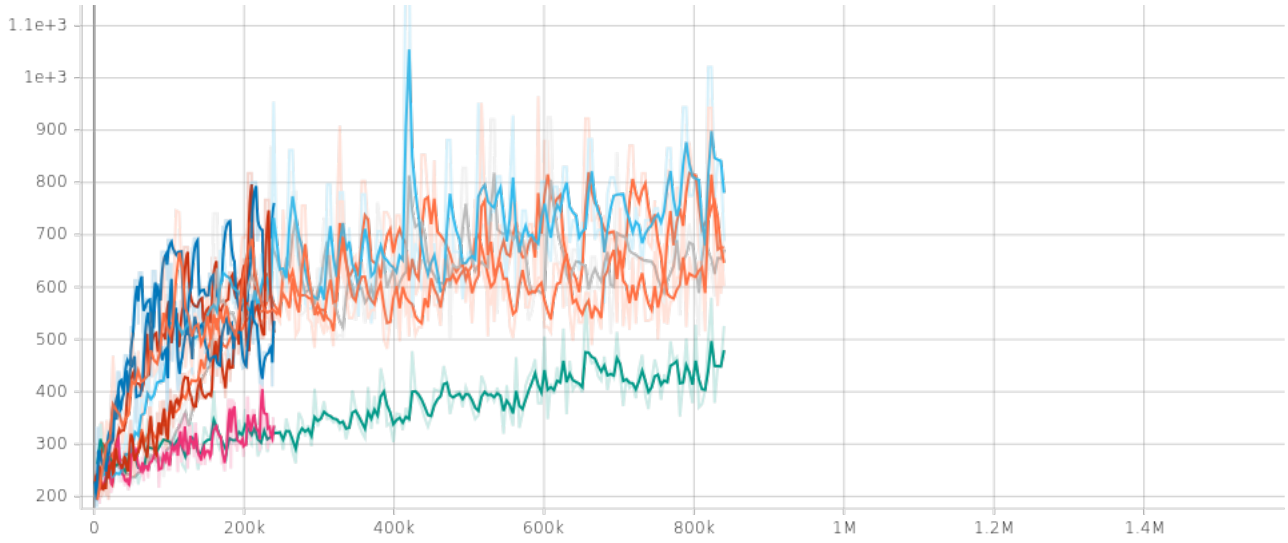

Figure 3: Max Episode Reward Tuning Result for Humanoid

related with improvements in the total rewards value, which can be attributed to the mutation rate diversifying the search space of the algorithm, with the reduction of the mutation rate allowing the individual models to converge towards local minima.

In Figure 3 and Figure 4, the results of parameter tuning the reinforcement learning algorithm, Proximal Policy Optimization, on the Humanoid and Half Cheetah physics environments from MuJoCo are displayed respectively. The different lines represent different combinations of hyper parameter values chosen when tuning the algorithm. By performing tuning, the best combination of hyper parameters from the sample can be decided before training with the algorithm over an extended period of time.

The performance graphs for reinforcement learning when trained on the Humanoid environment are displayed in Figure 5 and Figure 6 respectively. The graphs identify a strong positive correlation between the total rewards and number of iterations ran for the algorithm, however, the performance seems to taper off for the Humanoid environment around the 15 million iteration mark and the Half Cheetah environment around the 3 million iteration mark. Although the algorithm was run for 8 hours for both environments, the algorithm stopped improving significantly around the 6 hour mark for the Humanoid environment and around the 4 hour mark for the Half Cheetah environment. Given this amount of time, the algorithm was able to achieve a maximum total reward value of around 6,000 for the Humanoid environment and 1,500 for the Half Cheetah environment.

Comparing the performance of the two algorithms, although reinforcement learning required a tuning phase to decide the best parameters, the overall

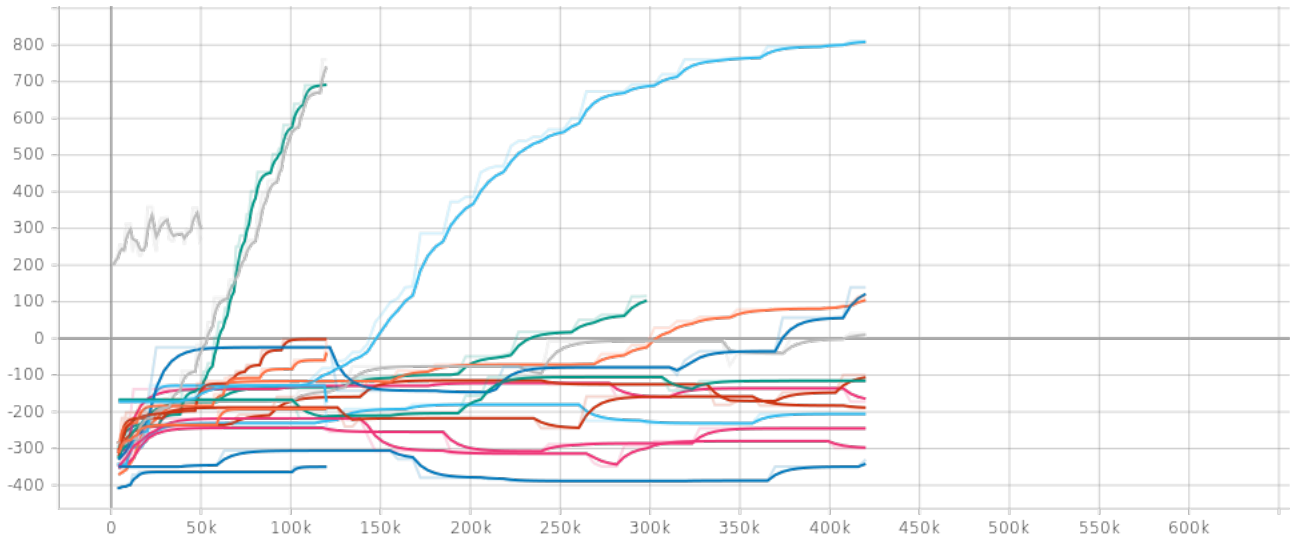

Figure 4: Max Episode Reward Tuning Result for Half Cheetah

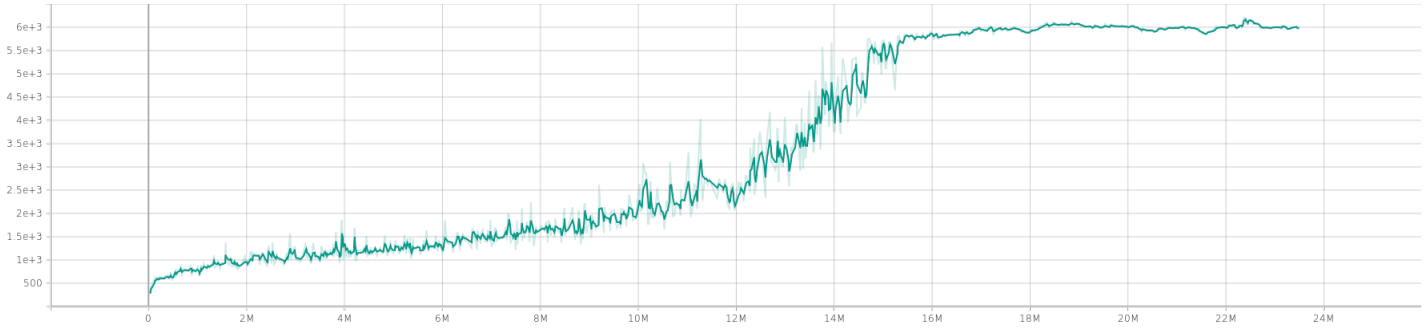

Figure 5: Training Results for Humanoid (Reinforcement Learning)

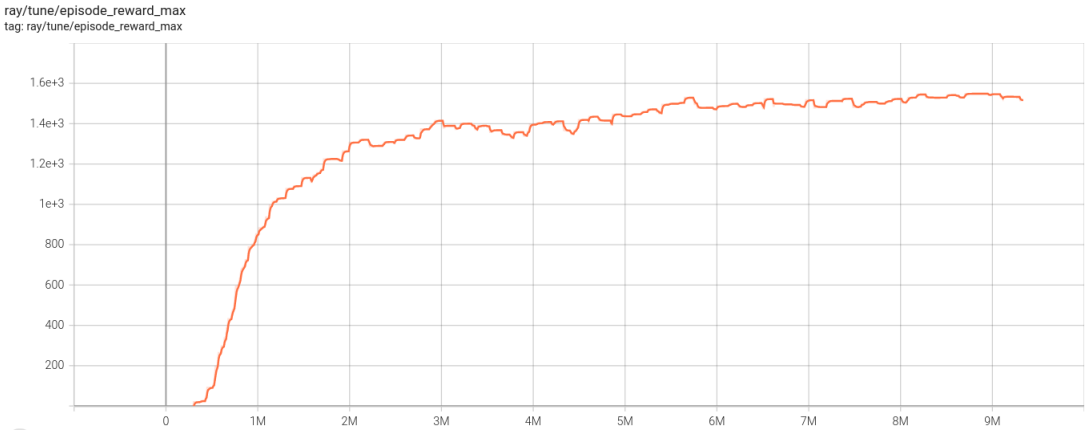

Figure 6: Training Results for Half Cheetah (Reinforcement Learning)

training time and maximum rewards were better for reinforcement learning. For the Humanoid environment, the reinforcement learning algorithm was able to achieve a total maximum reward of over 6,000 after 8 total hours of training, while the genetic algorithm achieved a total maximum of around 3,500 after 15 hours of training. For the Half Cheetah environment, the reinforcement learning algorithm was able to achieve a total maximum reward of over 1,500 after 8 total hours of training, while the genetic algorithm achieved a total maximum reward of around 450 after 15 total hours of training.

6 Conclusion

Based on the results mentioned above, as well as a visual analysis of the graphs produced by training with both optimization algorithms on the MuJoCo physics engine environment, it can be concluded that the reinforcement learning algorithm had a stronger overall performance. While both algorithms yielded strong positive correlations for the reward over the number of iterations, the reinforcement learning algorithm implemented using PPO was able to yield higher total maximum rewards quicker. Thus, it can be concluded, given the amount of time and processing power available, that the reinforcement learning algorithm has overall better performance than the genetic algorithm for the continuous, constrained optimization problem of maximizing reward in a simulated robot walking environment.

7 Possible Improvements

Although the Genetic Algorithm and Reinforcement Learning were able to simulate a half cheetah and humanoid walking, they can still attain superior re-

wards to what was presented in this report. Genetic Algorithm can be improved by using a feedback control algorithm, such as AIMD[4] or MIAD[5], to better control the mutation rate and next generation size. Furthermore, Genetic Algorithm could be implemented differently in a manner similar to NEAT[6] which is an evolutionary algorithm that creates artificial neural networks[9]. As for Reinforcement Learning, allowing more time and access to additional computational power would greatly improve the results. When tuning the hyperparameters for Reinforcement Learning, the time allotted for tuning did not exceed 8 hours and only 4-8 cores were allowed and no GPUs were used for further computational power. Allowing for more time would allow the tuner to find more hyperparameters to test while more computational power would allow testing to occur much faster. As a result, the tuner would be able to optimize the hyperparamters much faster and efficiently.

References

[1] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. *arXiv preprint*, 2012.

[2] G. Brokcman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint*, 2016.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017.

[4] E. Liang, R. Liaw, R. Nishihara, P. Mortiz, R. Fox, and K. Goldberg. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint*, 2016.

[5] M. J. Kochenderfer and T. A. Wheeler. *Algorithms for optimization*. MIT Press, 2019.

[6] N. Driker, F. Ghiasi, L. Ren, and J. Tran.

[7] Stephen McAleer, John Lanier, Pierre Baldi, and Roy Fox. Code for xdo: A double oracle algorithm for extensive-form games. `https://github.com/indylab/nxdo`.

[8] John D. Hunter. Matplotlib: A 2d graphics environment. *IEEE*, 2007.

[9] NEATPython. Neat overview. `https://neat-python.readthedocs.io/en/latest/neat_overview.html`.

[4] Additive Increase/Multiplicative Decrease
[5] Multiplicative Increase/Additive Decrease
[6] NeuroEvolution of Augmenting Topologies