To what extent is a Reinforcement Learning Algorithm more effective than a Genetic Algorithm in training a 2D Agent to walk and overcome obstacles?

A Computer Science Extended Essay

Word Count: 3998

Person Code: hqy392

Person Code: hqy392

1 Introduction

We constantly hear about new developments regarding humanoid robots, such as Boston Dynamics' Atlas, a robot that can run, jump, flip, and more.[1] These advancements will likely lead to an increase in global productivity, as more jobs become automated and efficiently done. One of the essential and difficult skills of humanoid robots is walking, which will be investigated in this essay.

Although simple for humans, teaching a robot to walk; which joints to move, how much force to exert, and more, is too much information to preprogram a walking system that would apply across varied environments. This is where Machine Learning (ML) is useful, ML is the study of algorithms that improve automatically through experience and/or the use of data, without the need to preprogram.

To simplify the problem of walking with the resources available, a 2D simulation from OpenAI Gym will be used. Gym is a collection of simulation environments for developing and comparing ML Algorithms.[2]

There are different subsections of ML, however the ones that will be researched, implemented and compared are a Reinforcement Learning Algorithm and a Genetic Algorithm.[3]

This leads to the Research Question: To what extent is a Reinforcement Learning Algorithm more effective than a Genetic Algorithm in training a 2D Agent to walk and overcome obstacles?

2 Background Information

2.1 Reinforcement Learning

Reinforcement Learning (RL) algorithms are algorithms where an agent makes observations and takes actions within an environment (typically a game or a simulation). An agent is anything which can sense its environment, and act upon it through effectors. In return, it receives rewards and punishments, numerical values which determine the agent's performance. The agent's goal is to maximize expected long term rewards.

The agent employs a policy, a set of rules the agent follows to gain rewards. In the example of walking, a preprogrammed policy would be to determine specific values for force, how much to move certain joints, and other factors. Then program the agent to always use those values. However, preprogrammed policies are often too convoluted and/or ineffective due to varied and complicated environments. Hence the agent must learn the optimal policy.

To learn the optimal policy, one must first understand the problem. Oftentimes RL problems can be modelled as Markov Decision Processes (MDPs). An MDP is a process that has states, actions, rewards, and punishments. MDPS have the Markov property, which states that the current state is dependent on only the previous state and action. Hence being completely deterministic, allowing for a clearer understanding of the problem and the ability to use common RL algorithms to solve it.

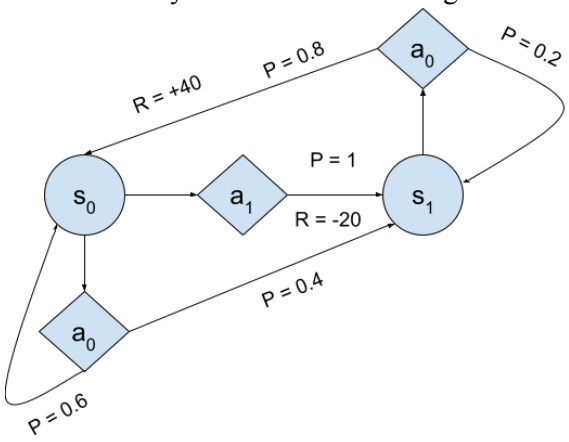

Figure 1: Example of an MDP

s = State
a = Action
R = Reward
P = Probability of path being taken

The diagram above displays a simple MDP, starting from s_0, the agent can perform either a_0 or a_1, if it performs a_0, there is a 60% chance nothing will change, and a 40% chance that the agent moves to s_1. If it performs a_1, the agent loses 20 points, but is guaranteed to move to s_1. From s_1, it can only perform a_0, which has an 80% chance of returning the agent to s_0 and adding 40 points, or a 20% chance of nothing changing. This is a simplified example, where an effective policy would be to always take a_0, as to never lose points, unless there was a time limit, then a_1 could be performed at times. The problem presented will be much more complex, and is infeasible to model in a diagram.

The agent must balance between exploring and exploiting an MDP, let us say that the agent only knew Figure 1, but there was also an s_2 which could gain the agent 100 points. If the agent were to only exploit what it knew, it would only be able to receive 40 points by moving from s_0 to s_1, rather than take advantage of the higher rewards. Therefore exploration and exploitation must be balanced.

One of the main issues found within MDPs is the credit assignment problem: When an action is taken, a reward is received. And many actions are taken with rewards accumulated. But not each action is independently responsible for its reward, as it's dependent on the previous state and action as described by the Markov Property. This leads to the credit assignment problem, which asks how much credit should each action be assigned for it's reward?

To solve this problem, an action is evaluated using a sum of the rewards afterwards, multiplied by a discount rate gamma (γ), which ranges from 0 to 1, typically around 0.95 to 0.99. For every action, we take a higher power of gamma. Hence each reward in the future has less credit assigned for that action.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = R_{t+1} + \gamma(G_{t+1})$$

Equation 1: Cumulative Reward.

G = *Cumulative Reward*

R = *Reward*

γ = *Discount Rate*

t = *Time step*

This computation can be simplified as simply adding the reward of the next timestep to the cumulative reward of the next time step, this allows the agent to train faster and better understand which actions are responsible for rewards and punishments. Then using the actions in the optimal policy.[3]

Another concept to introduce are global and local optimums: the long term expected rewards in an environment can be modelled as a reward function. For the sake of simplicity and explanation let us take a reward function with one parameter input:

Figure 2: Example of a Reward Function with a global and local optimum

As shown in the graph above, this function has two peaks, or optimums. Let us say that a policy starts at the origin, earning no reward. By exploring and exploiting the environment, let us say it finds a policy that earns rewards at the smaller peak, and seeing that slightly tweaking the parameter only results in worse outcomes, it assumes that this is the best outcome, with the highest reward. This is known as premature convergence, which is when a policy is stuck at a local optimum, or smaller peak. Optimally, we want the algorithm to reach the global optimum, the higher peak. In reality, reward functions tend to be much more complex than this, with many parameter inputs and many local optima, yet often local optima are deemed adequate.

2.2 The Problem

The Bipedal Walker Hardcore environment is a pseudorandomly generated environment in which a 2D bipedal walker must reach the end of a terrain - containing pitfalls, stairs, and blocks - as fast as possible. The agent receives upwards of 300 points for going forward, and if it falls (hull touching the ground; Figure 3) the environment terminates and it receives -100 points. Solving the environment is defined as having a policy where the agent receives an average of ≥300 points over 100 consecutive trials.[1]

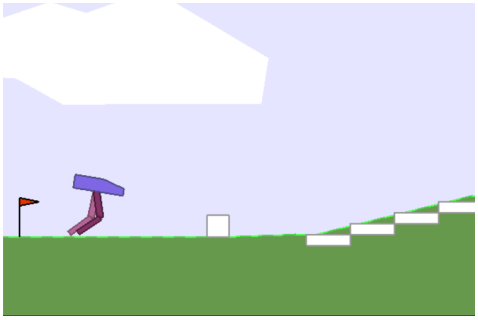

Figure 3: An example of the Bipedal Walker Hardcore Environment.

The agent's observations consist of the values in Figure 4, which detail the positioning and velocities of its body and legs, as well as 10 lidar readings to determine the terrain and upcoming obstacles. This is everything the agent knows about the environment, or a state in the MDP.

Num	Observation	Min	Max	Mean
0	hull_angle	0	2*pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	+inf	-
7	knee_joint_1_speed	-inf	+inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	+inf	-
12	knee_joint_2_speed	-inf	+inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-inf	+inf	-

Figure 4: The Bipedal Walker Hardcore Environment's Observation Space.[4]

The agent's actions consist of 4 values from -1 to 1, which are values of Torque/Velocity for the hips and knees. Hence an action outputted by the policy will be an array of 4 values ranging from -1 to 1, determining how each of these joints would move.

Num	Name	Min	Max
0	Hip_1 (Torque / Velocity)	-1	+1
1	Knee_1 (Torque / Velocity)	-1	+1
2	Hip_2 (Torque / Velocity)	-1	+1
3	Knee_2 (Torque / Velocity)	-1	+1

Figure 5: The Bipedal Walker Hardcore Environment's Action Space.[4]

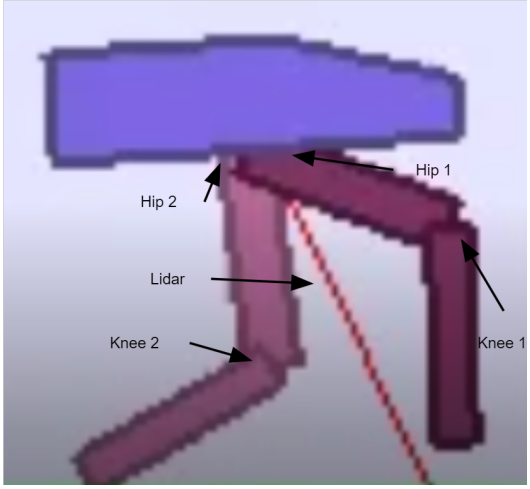

Figure 6: Labelled Diagram of Bipedal Walker (Note that numbering of hips and knees is not listed and these could be reversed, however this ultimately has no impact on the algorithm)

2.3 Neural Networks

Neural Networks (NNs) are one of the most common algorithms for ML. Artificial NNs are inspired by the brain, which works by having neurons and connections between them to create an NN. Each neuron on it's own is fairly simple, but when all connected together, they can recognize patterns and perform complex tasks.

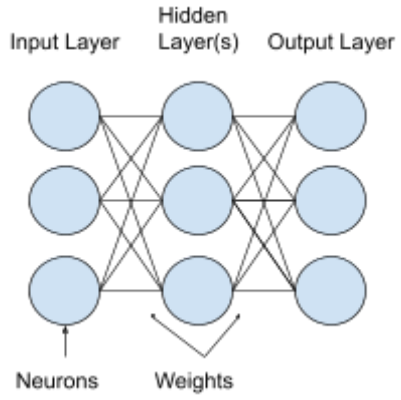

Figure 7: Simplified NN Diagram

In ML, neurons contain numbers used for computation. The neurons in the input layer will take in the observation space or a state (Here there would be 23 neurons; Figure 4). These neurons then activate different neurons in the hidden layer based on the weights. The weights are numbers that determine how strong of a connection there is between each neuron, and how much a neuron activates based on the previous ones. Therefore, each neuron's value is a sum of every neuron in the previous layer multiplied by every weight. Furthermore, a bias typically added to this sum when there is a threshold the neuron should meet to be active for a certain solution. These activations lead to the output neurons being activated a certain amount, resulting in the action.[5]

$$S = w_1 n_1 + w_2 n_2 + w_3 n_3 + \ldots + w_k n_k + b$$

Equation 2: Weighted sum with a bias

w = *Weight*
n = *Neuron*
b = *Bias*
S = *Sum*
k = *Number of Neurons in the previous layer*

If the network results in a low reward, it will change the weights between the neurons to learn which connections are most beneficial. The purpose of the layered structure is to generalize patterns. At the end of the network is the output layer, which here has 4 output neurons as described by the action space (Figure 5).

An aspect of NNs are activation functions. Within the network, the neurons could have any positive or negative value, and this large variety slows down the training of the NN. Furthermore the weighted sum is a linear result, and there are often nonlinear functions that need to be learnt by NNs. Hence, by inputting a linear sum into a nonlinear function, you receive a nonlinear result, allowing for more complex solutions from the NN.

At the end of the network the values need to be between -1 and 1 (Figure 5), hence an activation function is required, such as the hyperbolic tangent function.

$$tanh(w_1 n_1 + w_2 n_2 + w_3 n_3 + \ldots + w_k n_k + b)$$

Equation 3: Weighted sum with a bias inputted into the tanh activation function

An NN's weights must be initialized before any training can be done. Typically, initialization is done with random weights, however there are other initializers that can be used that have different levels of success depending on the activation functions used.

NNs are what are known as universal function approximators, meaning that given enough complexity, they are able to approximate any function. This allows them to be used for any ML problem, since they approximate the function associated with the problem, in this case being the function that represents the effectiveness of a policy.[3]

2.4 Proximal Policy Optimization

One of the most common RL algorithms are Policy Gradients, which implement Gradient Descent. Gradient descent is an optimization algorithm in ML used to find the minimum of a cost function. A

function that returns the error of the agent given the policy provided. This is done by following the gradients until they are 0. These are then used to update the weights of the NN (the same method can be used for a reward function, using Gradient Ascent).

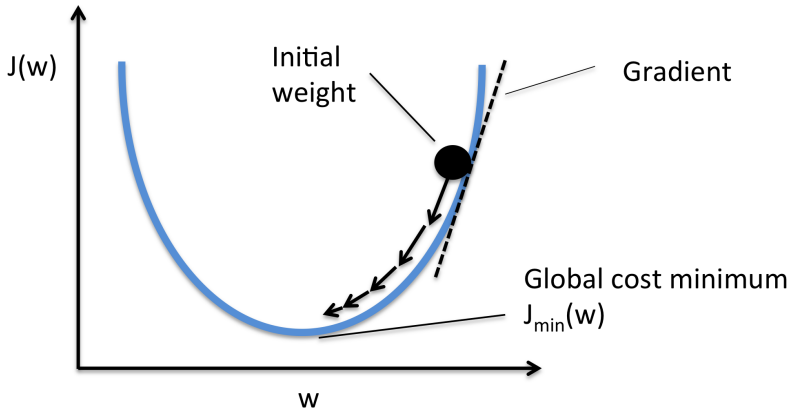

Figure 8: Illustration of Gradient Descent.[6]

Policy gradients employ this same idea but for RL policies. However, they tend to be too sensitive to the step size, the size of the arrows from the initial weight in Figure 8. If they're too small, the algorithm takes too long to converge to an optimum, and if they're too large, it tends to hop around the cost function, with difficulty converging. Because of this, OpenAI decided to develop a new algorithm known as Proximal Policy Optimization (PPO).

PPO deals with the step size problem using a clipped surrogate objective function. Which constrains the step size or policy change using a clip. The algorithm takes an action and computes the ratio of the action being taken in the new and old policy.

$$r_t(\theta) = \frac{\pi_\theta(a_t|\, s_t)}{\pi_{\theta old}(a_t|\, s_t)}$$

Equation 4: Ratio between new and old policy

θ = *Policy parameters, in this case the weights of the NN*
t = *Time Step*
a = *Action*
s = *State*
π = *Policy*

As can be seen in the equation, The input into the new and old policy is the probability that action a_t will be taken given that the agent is in state s_t. The ratio is computed by dividing the two probabilities, which can only be between 1 - ϵ and 1 + ϵ. Hence ensuring that the new policy is not drastically different from the older one, allowing training to be more stable.[7]

PPO is an Actor-Critic Algorithm, meaning that not only is there the agent network taking actions in the environment. But also a critic network, whose goal is to evaluate the action that the actor took, and adjust the actor into having a more optimal policy. Since NNs are universal function approximators, the actor is approximating the optimal policy, the best way to walk, meanwhile the critic is approximating the reward function, the best results that could be achieved. Which is approximated by the agent exploring and exploiting the environment. These two can be thought of as "competing" with each other, both gaining

information until the actor reaches an optimal policy, and the critic reasonably approximates the reward function.[8]

2.5 Genetic Algorithms

Genetic algorithms (GAs) are based on evolution and natural selection. Where there is a population of random agents, and over the course of many generations, the agents improve till they reach the global optimum. There are a few methods inspired from nature to create the next generation. The first one is crossover, in which 2 fit agents "breed" and exchange "genes" with each other to create a fitter agent (genes being the weights of the NN). Therefore, 2 networks exchange their weights to create a new fitter network. Crossover exploits the environment, as the agents are using what they already know works to make them even better.

Another method to exploit the environment is elitism, where the highest performing agents are not interfered with, and are taken to the next generation. This is to ensure that a strong policy is not possibly ruined.

A method to explore the environment is mutation, in which an agent is taken and a gene is randomly changed to a different value. Each agent has the possibility to mutate based on the mutation rate. This helps explore the environment as the algorithm could stumble upon a more optimal policy.[9]

To select the best and varied parents for the next generation, selection algorithms are used. Typical selection algorithms use the idea that if a parent has a higher fitness value, or an output from the reward function, it is more likely to be chosen as a parent, such as Roulette Wheel Selection. Roulette Wheel Selection normalizes the fitness values of every agent to a value between 0 and 1. Which is used as the probability of the agent being chosen to crossover.

However, this algorithm can be flawed, since if there are a few agents that vastly outperform the others, they are almost always guaranteed to be chosen, destroying genetic diversity. This could leave the algorithm with a more sub-optimal policy than if it had kept some of the more worse ones initially.

To solve this, there is another algorithm known as Ranked Selection. Rather than using fitness values, parents are ranked, and their probability of being chosen is based on their rank rather than fitness. Using these ranks, we can use the probability formula below to find the probabilities of each parent being chosen:

$$P(R) = \frac{1-\frac{R}{N}}{\sum_{r=0}^{N}(1-\frac{r}{N})}$$

Equation 5: Probability of an agent being chosen in Ranked Selection

$P(R)$ = *Probability of an agent being chosen*
R = *Rank*
N = *Number of Agents*

With a simple example of 5 agents we can see how Ranked Selection computes probabilities. The best agent, ranked one, would have a numerator of $1 - \frac{R}{N} = \frac{4}{5}$, and the denominator would then add up to $\frac{7}{24}$, leading to P(1) = 0.2503. The other agents have slightly lower values, with the exception of the 5th ranked one with a probability of 0. With these probabilities, there is much more diversity in the population.[10]

3 Methodology

3.1 Reinforcement Learning

I will be using Python for both algorithms, as it is one of the most common languages for ML and has extensive support. For the RL model, I decided to use a library called TensorFlow, as it would simplify the creation of the NNs with the functions provided. My algorithm was based on a different implementation of PPO, designed for the Bipedal Walker environment (the same environment with even terrain and no obstacles). I decided to increase the complexity of the NN as to hopefully learn more complex patterns required to overcome the obstacles. The original NN had 3 hidden layers of sizes 512, 256, and 64, and I added two layers of sizes 128 and 32, such that they were in descending order.[11]

For the activation function I decided to implement Leaky ReLU, which is an improvement of the ReLU function. ReLU stands for Rectified Linear Unit, which is:

$$ReLU(x) = \begin{cases} 0 & if\ x \leq 0 \\ x & if\ x \geq 0 \end{cases}$$

Equation 6: ReLU

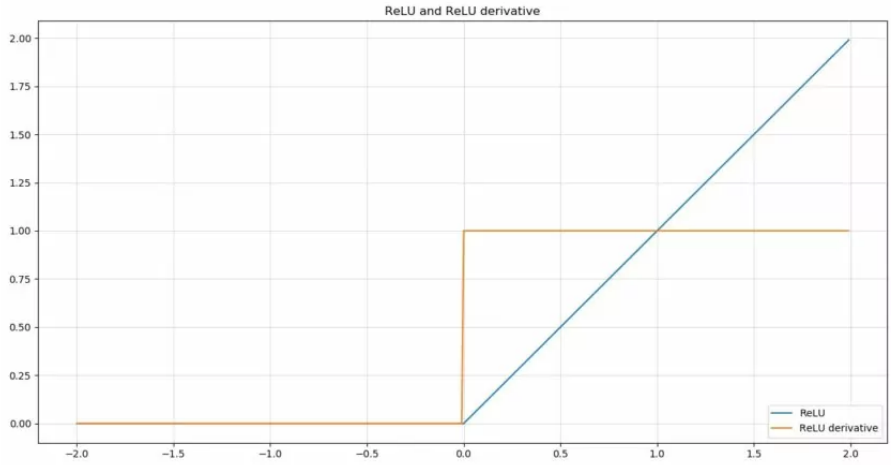

Figure 9: Graph of ReLU and ReLU Derivative[12]

This activation function tends to do better than others due to the vanishing and exploding gradients problems. When the network is implementing policy gradients, it goes layer by layer from the output to the first layer, the gradients used to train the network can either decrease or increase rapidly as they're going back. Meaning that earlier layers of the network are either barely affected or over affected. This problem can be remedied by both the activation function and the initialization.[3]

Unlike other activation functions, ReLU remedies the gradient problems by having it's derivative always being a constant. Increasing simplicity as well as a decreased likelihood of gradients increasing or decreasing rapidly. This leads to ReLU being commonly used in NNs.

As for Leaky ReLU, it is slightly different from ReLU in that instead of all values below 0 outputting 0. They are multiplied by a small value alpha, in this case 0.01.

$$Leaky\ ReLU(x) = \begin{cases} 0.01x & if\ x \leq 0 \\ x & if\ x \geq 0 \end{cases}$$

Equation 7: Leaky ReLU

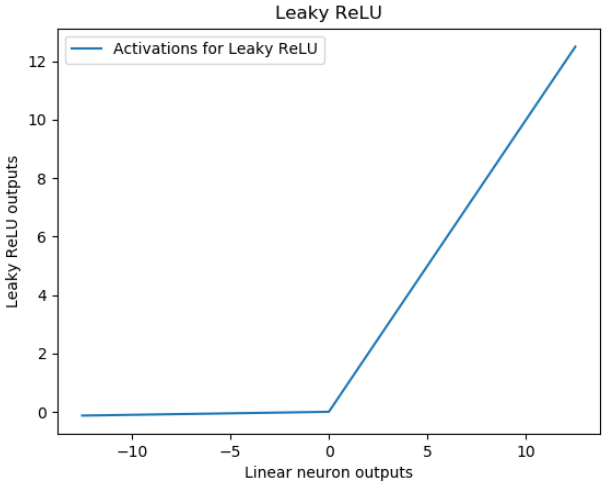

Figure 10: Graph of Leaky ReLU

This remedies what is known as the dying ReLU problem, in some cases, if a neuron is negative and through the activation function becomes 0, it has no way of becoming active again. Which can be detrimental to the network's performance. If the agent learns a sub-par walking strategy, leading it to deactivating certain neurons, it would be much harder for it to find a more optimal strategy as the neurons would have difficulty recovering from being deactivated. In Leaky ReLU, these neurons still have a chance of recovery.[13]

I also changed the initialization function from being random to using He initialization. Random initialization can have any variance, and is useful in most cases. However when variance is too low the network is once again prone to the vanishing gradients problem. And vice versa for high variance. Hence there are a few situations in which other initialization strategies can prove useful. One of these is He initialization, which is often used with ReLU and Leaky ReLU as it has been proven to theoretically work better with those activation functions. He initialization initializes weights with a variance of 2/N where N is the number of input weights.[14]

These were the main changes made to the PPO algorithm, all else was implemented as described in the original paper and article for the Bipedal Walker Environment.[7]

3.2 Genetic Algorithm

For the GA, I adapted code from another implementation of GAs for Bipedal walker, and changed the selection probability functions, parameters, and more to implement it in the hardcore environment. For selection probability, I implemented Ranked selection to ensure genetic diversity within the model. As for the other parameters I had:

Number of generations	100
Number of chromosomes/agents	500
Mutation rate	10%
Elitism percentage	30%

Table 1: GA parameters

I had chosen a higher number of agents and lower amount of generations to implement a more exploratory approach. This was in hopes of a few of the agents being lucky and successful, then passing those on to the next generation through elitism and crossover. The lower amount of generations was due to my assumption that there would be marginal returns after many generations.

Mutation was implemented by randomly changing one of the weights in an agent's NN. With all agents being initialized using random weights.[15]

4 Results

4.1 Reinforcement Learning Results

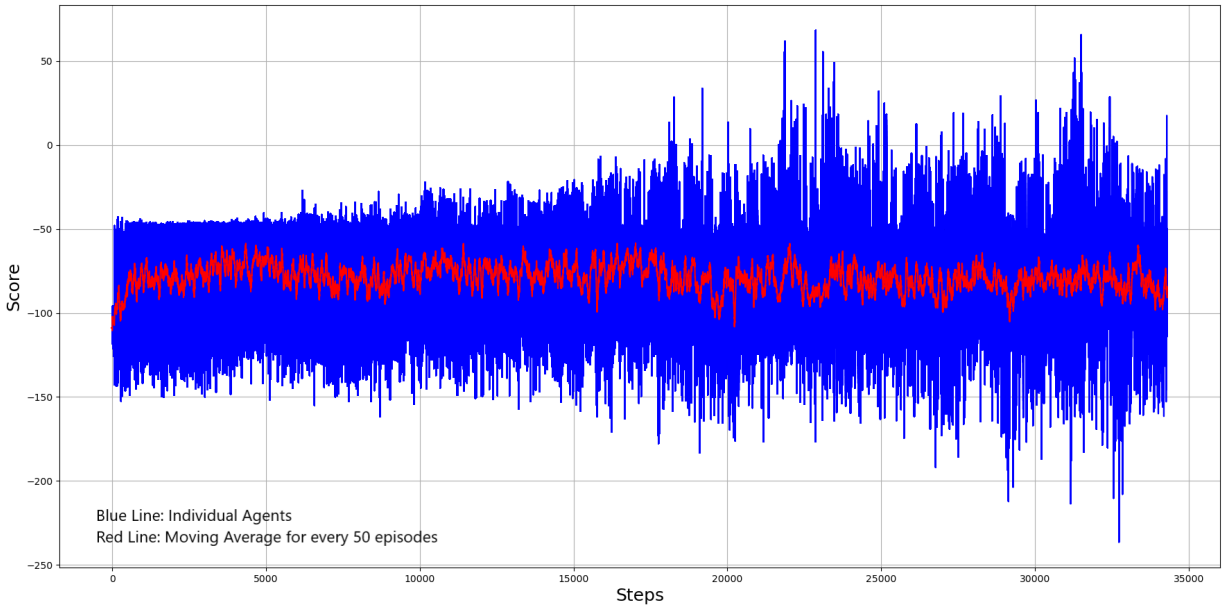

Figure 11: Graph of the score of the RL Algorithm

On this diagram, the x-axis represents the steps or episodes of the simulation, which ends each time the hull falls or the time limit is over. And the y-axis is the score accumulated throughout the episode. The algorithm reaches only about a -55 score for it's moving average, and just over 50 in a few of the agents. Far off the 300 required to solve the environment. Because of this, I decided to tweak a few parameters in the network to achieve better results, but I was unable to gain any significant improvement after 4 trials.

It is worth noting that as time went on, training became more and more unstable, with the blue lines having a larger range, while the moving average remained relatively constant. I am unclear as to what could have caused this, with my best guess being that the algorithm realized it was in a local optimum, and used more of an exploratory approach to try and exit it. Leading to more variation. However it is likely that the more successful agents were due to lucky environment generation, rather than an improvement in the policy.

To improve the model, I recalled an ML technique known as Transfer Learning. Transfer learning is when an ML model is trained to solve one problem, then taken and retrained to solve a different one. Aiming to speed up learning as well as increase efficiency, since there are often similar problems that require similar solutions.[3] Knowing this, I trained the model on the Bipedal Walker environment. Then retrained it in the

Hardcore environment.

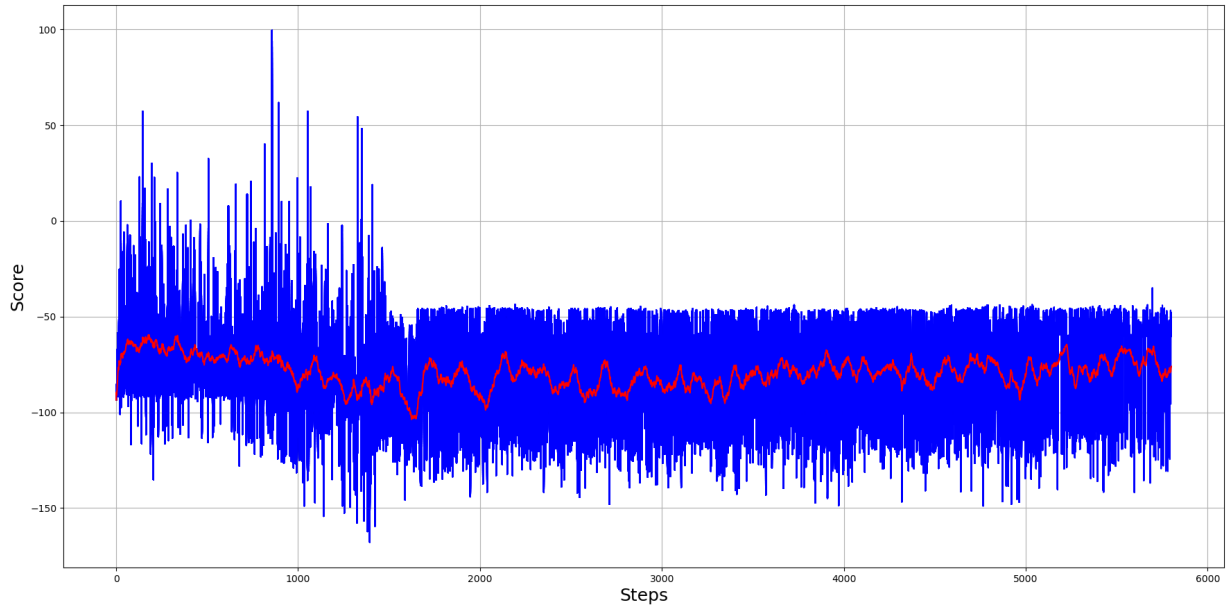

Figure 12: Graph of the score of the RL Algorithm using Transfer Learning (training was stopped early as there was no improvement)

As can be seen, initially there appears to be an improvement, with individual agents scoring much higher than before and a moving average that could improve. However the algorithm quickly plummets and is stuck in a local optimum. After changing many of the parameters, totalling in 7 more trials, these were the best results I was able to achieve.

In contrast to Figure 11, the training of this model was more unstable, then converged onto a local optimum. Since the model started with knowing how to walk, these high scores were likely due to lucky environment generation. The model likely learnt that it had to adapt it's method to suit these obstacles, but was unsuccessful in doing so, leading to the local optimum.

When rendering the environment, I noticed that the agent was able to walk perfectly fine, would be able to get past stairs with relative ease, occasionally overcome pitfalls, and mostly fail on blocks.

Figure 13: Examples of the agent falling over a block, and succeeding at stairs and pitfalls.

4.2 Genetic Algorithm Results

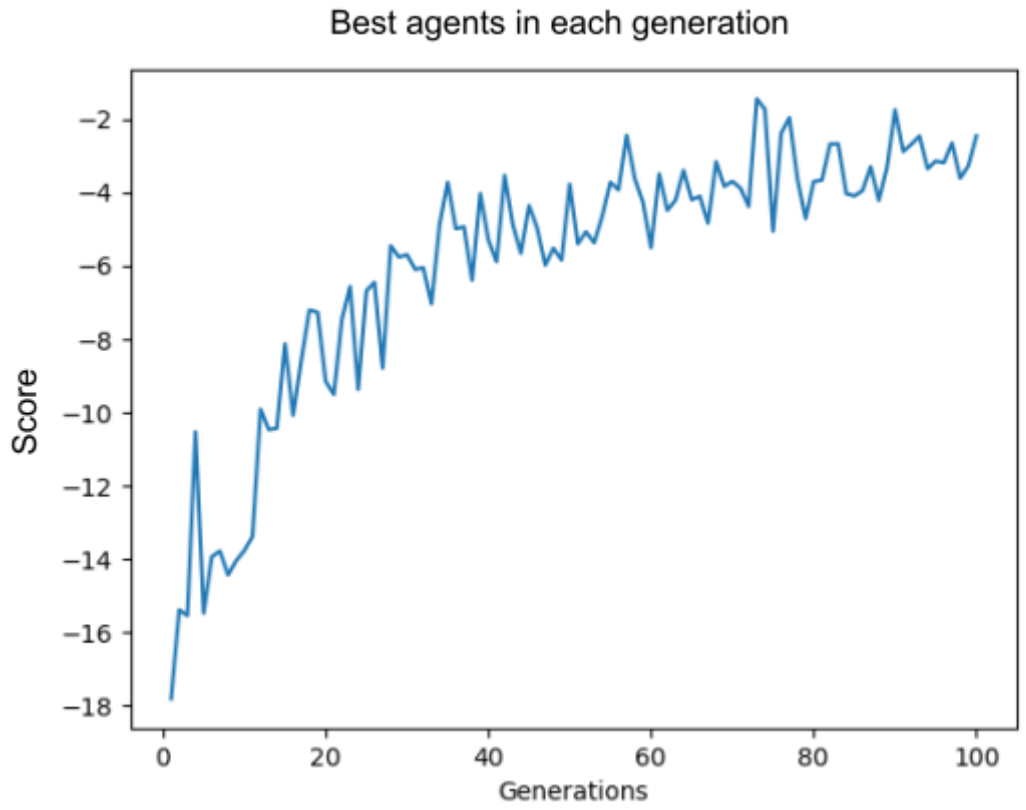

Figure 14: Graph of the score of the GA

As shown on the graph, the best agents in the GA were far worse. With the best agent having a score of around -2, and the best agent in Figure 11 having around 60. Initially, there was a decent amount of growth, but the algorithm plateaued fairly quickly. Once again, changing parameters did not yield any significant improvements (graphs in Appendix), and implementing transfer learning would have been much more difficult in a GA, as the initial population is always initialized randomly.

Although it may seem like training here was more unstable, note that this graph was over a shorter time period and only shows the best agents rather than a moving average.

5 Conclusion

In conclusion, PPO was significantly better than the GA, especially with the usage of transfer learning. However one cannot conclude that every RL algorithm is superior to every GA, since this is only one example and there is a wide variety of algorithms within these fields. Neither can one conclude that PPO is superior to this GA implementation in every environment. Assuming parameters were tuned adequately for both algorithms, the data collected here shows that PPO was superior to this GA implementation in the Bipedal Walker Hardcore Environment.

Unfortunately, neither algorithms were near solving the environment. After observing the leaderboards, I noticed much more complex algorithms were used. Such as Soft Actor-Critics, and a GA implementation using a Covariance Matrix. A more complicated GA than the one implemented here. [16]

However learning completely new algorithms is not necessary for me to improve the results of my programs. One implementation of PPO by a research scientist at Google Brain was able to achieve scores of around 240 to 250.[17] Unfortunately this implementation is not public, so I cannot learn from it, however it demonstrates that there is still significant improvements to be made.

It should be noted that GAs have less libraries and resources for them. TensorFlow is an extremely robust library from Google, with support for many techniques. Meanwhile GAs are significantly less supported, with the best library I was able to find after training the GA being PyGAD.[18] This lack of resources may be a reason as to why the GA was outperformed. Though it could be argued that the reason for RL having more support in the first place is because it tends to work better than GAs.

In conclusion, although these implementations were not successful, they show potential in both RL and GAs. And when more complex algorithms are used, they can have fairly decent outcomes. Furthering this research could involve using a more realistic walking simulation, such as the MuJoCo Humanoid-v2 environment in Gym, which uses a sophisticated physics engine and a walking 3D Humanoid.[19] And hopefully one day techniques such as these will be used in the real world with many varied environments.

Works Cited

1. “Atlas.” *Boston Dynamics*, https://www.bostondynamics.com/atlas.
2. OpenAI. “A Toolkit for Developing and Comparing Reinforcement Learning Algorithms.” *Gym*, http://gym.openai.com/envs/BipedalWalkerHardcore-v2/.
3. Géron, Aurélien. *Hands-on Machine Learning with Scikit-Learn & TENSORFLOW: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2019.
4. Openai. “BipedalWalker v2 · Openai/Gym Wiki.” *GitHub*, https://github.com/openai/gym/wiki/BipedalWalker-v2.
5. Sanderson, Grant. “Neural Networks.” *3Blue1Brown*, https://www.3blue1brown.com/topics/neural-networks.
6. DeepAI. “Stochastic Gradient Descent.” *DeepAI*, 17 May 2019, https://deepai.org/machine-learning-glossary-and-terms/stochastic-gradient-descent.
7. Schulman, John, et al. “Proximal Policy Optimization Algorithms.” *ArXiv*, 28 Aug. 2017.
8. Konda, Vijay R., and John N. Tsitsiklis. “Actor Critic.” *Helen(Mengxin) Ji*, 8 Apr. 2019, https://mengxinji.github.io/Blog/2019-04-08/Actor-Critic/.
9. Salimans, Tim, et al. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning.” *ArXiv*, 7 Sept. 2017.
10. Basak, Setu Kumar. “How to Perform Roulette Wheel and Rank Based Selection in a Genetic Algorithm?” *Medium*, Medium, 5 July 2018, https://setu677.medium.com/how-to-perform-roulette-wheel-and-rank-based-selection-in-a-genetic-algorithm-d0829a37a189.
11. Balsys, Rokas. “BipedalWalker-v3 with Continuous Proximal Policy Optimization.” *Python Lessons*, 23 Nov. 2020, https://pylessons.com/BipedalWalker-v3-PPO/.
12. Versloot, Christian. “ReLU, Sigmoid, Tanh: Activation Functions for Neural Networks.” *MachineCurve*, 4 Sept. 2019, https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/.
13. Versloot, Christian. “Leaky Relu: Improving Traditional Relu.” *MachineCurve*, 15 Oct. 2019, https://www.machinecurve.com/index.php/2019/10/15/leaky-relu-improving-traditional-relu/.
14. Kumar, Siddharth Krishna. “On Weight Initialization in Deep Neural Networks.” *ArXiv*, 4 May 2017.
15. Medeiros, Leandro Couto. *Genetic Algorithm for Robot Planning in the BipedalWalker-v2 Environment*. 12 May 2019, https://github.com/leandrocouto/genetic-bipedal-walker-v2/blob/master/Report_GeneticAlgorithm_Leandro.pdf.
16. Openai. “Leaderboard · Openai/Gym Wiki.” *GitHub*, https://github.com/openai/gym/wiki/Leaderboard.
17. Ha, David. “Evolving Stable Strategies.” 大トロ・*Machine Learning*, 12 Nov. 2017, https://blog.otoro.net/2017/11/12/evolving-stable-strategies/.
18. “PyGAD - Python Genetic Algorithm!.” *PyGAD*, https://pygad.readthedocs.io/en/latest/#.
19. OpenAI. “A Toolkit for Developing and Comparing Reinforcement Learning Algorithms.” *Gym*, https://gym.openai.com/envs/Humanoid-v2/.

Appendix

Reinforcement Learning

15 Trials of the best agent:
https://drive.google.com/file/d/1Fkh79OjCYTDBBK99yY-0J-cURronnaNi/view?usp=sharing

Actor and Critic Architecture:

```
class Actor_Model:
    def __init__(self, input_shape, action_space, lr, optimizer):
        X_input = Input(input_shape)
        self.action_space = action_space
        kernel_initializer = tf.keras.initializers.HeNormal()
        activation = tf.keras.layers.LeakyReLU(alpha=0.01)

        X = Dense(512, activation=activation, kernel_initializer=kernel_initializer)(X_input)
        X = Dense(256, activation=activation, kernel_initializer=kernel_initializer)(X)
        X = Dense(128, activation=activation, kernel_initializer=kernel_initializer)(X)
        X = Dense(64, activation=activation, kernel_initializer=kernel_initializer)(X)
        X = Dense(32, activation=activation, kernel_initializer=kernel_initializer)(X)
        output = Dense(self.action_space, activation="tanh")(X)

        self.Actor = Model(inputs = X_input, outputs = output)
        self.Actor.compile(loss=self.ppo_loss_continuous, optimizer=optimizer(lr=lr))
```

PPO Loss:

```
def ppo_loss_continuous(self, y_true, y_pred):
    advantages, actions, logp_old_ph, = y_true[:, :1], y_true[:, 1:1+self.action_space], y_true[:, 1+self.action_space]
    LOSS_CLIPPING = 0.2
    logp = self.gaussian_likelihood(actions, y_pred)

    ratio = K.exp(logp - logp_old_ph)

    p1 = ratio * advantages
    p2 = tf.where(advantages > 0, (1.0 + LOSS_CLIPPING)*advantages, (1.0 - LOSS_CLIPPING)*advantages) # minimum advantage

    actor_loss = -K.mean(K.minimum(p1, p2))

    return actor_loss
```

Discounted Rewards:

```
def discount_rewards(self, reward):
    # Compute the gamma-discounted rewards over an episode
    # We apply the discount and normalize it to avoid big variability of rewards
    gamma = 0.99    # discount rate
    running_add = 0
    discounted_r = np.zeros_like(reward)
    for i in reversed(range(0,len(reward))):
        running_add = running_add * gamma + reward[i]
        discounted_r[i] = running_add

    discounted_r -= np.mean(discounted_r) # normalizing the result
    discounted_r /= (np.std(discounted_r) + 1e-8) # divide by standard deviation
    return discounted_r
```

Genetic Algorithms

Ranked Selection:

```python
def rank_prob(R, N):
    denominator = 0
    for r in range(N):
        denominator += (1-(r/N))
    return (1-(R/N))/denominator

def calculate_rank(self):
    self.fitness_values = np.array(self.fitness_values)
    self.fitness_values = np.sort(self.fitness_values)
    self.fitness_values = self.fitness_values[::-1]

    ranks = np.empty(len(self.fitness_values))

    for i in range(len(self.fitness_values)):
        ranks[i] = i

    final = np.array([ranks, self.fitness_values]) # 1 - n, self.chromosomes, first row ranks

    probs = []

    for i in final[0]:
        probs.append(rank_prob(i, len(final[0])))
    final = np.vstack((probs, ranks))

    for i in range(len(probs)):
        self.chromosomes[i].set_selection_probability(probs[i])
```

Crossover:

```python
def crossover_1(self, N_ACTIONS, elite):
    new_population = []
    new_population.extend(elite)
    #Ignoring the elite, iterate until the end of the chromosomes

    for i in range(len(elite), len(self.chromosomes), 2):
        #Selection stage - Select two chromosomes for crossover (elite CAN be selected)
        index_1, index_2 = self.sample_chromosome_pairs()
        #Crossover
        new_chromosome_1 = self.chromosomes[index_1].crossover_between_chromosomes(self.chromosomes[index_2], N_ACTIONS)
        new_chromosome_2 = self.chromosomes[index_2].crossover_between_chromosomes(self.chromosomes[index_1], N_ACTIONS)

        new_population.append(new_chromosome_1)
        new_population.append(new_chromosome_2)
    self.chromosomes = copy.deepcopy(new_population)
```

Mutation:

```python
def mutation(self, N_ACTIONS,MUTATION_PERCENTAGE, N_MUTATIONS):
    for chromosome in self.chromosomes:
        will_it_mutate = random.randint(0, 100)
        if will_it_mutate >= MUTATION_PERCENTAGE*100:
            continue
        #generate the action indexes to be fully mutated
        mutation_indexes = random.sample(range(N_ACTIONS), N_MUTATIONS)
        for m in mutation_indexes:
            chromosome.actions[m] = env.action_space.sample()
```

Person Code: hqy392

Using different parameters for Genetic Algorithm Results: