OpenAI Bipedal Walker

by Derek Brenner and Sidney Lafontaine

Introduction

Motivation

OpenAI's Bipedal Walker problem consists of a virtual environment that contains a bipedal robot and terrain for the robot to learn to walk on. The bipedal walker contains four components, the first of which is the hull - or the head of the robot. When the walker is learning to walk, it considers itself to have fallen whenever the hull touches any part of the terrain. Attached to the hull of the walker is a lidar, which can be angled by the walker to determine how far away the terrain is from the source of the lidar. Besides these components, the walker has its two joints/legs that can be manipulated through an array of four numbers each.

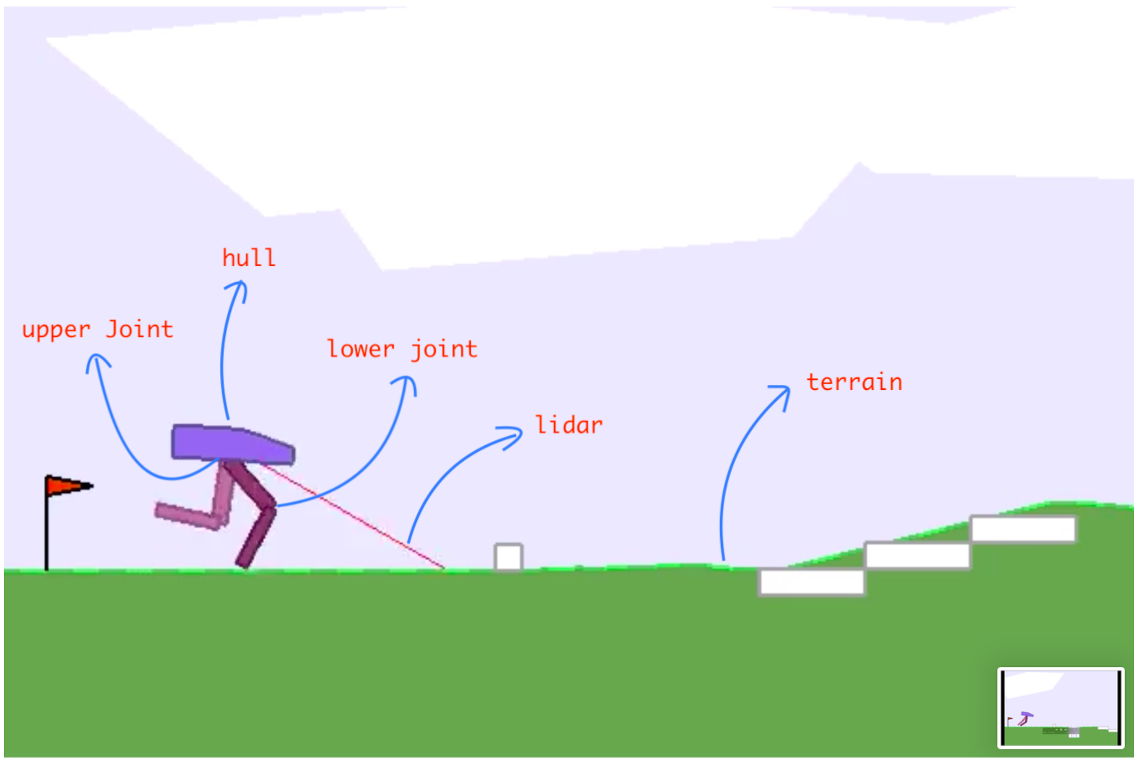


Figure 1: Components of Bipedal Walker Environment

In terms of the metrics that are used to measure the success of the walker during each episode, the environment is based on a point system where the AI algorithms attempt to maximize these points. The walker is rewarded with up to +300 points if it reaches the end of the level. If the robot ever falls, which is characterized by the hull touching the ground, it gets -100 points. There is no punishment for the duration of the robot being alive, and whenever the joints of the walker apply torque, it loses a fraction of a point.

This problem first grabbed our attention because it could be adapted to and applied to real-life situations. While the environment is only two-dimensional, the methods and implementations could be applicable when adapted to a three-dimensional environment. Another interesting aspect of the problem is that although the user can see the entire environment, the robot is only aware of its speed, angular velocity, and lidar rangefinder measurements, making it an accurate representation of a real-world problem where the walker may not have perfect information - only what it can observe itself.

Another reason this project interested us is that the bipedal walker has an infinite state space, as opposed to discrete state space. What is unique about this problem is that unlike other applications of artificial intelligence, there may not be an existing optimal solution due to the infinite possible policies. Even though there are existing solutions that implement various AI algorithms there may be ways to continue to improve upon them.

Challenges

The most challenging aspect of this problem is the infinite state space for every move. There are literally infinite possible ways for the bipedal walker to attempt to move at any given moment. In many other AI problems such as Blackjack, Poker, or Chess the agent has a very limited number of possible moves at each moment, therefore, allowing the algorithm to find a suitable or ideal move from a discrete set of possible moves. So our challenge was to find a way to determine the best possible move while considering the fewest of the infinite possible moves because there simply is not enough computing power in the world to consider infinite moves even if time wasn't a factor.

We solved the infinite state space problem by leveraging data to find the ideal move in any situation, which brings us to our next big challenge which is a complete lack of publicly available data for this specific problem. Therefore, we had to use reinforcement learning algorithms that could learn by collecting data while they trained to get better results in future episodes.

Another huge challenge was the limited amount of data available to the bipedal walker. While the user can see the entire environment and terrain at any given point during the simulation, the walker is not able to see this. The robot has to learn not only to move its legs in a way that it can stay upright and walk but also to use the lidar configurations to understand the terrain. Some levels are easier than others, but no level is completely flat, making it more difficult for the bipedal walker to move forward.

Furthermore, the state representation also posed a challenge to using many reinforcement learning algorithms that require a more general state to learn from. The state for the bipedal walker consists of several factors, such as hull angle speed, angular velocity, horizontal speed, vertical speed, the position of joints and joints' angular speed, legs contact with the ground, and 10 lidar rangefinder measurements. For certain learning algorithms, such as Monte Carlo learning or Direct Evaluation, the given state representation is far too specific to learn from, so we had to convert the state into an alternative state representation that is more general. Also, the reward results of state-action pairs were hard to store considering how large each state representation is at every frame. So, a more general state representation helped with the computational complexity and storage challenges we faced.

Overview

While tackling this task with a few different algorithms, we had to find ways to overcome each of these challenges in each of those approaches. The three algorithms we implemented to solve the bipedal walker problem include Monte Carlo Learning, Q-Learning, and TD3 (Twin Delayed Deep Deterministic Policy Gradient). After implementing each of these algorithms, we learned that some of the challenges presented in the problem could not be easily solved with all of the algorithms. We saw unimpressive results with Monte Carlo and Q-Learning, while TD3 saw great success in training the bipedal walker to walk across the terrain to the end of the level.

Related Work

Since the bipedal walker is an OpenAI Project, there are currently existing solutions to the problem. The most successful solutions to this problem include some form of deep learning, such as deep q-learning or LSTMs. We decided to go a different direction with the project, by not only utilizing a form of deep learning in our TD3 approach but using other methods such as Monte Carlo learning that have not been otherwise implemented for the bipedal walker problem to see if those would yield any positive results.

During our research phase, we viewed several approaches that others have tried implementing. The first that we looked into was utilizing various forms of machine learning. An extensive research paper by Levi Fussell explores several of these machine learning methods. Interesting algorithms explored in this paper included the REINFORCE Algorithm, Evolutionary Strategies Algorithms, and Cross-Entropy. The first, REINFORCE, is an unbiased on-policy algorithm that utilizes a Monte Carlo sampling of expected return to approximate a return over the samples. This includes a supervised learning algorithm over a dataset as the bipedal walker explores the environment, and uses a stochastic policy function - one that has to be continuous and differentiable for all values - to determine each action as a vector of continuous values. This algorithm performed the worst out of all the discussed algorithms.

The evolutionary strategies used a gradient ascent approximation (because they are trying to maximize reward) along with replication, variation, and competition to determine which parts of the algorithms would be passed down. In the bipedal walker's case, it would pass down a list of neural networks and calculate the scores of each of these neural networks, which represent each bipedal walker agent. Fussell then created a new generation using a mutation rate of 0.01

using the highest performing agents. This method worked very well, but only as long as small models were used for the policy function. The Cross-Entropy Method samples independent distributions over individual parameters and uses hill-climbing methods to improve on its parameters. This method was able to complete the task effectively and efficiently but only when smaller models were used for the policy function, similarly to the evolutionary strategies method. (Fussel, 2018)

We also found several GitHub repositories attempting to implement solutions to the OpenAI problem. These included A Distributional Perspective on Reinforcement Learning (Parilo), Deep-Q Learning Networks, Recurrent Deterministic Policy Gradient Methods, Actor-Critic Networks, and Proximal Policy Optimization Methods (Kyziridis).

We only dove deep into the research papers and Github repositories that are mentioned above, but several other developers have attempted similar methods to find an optimal solution to the Bipedal Walker problem. While all these methods intrigued us, most were out of the scope of our knowledge and ability to implement, and so we adapted several ideas from our research findings into our implementation.

Approach

After researching alternative approaches to the bipedal walker problem, we concluded that we wanted to implement three different learning algorithms: Monte Carlo Learning, Q-Learning, and TD3 (Twin Delayed Deep Deterministic Policy Gradient).

Control: Random Movement

To have something to compare and evaluate our approaches against, we used a control group of randomly sampled actions within the action space. We ran 1000 episodes and took the average final reward from all the episodes as well as the maximum final reward from each episode.

Monte Carlo

We decided to implement a Monte Carlo learning algorithm because although we did not see any documentation for other people trying this method, we thought it would be a good place to start the training. The thought process behind this learning method is that the bipedal walker would hopefully start to understand what positions would be rewarded more, and might learn how to stay upright to avoid losing -100 reward, or even learn how to walk forward to get the +300 reward at the end of the level. In this approach, we ran 1000 randomly simulated episodes of the bipedal walker, all while storing the training data as state-action pairs to determine which actions at which states would yield better rewards. We then stored the data as a reference when running the simulation another 100 times to see if the bipedal walker had learned from the randomly generated episodes.

Unfortunately, we saw unimpressive results with our Monte Carlo learning algorithm, with an average reward of -99.9856 and a maximum reward over 1000 episodes of -0.3238. While we were implementing this approach, we quickly realized the complexity of handling an infinite state space, and this would lead to the bipedal walker only understanding how to act with extremely specific state-action pairs. Furthermore, the Monte Carlo learning algorithm would only work on the first few frames, while the state space starts the exact same every time. After the first hundred episodes, the bipedal walker would lunge forward to maximize the initial reward of moving forward - with no regard for falling. Only a few frames later would the bipedal walker either fall forward due to its momentum or fall backward again, but never move more than that. For Monte Carlo learning to work, we would need to run an infinite number of episodes to determine the best action in any state space, but this is unrealistic and most certainly not the most effective way to train the bipedal walker. At the end of our 1000 episodes of testing, the bipedal walker only knew the results of one action from each state in the continuous state space, which is not nearly enough to learn how to walk.

Q-Learning

We chose to implement Q-Learning because it is an algorithm we are familiar with, and has presented decent results in similar tasks. The point of this algorithm is to have the value-based algorithms update the value function based on the Bellman equation. Instead of having to simulate through millions of iterations and episodes like we would have had to with Monte Carlo Learning, Q-Learning would hopefully learn the value of the optimal policy while the episodes were simulating.

In our particular implementation of Q-Learning, we realized after a few hours of countless errors that we needed a way to represent the state space, as also mentioned earlier in the paper as a major challenge. We found one source that uniquely tackled this problem - by dividing the state space into various “buckets”. We utilized this bucket method to allow our bipedal walker to have a better understanding of the state space, and have a way to store action-value pairs in an easier way (Hifly). Then, utilizing the equations we learned in classes, we created a function that would update the value of Q(state, action), when given the learning rate, discount rate, the environment (in the form of the current state, action, and reward).

We then had the q-learning method run 1000 episodes to see how it performed, while updating every 10 episodes to track performance. Unfortunately, we did not see the results we were hoping for, as after about the first 100 episodes, the results plateaued at an average score of about -95. After viewing the simulations at each multiple of 100, it was clear that the bipedal walker was prioritizing staying alive more than anything, and so it would stick both of its legs out and shimmy forward extremely slowly until the time ran out on the simulation or the bipedal walker reached a state where it learned to fling itself forward.

As seen in the evaluation section, we did not succeed with the implementation of Q-learning, most likely because the infinite state space makes it extremely difficult to allow the bipedal walker to learn. Our attempt to make this state-space discrete and finite with the use of buckets failed to produce positive results, and so we realized we needed to utilize deep learning to effectively handle the infinite state space. Q-learning only works in environments with discrete and finite state and action spaces, and so it would not be possible to implement without finding a better and more effective way to turn the infinite state and action space into a discrete space.

TD3

While we researched potential algorithms to implement for this project we quickly realized to effectively handle such a demanding problem we needed to utilize deep learning. After looking into several algorithms that leveraged the power of deep learning we decided to focus on implementing TD3 because we saw it had been effective at solving this type of problem in similar environments. Also, while it does utilize deep learning, it is based on Q-Learning, an algorithm that we are very familiar with.

TD3 is a Double Q-Learning algorithm that is based on an actor-critic model, specifically with two critics. Each agent, the actor, and both critics are represented by their own feed-forward neural network. Each agent's network consists of two hidden layers, the first has 400 nodes, the second has 300 nodes, with the input and output layers having the same dimensions as the state of the Bipedal Walker. Both the actor and critic neural networks use Rectifier (ReLU) function as the activation function for the hidden layers. The Actor uses a hyperbolic tangent function (tanh) as the final activation function, to keep the output within the [-1,1] state space for every move, and the critics simply do not apply an activation function to the final layer (Fujimoto, Van Hoof, and Meger).

Rectifier Equation

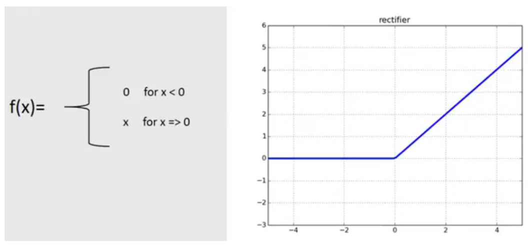


Figure 2: Rectifier equation and graph

Tanh Equation

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

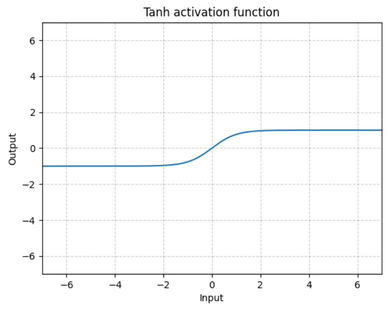


Figure 3: Tanh equation and graph

Once enough data has been collected each critic neural network is re-trained at the end of each episode. Then the minimum loss value, using mean squared error, of the two critic models is used to train the actor model (the model that actually decides the movements of the walker). Using the minimum of two critics is key to limiting overestimation bias which is consistently a problem with single Q-Learning algorithms (Fujimoto, Van Hoof, and Meger).

$$y = \text{reward} + \gamma Q_{\theta'} (s', \pi_{\varphi}(s')).$$

$$y_1 = \text{reward} + \gamma Q_{\theta'2} (s', \pi_{\varphi 1} (s'))$$

$$y_2 = \text{reward} + \gamma Q_{\theta'1} (s', \pi_{\varphi 2} (s')).$$

$$y1 = \text{reward} + \gamma \min_{i=1,2} Q_{\theta' i} (s', \pi_{\varphi 1} (s')).$$

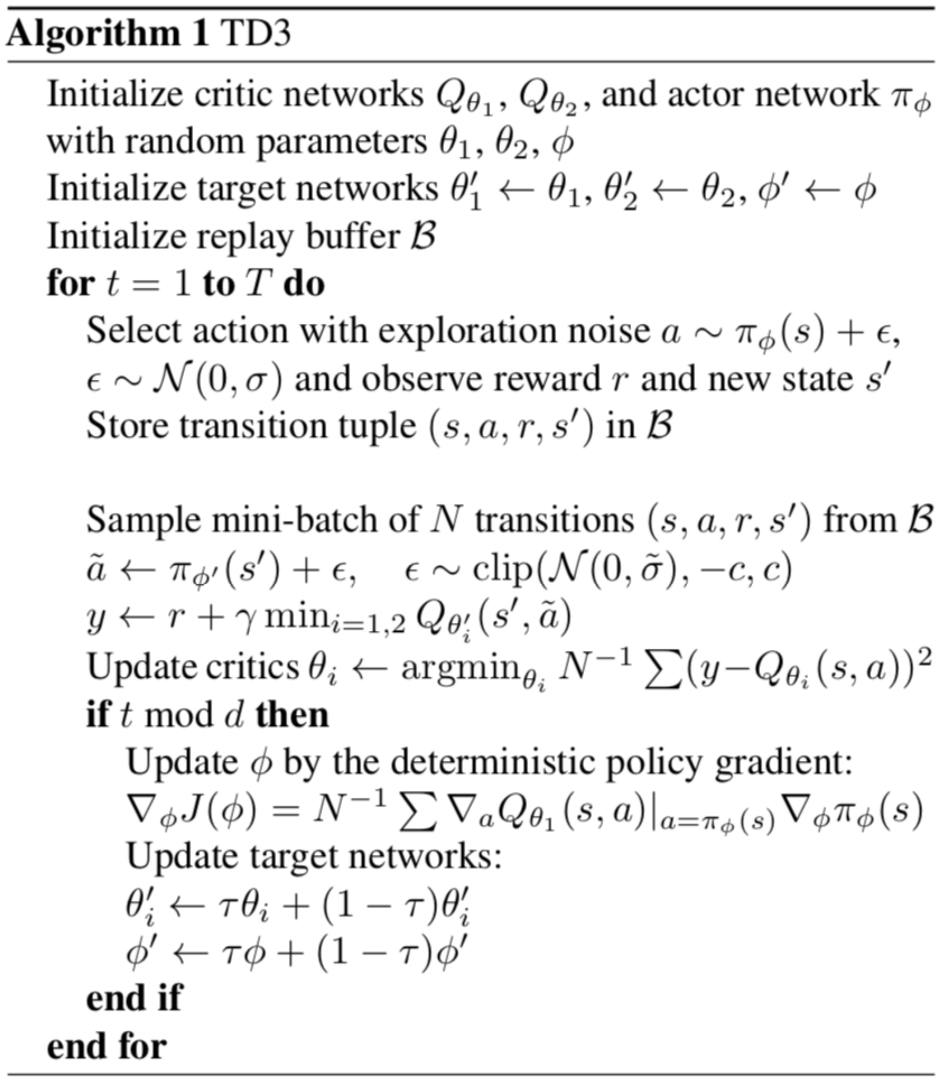
An important aspect of this algorithm is the addition of noise and state-space restrictions to every move. After any agents' neural network determines the optimal action, noise is added to the action by adding a randomly generated number from a normal distribution with the range (0, σ). This noise encourages exploration, which will increase model performance by attempting new and untried moves that could be very rewarding (Fujimoto, Van Hoof, and Meger). Finally, before a move is made it is "clipped" meaning it is limited to the range [-1,1], (if an action is > 1 or <-1 it is simply replaced with 1 or -1 respectively) the state space for every move so that the move is valid.

$$y = \text{reward} + \gamma Q_{\theta'} (s', \pi_{\varphi'}(s')+\varepsilon),$$
$$\varepsilon \sim \text{clip}(N (0, \sigma), -1, 1)$$

Another key element to my implementation of TD3 is using a replay buffer that stores all the state, action, and reward pairs. Instead of using all available data to retrain each agents' neural network at the end of every episode we used a random sample of 100 state, action, and reward pairs from the set of all collected state, action, and reward pairs. This approach has its drawbacks, such as the randomly generated sample could be wildly misrepresentative, but it allowed us to train the algorithm much faster. In the end, this decision was necessary due to the limited computational power available to our group for this project.

At the end of every episode the we re-train (update the policies) all the agents' neural networks. But it is important to note that while the critic agents' neural networks are re-trained at every timestep in the previous episode, the actor agent's neural network is not. Instead the actor's neural network is re-trained periodically, based on a specific frequency (in our implementation it was every other timestep). This helps handle the computational complexity of training and re-training multiple neural networks so many times, but its primary purpose is to make sure the actor, which is trained using the minimum of the critics' losses, isn't repeatedly re-trained when the critics' haven't changed (Fujimoto, Van Hoof, and Meger). Timesteps for the Bipedal Walker are so short that it usually takes multiple timesteps for any substantial change in the state.

Training this algorithm was very computational intense, especially for the laptops we had available. So we trained it in 50 episode sessions, even though the original paper recommended 100 episode sessions because it was more manageable for my computer. For training the algorithm we used the specific values for variables provided in the paper for the original implementation (Fujimoto, Van Hoof, and Meger).



```
Algorithm 1 TD3
  Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$
  with random parameters $\theta_1$, $\theta_2$, $\phi$
  Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
  Initialize replay buffer $\mathcal{B}$
  for $t = 1$ to $T$ do
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
    $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
    Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$

    Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
    $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
    Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if $t$ mod $d$ then
      Update $\phi$ by the deterministic policy gradient:
      $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
      Update target networks:
      $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
      $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
    end if
  end for
```

Figure 4: TD3 pseudo-code from Addressing Function Approximation Error in Actor-Critic Methods (Fujimoto, Van Hoof, and Meger)

Evaluation/Results/Testing

Throughout our evaluation/results/testing we largely relied upon the points system provided with this environment that we previously explained.

Control: Random Movement

Randomly Generated Movement	
Average Reward (over 1000 episodes)	-102.9248
Maximum Reward (over 1000 episodes)	-0.3012

Table 1: Randomly Generated Movement Results

Monte Carlo Learning

Monte Carlo Learning	
Average Reward (over 1000 episodes)	-97.9856
Maximum Reward (over 1000 episodes)	-0.3238

Table 2: Monte Carlo Learning Results

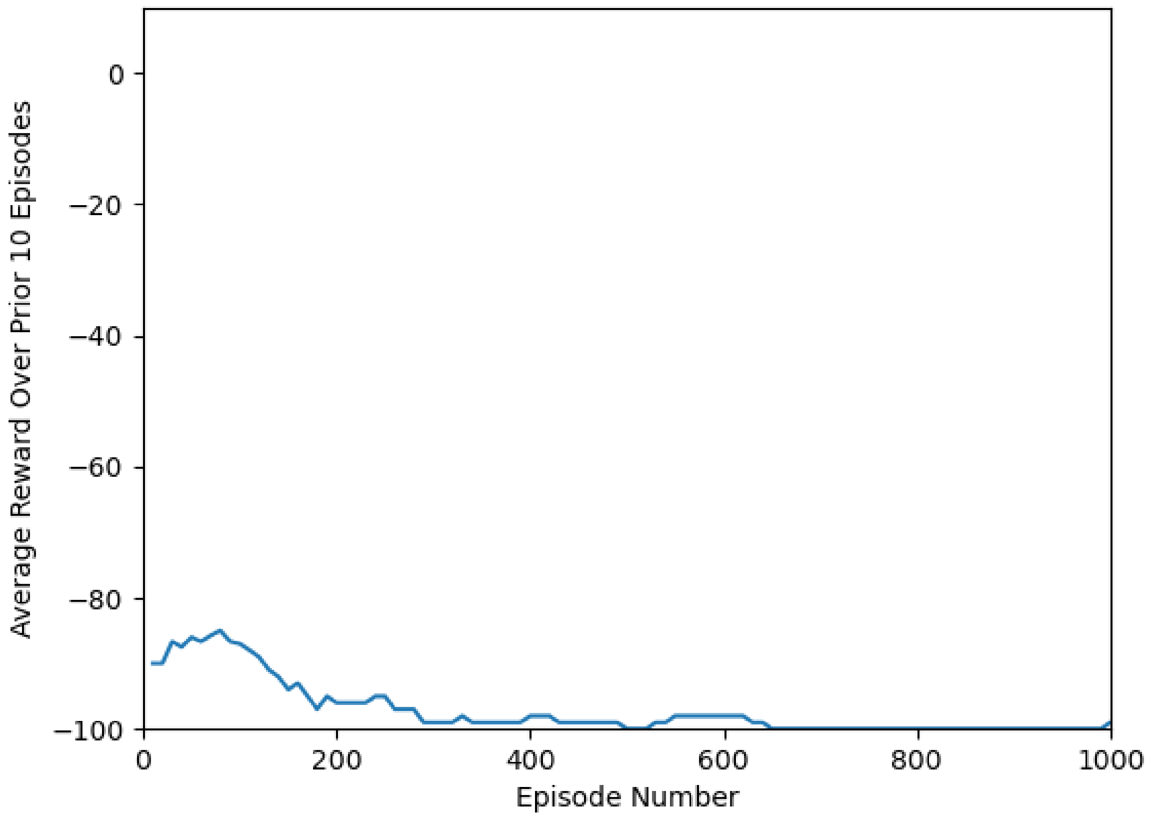


Figure 5: Line chart depicting average score over past 10 episodes of Monte-Carlo model after every 10 episodes of training

Q-Learning

Q-Learning	
Average Reward (over 1000 episodes)	-79.1221
Maximum Reward (over 1000 episodes)	0.1280

Table 3: Q-Learning Results

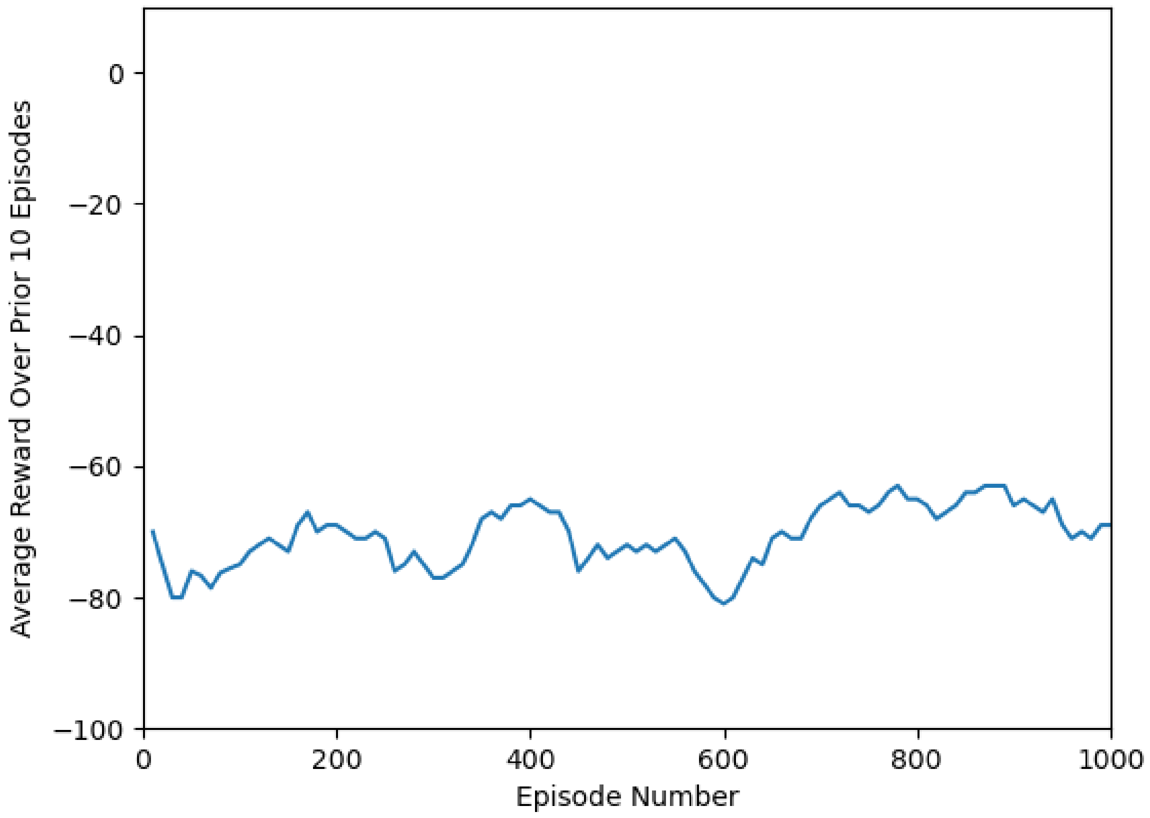


Figure 6: Line chart depicting average score over past 10 episodes of Q-learning model after every 10 episodes of training

TD3

TD3	
Average Reward (over 610 episodes)	-74.587945
Maximum Reward (over 610 episodes)	277.41269438128

Table 4: TD3 training results

Due to computational power and time constraints we only trained TD3 on 610 episodes, but by the end it was consistently scoring greater than 100 points and completing over 80% of levels (gets to the end without falling). Our final model is not fully optimal, it applies more torque than necessary and it falls on less than 20% of levels, but it is largely successful. In fact, the final model averaged 130.91474 points over 25 episodes. We are sure the model would continue to improve with more training, and could even become optimal (>300 points per episode on average).

Number of Episodes Trained	Average Score over 25 Episodes
50	-99.96412781
100	-102.7370097
150	-114.7822906
200	-115.5708295
250	-119.5300266
300	-121.6169756
350	-112.5445447
400	-58.61541779
450	-105.180971
500	-57.70548131
550	-68.11394729
600	82.52083344
610	130.9147465

Table 5: average score over 25 episodes of TD3 model after every 50 episodes of training

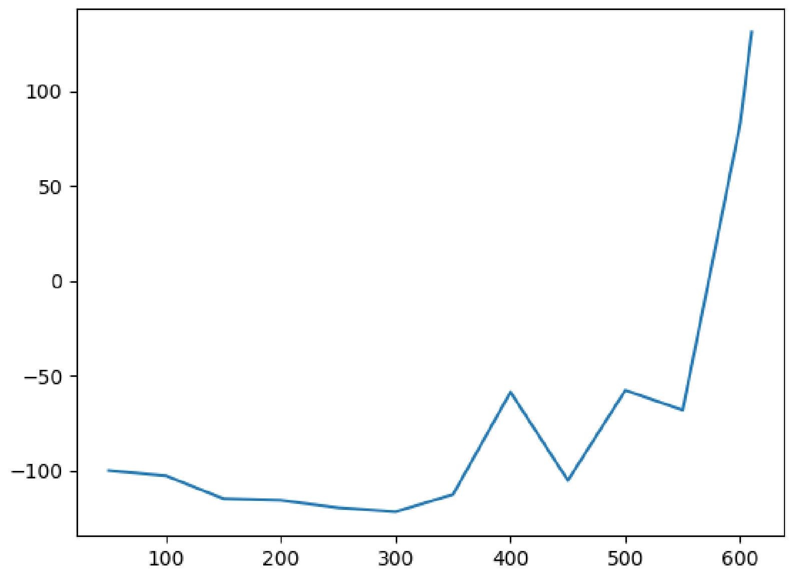


Figure 7: Line chart depicting average score over 25 episodes of TD3 model after every 50 episodes of training

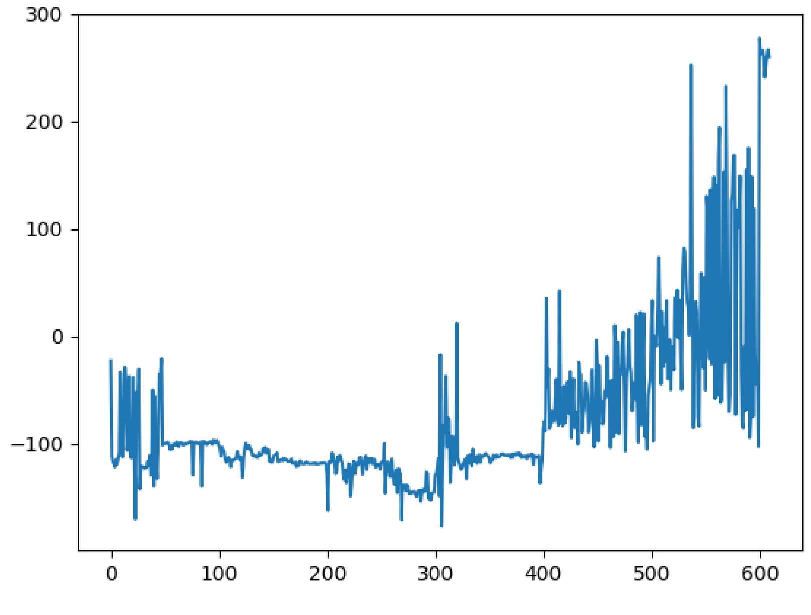


Figure 8: Line chart depicting episode score for each episode throughout 610 training episodes for TD3

Conclusions and Future work

While the Q-Learning and Monte Carlo approaches were unsuccessful, TD3 was very successful, on average 130.91474 points per episode over 25 episodes. By the end of training we had successfully taught a bipedal robot to walk, which is a pretty amazing achievement. We are ecstatic with the results we got given the many limitations we faced such as, lack of computational power, the problem's infinite state space, lack of information available to the bipedal walker, no publicly available data for this problem, etc.

Our main conclusion is that to successfully solve such a complicated problem without any publicly available data you need to leverage the power of deep learning. There are a few different algorithmic approaches to leverage the power of deep learning that we believe could be successful on this problem including: Deep Q-Learning, LSTMs, and Genetic Algorithms, as stated in our research in the related works section. We choose to implement a Deep Q-Learning algorithm, but the other two options promise to be similarly successful. The genetic algorithm approach where the population is a list of neural networks and the propagation occurs by mutating weights and biases of successful neural networks would be particularly interesting.

Acknowledgements

Project members: Derek Brenner and Sidney La Fontaine

We largely worked together helping each other throughout this project. Derek Brenner was largely responsible for implementing and testing the Q-Learning and Monte Carlo approaches and Sidney La Fontaine was largely responsible for implementing and testing the TD3 approach. But we want to emphasize we worked together on all of these approaches.

We also want to acknowledge Professor Marsella for not only introducing us to several of the methods and algorithms we implemented in class, but for also suggesting alternative approaches and helping us through the process.

References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *ArXiv Preprint ArXiv:1606.01540*.

Fussell, Levi. (2018). *Exploring bipedal walking through machine learning techniques*. [BS Dissertation, The University of Edinburgh].

Hifly, Claude. “Data-Driven Project Based on BipedalWalker-v3.” GitHub, https://github.com/claudeHifly/BipedalWalker-v3

Kyziridis. “OpenAI Bipedal Walker Deep Reinforcement Learning.” GitHub, https://github.com/Kyziridis/BipedalWalker-v2.

Scott Fujimoto, Herke Van Hoof, David Meger, 2018, *Addressing Function Approximation Error in Actor-Critic Methods*, arXiv, 1802.09477.

Parilo. “A Distributional Perspective on Reinforcement Learning.” GitHub, https://github.com/parilo/gym_bipedal_walker_v2_solution.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf