CS 534 Final Project Writeup

Neil Kale, Keshav Bimbraw
(GitHub Repository - https://github.com/bimbraw/RL_PyTorch)

Introduction

Project Goals

In the final project, our team wanted to learn more about applying reinforcement learning to physical systems. We were curious about the OpenAI Gymnasium toolkit for simulations and the recently developed Deep Q-Network algorithm (2015) which we did not study in detail in class.

Learning Environment

To best achieve these goals, our team decided to tackle the Cart-Pole-v1 simulation provided in OpenAI Gym.[1] The objective of the Cart-Pole simulation is to keep a vertical pole on a rolling cart from falling over.

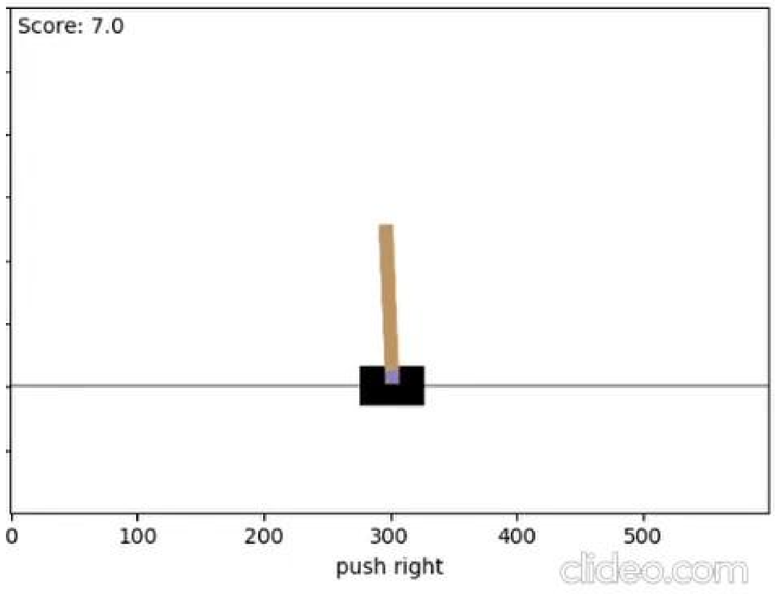


Figure 1. OpenAI Gym provides a graphic interface for the simulation. The pole (brown) and cart (black) positions update at each time step. Our team added time-alive in the top left and the most-recent move at the bottom. For a video demonstration, see https://youtu.be/UjcjXh7-Qp0.

[1] Cart Pole - Gymnasium Documentation (farama.org)

The state space consists of 4 variables: cart position, cart velocity, pole position, and pole velocity. The variables are each initialized to a small random value from -0.05 to 0.05. At each state, the agent chooses from pushing the cart left or right. It must apply a push and the force is fixed.

The agent receives a +1 reward for each timestep that the pole remains standing and the cart remains on the graphic interface. If the pole angle passes 22.5 degrees (the pole falls) or the cart position exceeds 4.8 units (note that the x-axis on the graphic interface is not the same), the simulation terminates. The simulation will also terminate if the pole remains standing for 500 moves, so the maximum possible reward is 500.

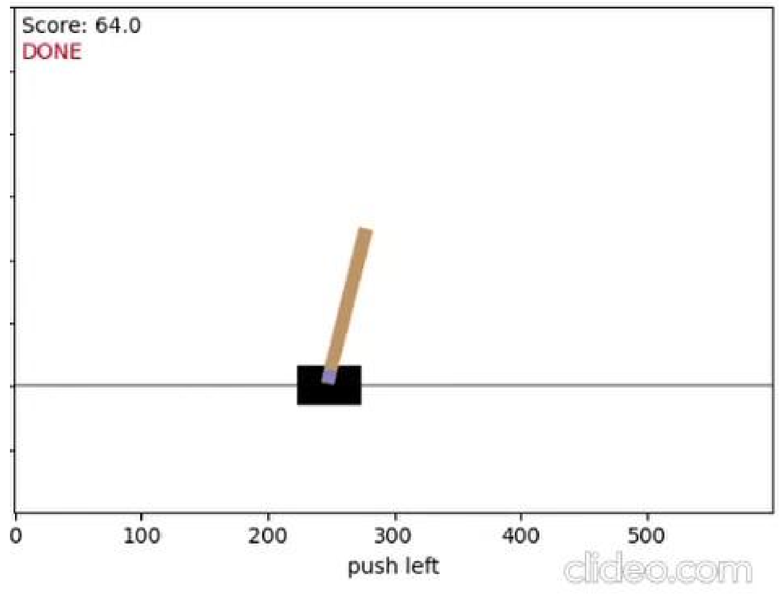


Figure 2. In this frame, the pole has fallen past 22.5 degrees so the simulation is "DONE." The interface indicates that the pole remained standing for 64 timesteps and the last move that the agent made was pushing left.

Simple Agents

To establish a baseline performance, our team first explored some simple agents that use hard-coded rules to decide their next move.

Always-Push-Left Agent

Our first agent always pushes the cart left. Over 2000 evaluation episodes, it achieves a mean reward of 9.3. This tells us that on average, the pole begins 9-10 timesteps from falling over.

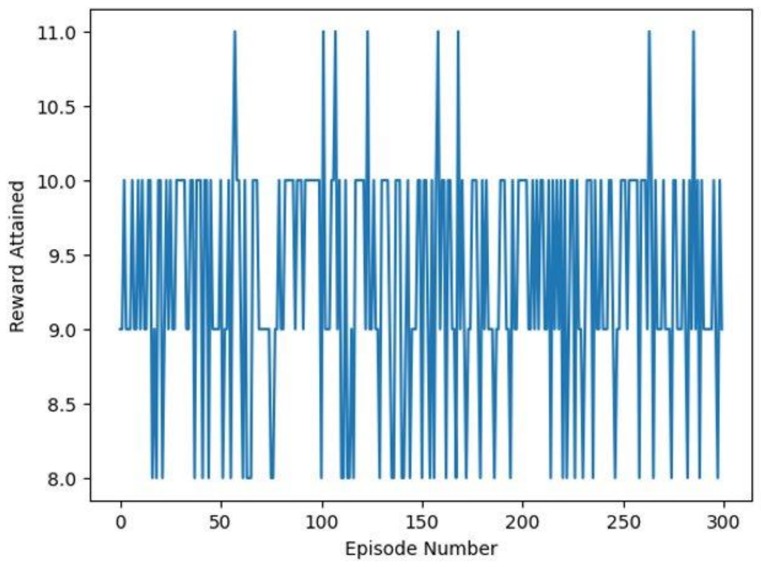


Figure 3. The graph above illustrates reward attained for each "training" episode. Note that this agent does not actually learn anything so the performance does not improve with training. Similar graphs will appear throughout this paper.

Push-in-Direction-of-Fall Agent

Our second attempt pushes the cart in the direction that the pole is currently falling. Over 2000 evaluation episodes, it achieves a mean reward of 28.2. This agent acts how a human would respond intuitively to the falling pole. So we can roughly estimate that if this simulation were created in real life, a human might keep the pole up for about 28 timesteps. A better prediction would be to actually implement human interaction (i.e. left-right arrow keys as input) and allow people to play the simulation.

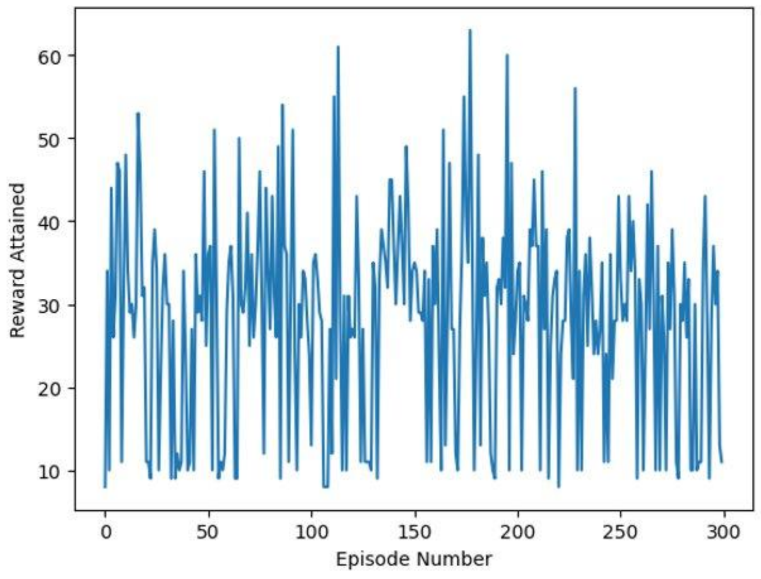


Figure 4. This agent performs much better than the agent that always pushes left. However, it also does not learn anything. Once again, the performance does not improve with "training."

A Note on Evaluation Metrics

Throughout this paper, models will always be evaluated on 2000 training episodes. To be concise, we will omit this in the rest of the paper.

Q-Learning

Our team next attempted to develop a Q-learning agent that performed better than the push-in-direction-of-fall agent. Q-learning works on a discrete state space. But the positions and velocities in the Cart-Pole simulation are continuous. Before attempting Q-learning, we had to discretize the Cart-Pole state space. See the appendix for details about discretization.

Implementing Q-Learning

We now train and fine-tune a Q-learning agent that can learn to balance the Cart-Pole simulation. The agent maintains a [30 x 30 x 50 x 50 x 2] table of state-action pairs. At each time step, it takes an action and updates the table according to the Bellman Equation. We omit the details here, but importantly, we have three model hyperparameters which are discount, learning rate, and epsilon (which balances exploitation and exploration). In addition, we can vary the number of training episodes and the maximum length of training episodes.

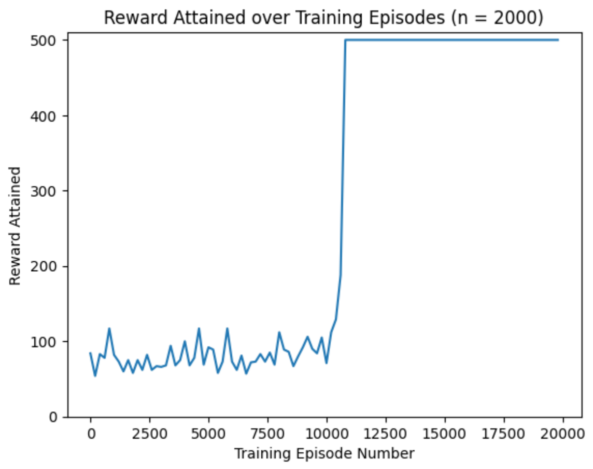
Base Case

Our first Q-learning agent has hyperparameters as listed below.

Table 1. Hyperparameters for the base case.

Discount	0.95	**Learning rate**	0.1
Training episodes	20000	**Training episode length**	500
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 10000. Decay constant = 0.9995.		

The agent performs extremely well, achieving the maximum possible reward of 500 on over 90% of the evaluation episodes. The mean reward in testing episodes is 470.8.



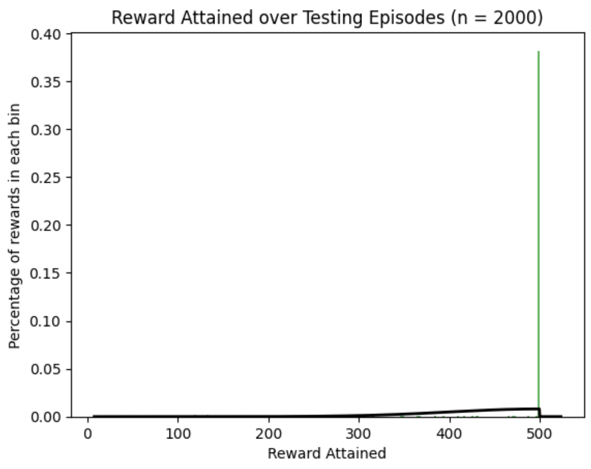


Figure 5. The graph at the left tracks the reward during training. The agent spends the first half of training making random moves. After 10,000 steps, the agent begins exploiting and learns to balance the Cart-Pole. The graph at right shows a histogram of the reward attained in testing episodes.

Reducing the Training Episode Length

Since training the original agent takes very long, we attempt to accelerate the training process by reducing the maximum training episode length to 100 steps.

Table 2. Hyperparameters for 'Reducing the Training Episode Length.'

Discount	0.95	**Learning rate**	0.1
Training episodes	20000	**Training episode length**	100
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 10000. Decay constant = 0.9995.		

Remarkably, the agent performs even better, achieving a perfect reward of 500 in over 95% of testing episodes. The mean reward in testing episodes jumps from 470.8 to 481.2. This improvement might be because when the training episode length is reduced, the agent spends a larger part of each episode in a new random situation.

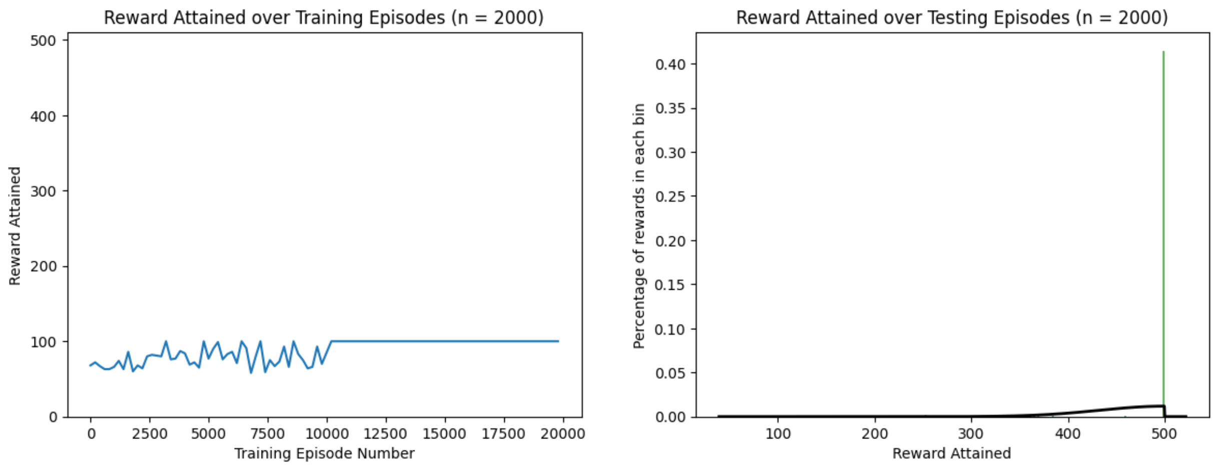


Figure 6. This agent achieves a reward of a maximum of 100 in training but performs better in testing than the original agent.

Reducing the Number of Training Iterations

Since the Q-learning agent performs so well with 20,000 episodes of training, it is difficult to study the impact of hyperparameter optimization there. Instead, we shorten the training time to 2,000 episodes. We want to create a less perfect agent to study.

Table 3. Hyperparameters for Reducing the Number of Training Iterations.'

Discount	0.95	**Learning rate**	0.1
Training episodes	2000	**Training episode length**	100
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1000. Decay constant = 0.9995.		

As desired, this agent performs worse than the original Q-learning agent. The mean reward in testing episodes drops to 30.8.

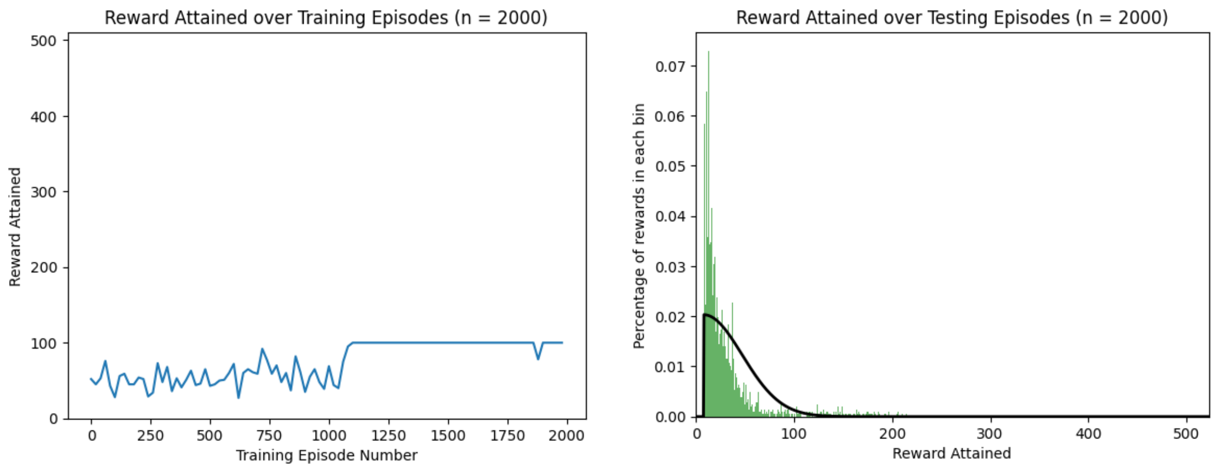


Figure 7. The histogram at right is finally useful for this agent. The testing episode rewards seem to follow a skew-left distribution with some as high as 200, but mostly below 50.

Increasing the Training Episode Length

The previous agent is limited to a reward of 100 in training. As seen in Figure 7a, it reaches this cap almost immediate after it begins exploiting. Since we have already reduced the training time by reducing the number of training episodes, we can now explore whether this reward cap is preventing it from learning further. We reincrease the training episode length to 200.

Table 4. Hyperparameters for Reducing the Number of Training Iterations.'

Discount	0.95	**Learning rate**	0.1
Training episodes	2000	**Training episode length**	200
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1000. Decay constant = 0.9995.		

This agent performs significantly better than the previous one. The mean reward on testing episodes increases from 30.8 to 39.1.

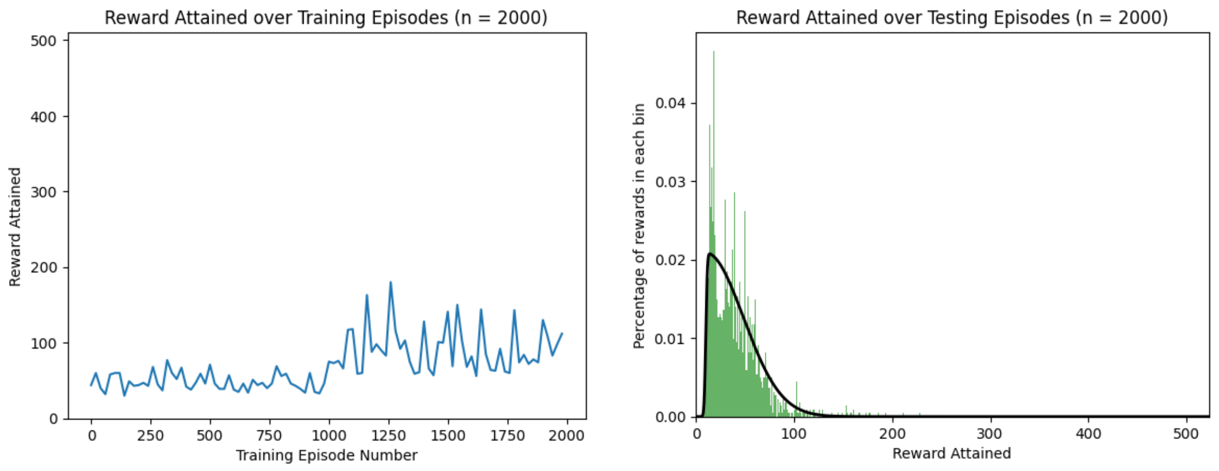


Figure 8. The training reward no longer plateaus at 100, indicating that the agent continues learning during exploitation.

Starting Epsilon Decay Later

The earlier agents improved drastically halfway through training when exploitation starts. But this agent doesn't perform that much better after it starts exploiting. In Figure 9a, there is not a large increase in reward after episode 1000. This suggests that the agent might not be exploring enough of the state space before it starts to exploit. We try delaying epsilon decay until episode 1500.

Table 5. Hyperparameters for Reducing the Number of Training Iterations.'

Discount	0.95	**Learning rate**	0.1
Training episodes	2000	**Training episode length**	200
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1500. Decay constant = 0.9995.		

This agent again performs significantly better than the previous one. The mean reward on testing episodes increases from 39.1 to 53.7.

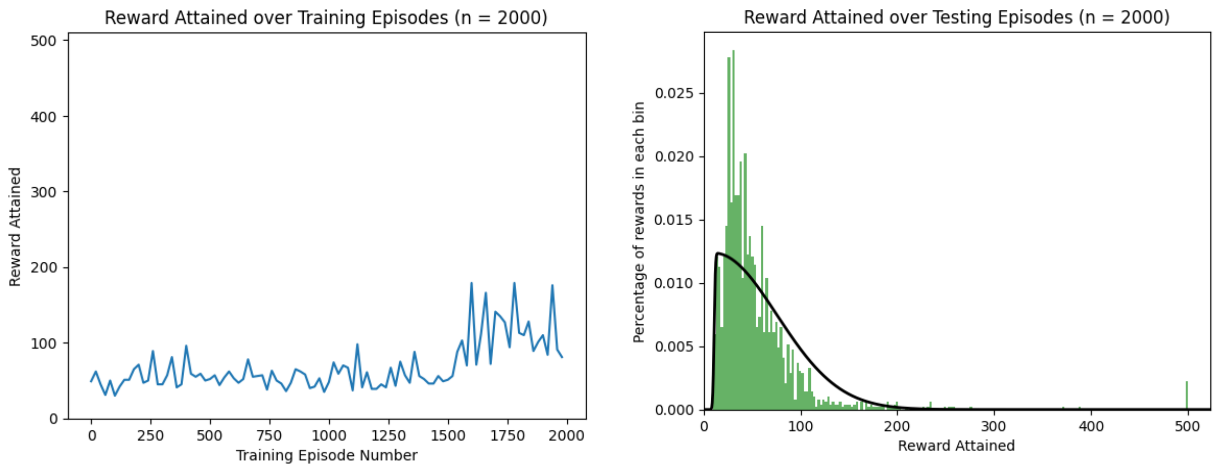


Figure 9. As desired, the training reward improves significantly after the agent begins exploiting.

Annealing the Learning Rate

Next, we attempt to anneal the learning rate so that the agent performs more consistently well at the end of training. We initialize it to 0.3 for the first 500 training episodes, reduce to 0.2 for the next 500, to 0.1 for the next 500 and then to 0.05 for the last 500 episodes.

Table 6. Hyperparameters for Reducing the Number of Training Iterations.'

Discount	0.95	**Learning rate**	Decay from 0.3 to 0.05
Training episodes	2000	**Training episode length**	200
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1500. Decay constant = 0.9995.		

Surprisingly, this agent performs significantly worse. The mean reward on testing episodes reduces to 47.4 from 53.7.

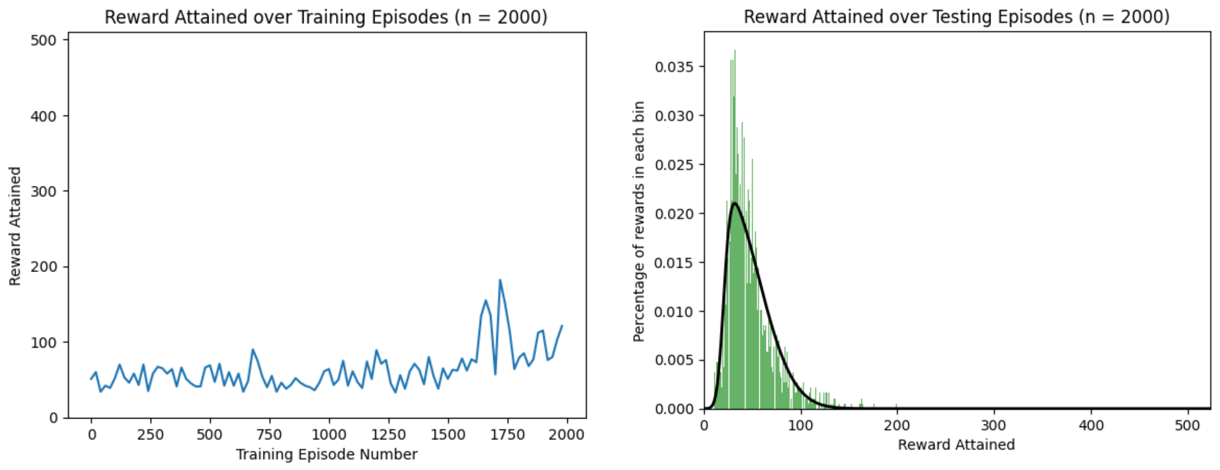


Figure 10. This agent rarely achieves a testing reward over 100.

Increasing the Learning Rate Overall

Instead of giving up on annealing, we notice that the agent might not be learning as much since the learning rate ends so low. Instead of decaying the learning rate from 0.3 to 0.05, we try decaying it from 0.5 to 0.1.

Table 7. Hyperparameters for Reducing the Number of Training Iterations.'

Discount	0.95	**Learning rate**	Decay from 0.5 to 0.1
Training episodes	2000	**Training episode length**	200
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1500. Decay constant = 0.9995.		

This adjustment greatly improves the agent's performance. The mean reward on testing episodes increases to 73.4, making this the best agent so far.

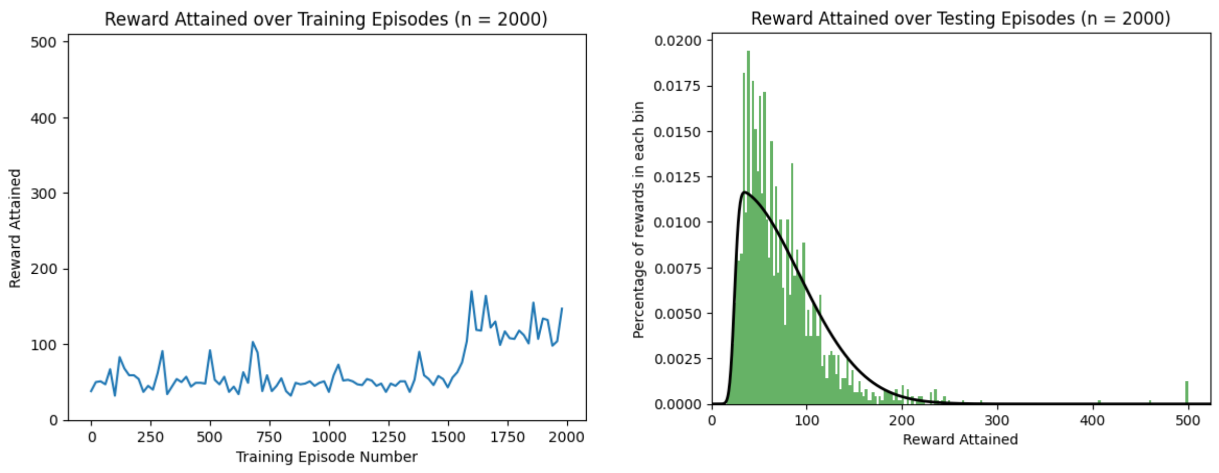


Figure 11. This agent achieves a testing reward over 100 in more than 10% of testing episodes.

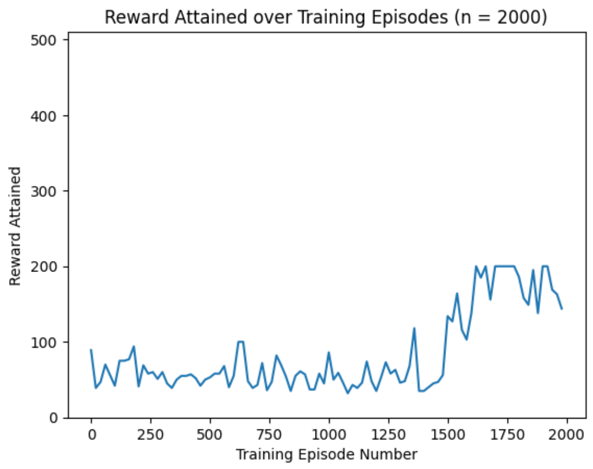
Reincreasing the Learning Rate during Exploitation

The Q-learning agent is now performing well. It is achieving an average reward on testing episodes that is over twice the push-in-direction-of-fall agent. Still, it performs unreliably even after it has started exploiting. This could be because the learning rate is too low during the last 500 episodes, so the agent cannot learn enough during exploitation. We try increasing the learning rate back to 0.5 for the last 500 episodes.

Table 8. Hyperparameters for Reducing the Number of Training Iterations.

Discount	0.95	**Learning rate**	Decay from 0.5 to 0.2, then 0.5 for last 500 episodes.
Training episodes	2000	**Training episode length**	200
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1500. Decay constant = 0.9995.		

This adjustment again greatly improves the agent's performance. The mean reward on testing episodes increases dramatically to 96.7.



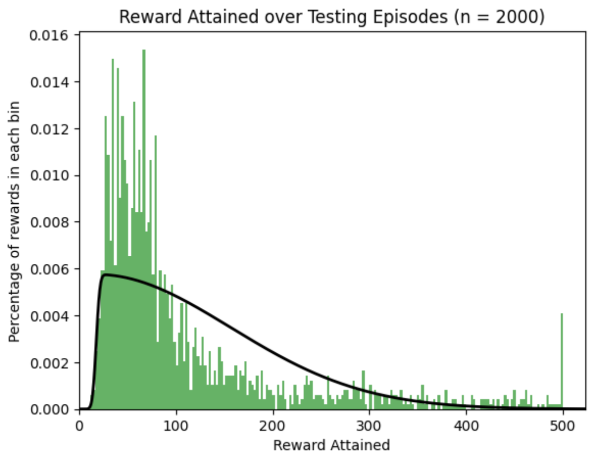


Figure 12. This agent is the first one to hit the maximum training episode length of 200 steps. It would likely perform better if the limit was raised to 300 steps, but that would take longer to run. We look for other improvements instead.

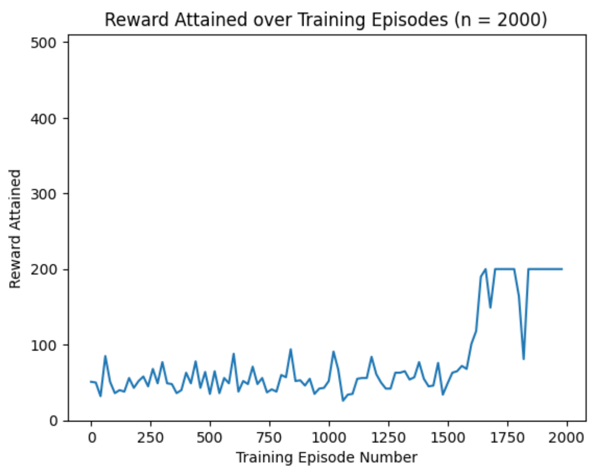
Reducing the Epsilon Decay Constant

Lastly, we realize that the epsilon decay constant is too small since we reduced the number of training episodes. At the old value of 0.9995, at the end of training, epsilon is 0.6. We reduce epsilon to 0.9, so the full epsilon decay from 1 to 0.05 takes place within 500 episodes. Note that epsilon stops decaying once it drops below 0.05.

Table 9. Hyperparameters for Reducing the Number of Training Iterations.'

Discount	0.95	**Learning rate**	Decay from 0.5 to 0.2, then 0.5 for last 500 episodes.
Training episodes	2000	**Training episode length**	200
Epsilon	Exponential decay from 1 to 0.05. Decay starts on episode 1500. Decay constant = 0.9.		

The agent performs better with a smaller epsilon decay constant. The mean reward on testing episodes increases again to 106.7.



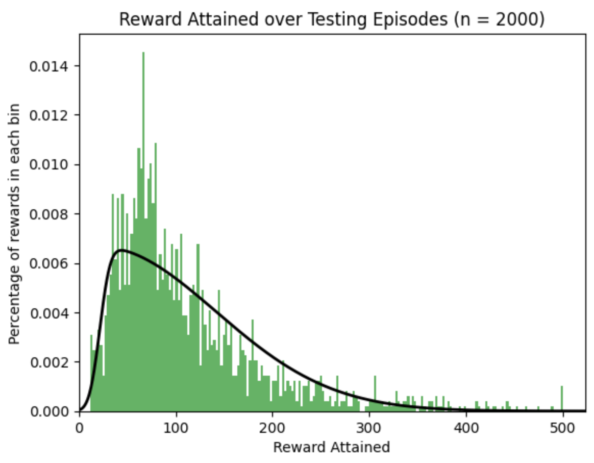


Figure 13. This agent does much less exploration after episode 1500, so the reward improves more drastically than in the previous agent.

Final Remarks

At this point, we move on from the discussion of Q-learning. To be clear, the agent is not fully optimized. We have not even touched the discount hyperparameter yet. However, it performs reasonably well and three times better than the push-in-direction-of-fall agent. We achieved our initial goal.

We learned that in the Cart-Pole simulation, it pays off to spend more time exploring than exploiting. Our team is curious whether this extends to other physical systems, especially since teams that tackled other types of RL problems like playing Atari Breakout found that agents perform best with a fixed epsilon-greedy approach, exploiting as much as 80% of the time.

We also learned that backtracking on changes is OK. For example, we first reduced the maximum training episode length from 500 to 100, then increased it back to 200 as we finetuned the model. This sort of back-and-forth is natural and expected.

Deep Q-Networks

About DQNs

Deep Q-Networks (DQNs) are deep neural networks trained to output the best action from an agent's state. The algorithm was developed by DeepMind in 2015 and has proven useful for high-dimensional spaces such as Atari games.[2] We created a DQN agent for the CartPole simulation to try and outperform traditional Q Learning.

The hyperparameters for DQNs include all the standard Q-learning hyperparameters of learning rate, discount factor, epsilon, number of training episodes, and maximum training

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

episode length. There are also hyperparameters associated with the deep neural network, such as the number of hidden layers, hidden layer size, and replay buffer size.

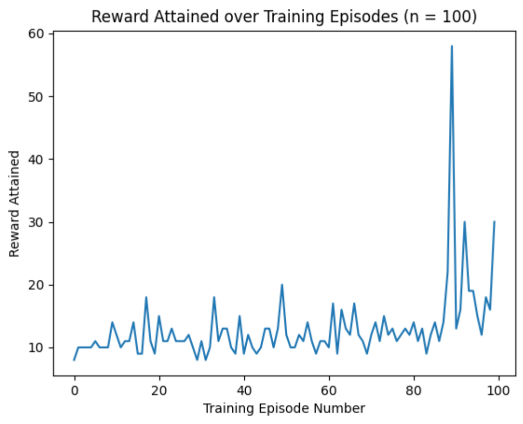
Base Case

Since there are so many hyperparameters, and DQNs take relatively long to train, we couldn't systematically search through every possible hyperparameter setting. Instead, we start from a base case and make informed changes to the hyperparameters. Our base case DQN agent has the following agent hyperparameters. A detailed discussion of the network structure is in Appendix B, but in short, the base case uses a 3-layer neural network with ReLU activations.

Table 10. Hyperparameters for the base case.

Discount	0.95	**Learning Rate**	0.01
Training Episodes	100	**Training Episode Length**	100
Epsilon	0.3	**Replay Buffer Size**	10^6

The agent likely needs more training time. We achieved a mean training reward of 13.0 ± 5.9. The mean testing reward is 15.0 ± 2.4. These results are poor but are understandable.



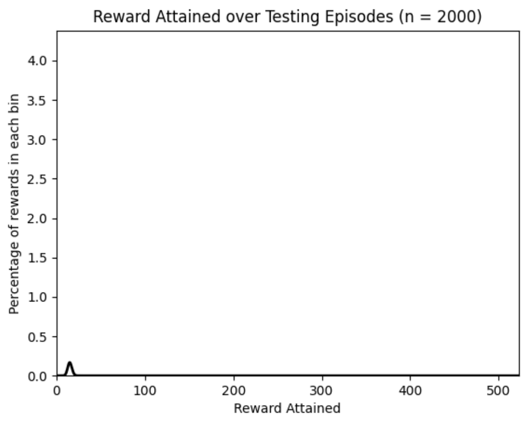


Figure 14. We can see from the training rewards curve that the reward rises after about 80 episodes, but the mean testing reward is poor.

Optimizing the Agent Hyperparameters

Next, we optimize the agent hyperparameters. Since there are so many, we fixed the discount factor to 0.95, epsilon at 0.3, and the replay buffer capacity at 10^6. We chose these values since they were used in an existing implementation of DQN.[3] We focused on optimizing number of episodes, timesteps per episode, and learning rate. In the next section, we discuss the network parameters in more detail.

[3] https://github.com/pfnet/pfrl/blob/master/examples/quickstart/quickstart.ipynb

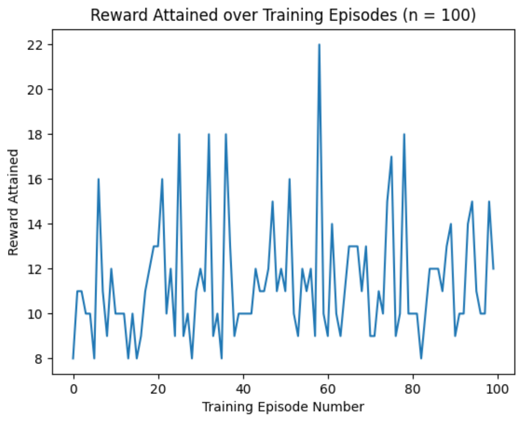
Optimizing the Learning Rate

Our base case used a learning rate of 0.01. We tried increasing it to 0.1.

Table 11. Hyperparameters for the learning rate of 0.1.

Discount	0.95	**Learning Rate**	0.1
Training Episodes	100	**Training Episode Length**	100
Epsilon	Constant Eps. Greedy (0.3)	**Replay Buffer Size**	10^6

The agent performs similarly poorly. The mean testing reward is similar at 9.4 ± 0.8.



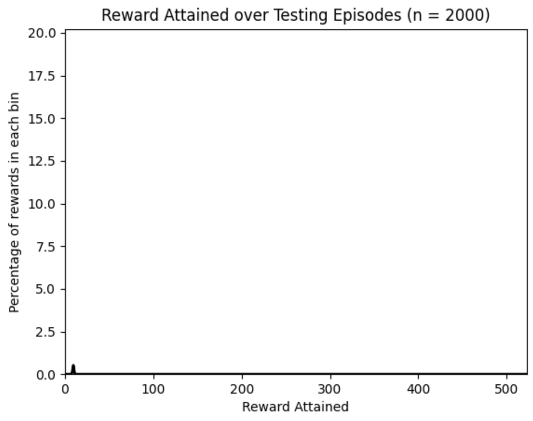


Figure 15. The left graph shows the reward during training. The reward during testing never improves past 9-10, so the histogram appears empty.

With a higher learning rate, the model should learn faster, but this might be hard to see since the model simply can't learn with so few training episodes.[4]

Optimizing the Number of Training Episodes

We theorize that the agent just needs more time to train. So first we try upping to 300 training episodes.

Table 12. Hyperparameters for 300 episodes case.

Discount	0.95	**Learning Rate**	0.01
Training Episodes	300	**Training Episode Length**	100
Epsilon	Constant Eps. Greedy (0.3)	**Replay Buffer Size**	10^6

The agent performs astonishingly better. The mean testing reward is 139.1 ± 39.5.

[4] https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

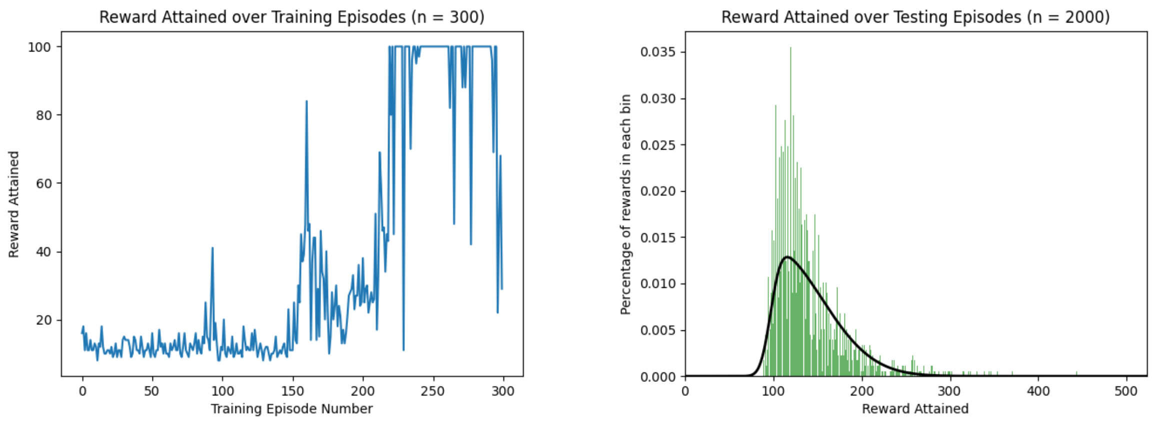


Figure 16. The testing reward is rarely below 100.

Next, we try upping to 500 training episodes.

Table 13. Hyperparameters for the 500 episodes case.

Discount	0.95	**Learning Rate**	0.01
Training Episodes	500	**Training Episode Length**	100
Epsilon	Constant Eps. Greedy (0.3)	**Replay Buffer Size**	10^6

The mean testing reward jumps further to 162.3 ± 21.0.

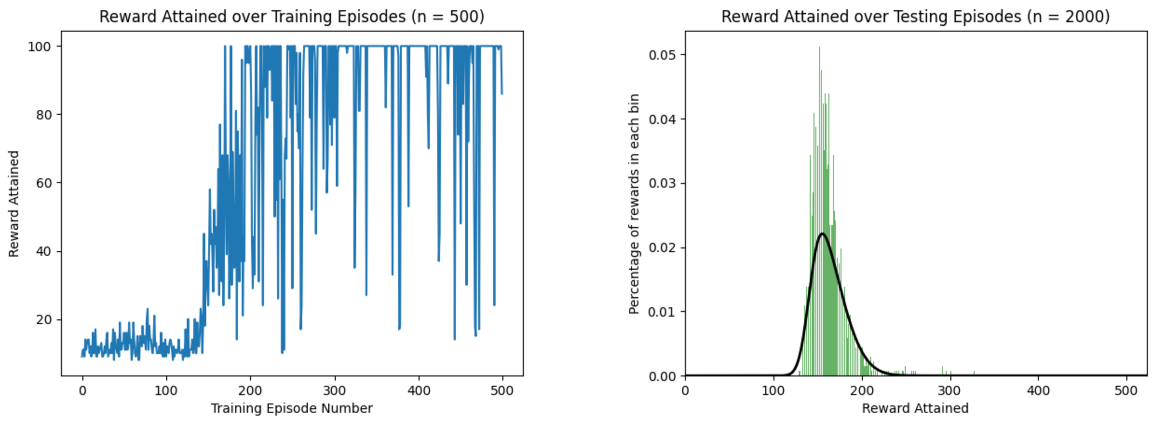


Figure 17. The left graph shows the reward during training. The reward during testing averages between 100 and 200.

As expected, increasing the number of training episodes improves performance. The detailed results are presented in the appendix.

Optimizing the Timesteps per Episode

Next, we experiment with increasing the maximum training episode length. First, we up it to 300 timesteps.

Table 12. Hyperparameters for 300 timesteps.

Discount	0.95	**Learning Rate**	0.01
Training Episodes	500	**Training Episode Length**	300
Epsilon	Constant Eps. Greedy (0.3)	**Replay Buffer Size**	10^6

The agent's mean testing reward reduces from 162.3 to 116.8 ± 5.2. However, the agent is more consistent.

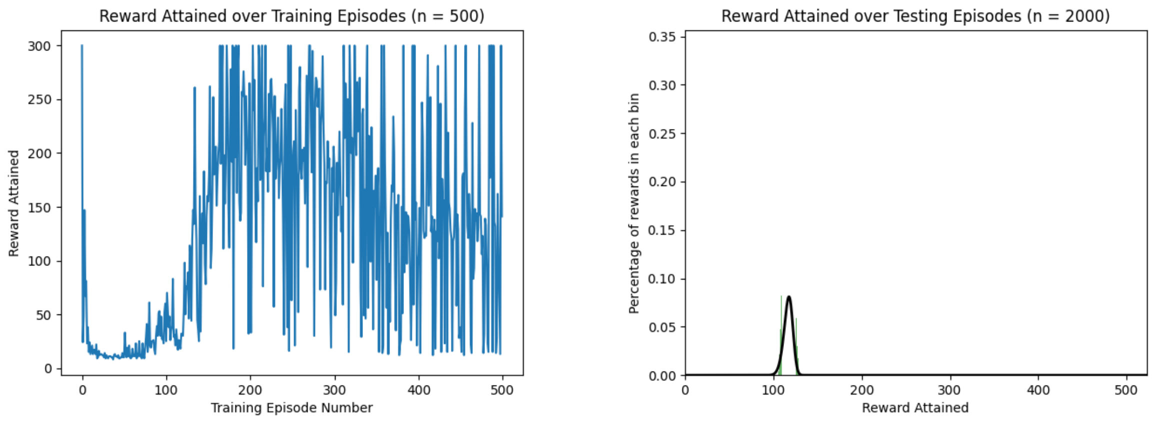


Figure 18. This agent consistently achieves rewards from 100 to 140.

The consistency is hard to explain. But the agent may perform worse since with a higher maximum episode length, it spends less time on new random starts. Just to be sure, we try increasing to 500 timesteps.

Table 13. Hyperparameters for 500 timesteps.

Discount	0.95	**Learning Rate**	0.01
Training Episodes	500	**Training Episode Length**	500
Epsilon	Constant Eps. Greedy (0.3)	**Replay Buffer Size**	10^6

The agent performance deteriorates further to a mean testing reward of 102.1 ± 3.8.

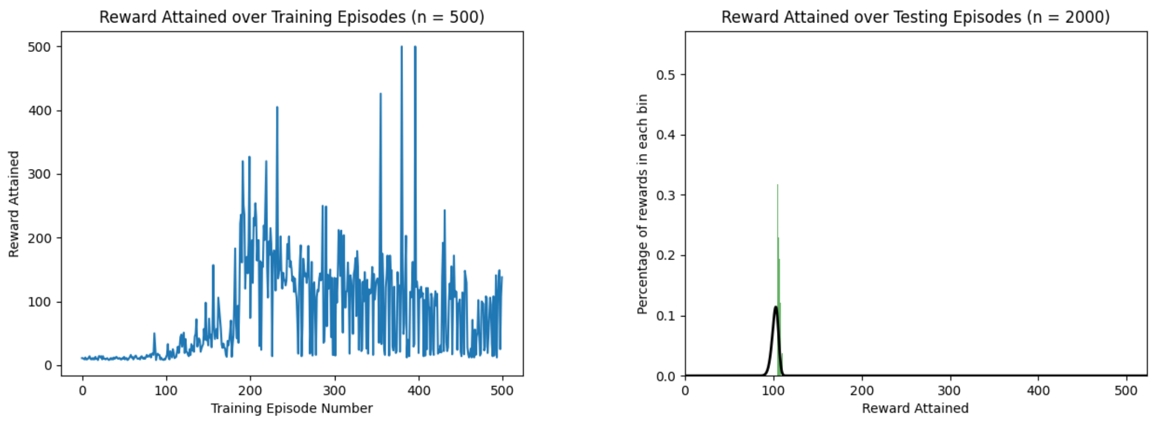


Figure 19. The reward during testing is again consistent, but worse.

This further corroborates that increasing episode length will not help.

Optimizing the Q-Network Parameters

Next, we optimize the network parameters. We fix 300 training episodes, a max episode length of 100, and a learning rate of 0.01. With the original 3-layer network with no dropout, the mean testing reward was 139.1 ± 39.5.

Two Hidden Layers

In class, we learned that deeper networks can learn more complex features, so we explore whether adding more hidden layers better approximates the Q-values of the state-action pairs. First, we add one more hidden layer (the hidden layers have a fixed size of 50).

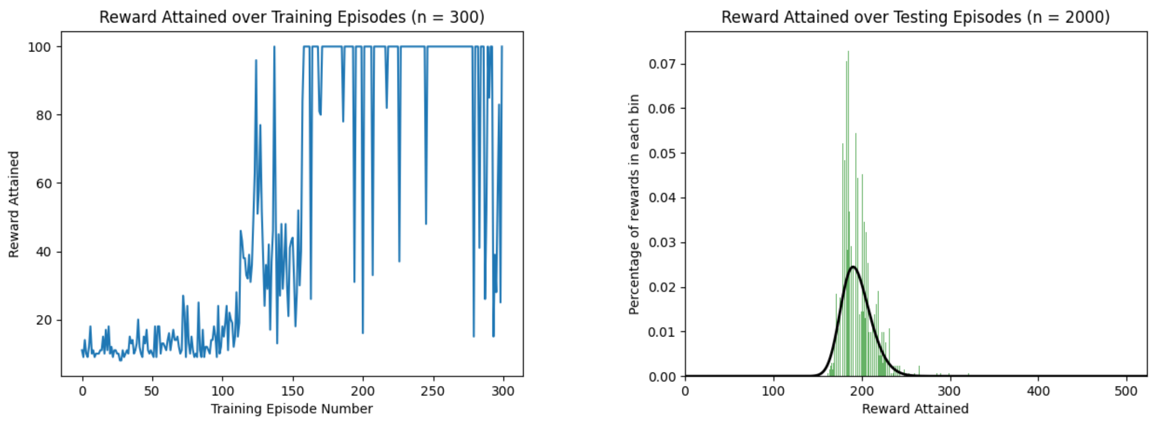


Figure 20. The reward during testing appears normally distributed around 200.

With two hidden layers, the mean testing reward is 194.1 ± 17.4. This is a significant improvement.

Four Hidden Layers

Since adding one hidden layer helped, we try adding two more. But then the testing reward drops drastically to 84.7 ± 29.1.

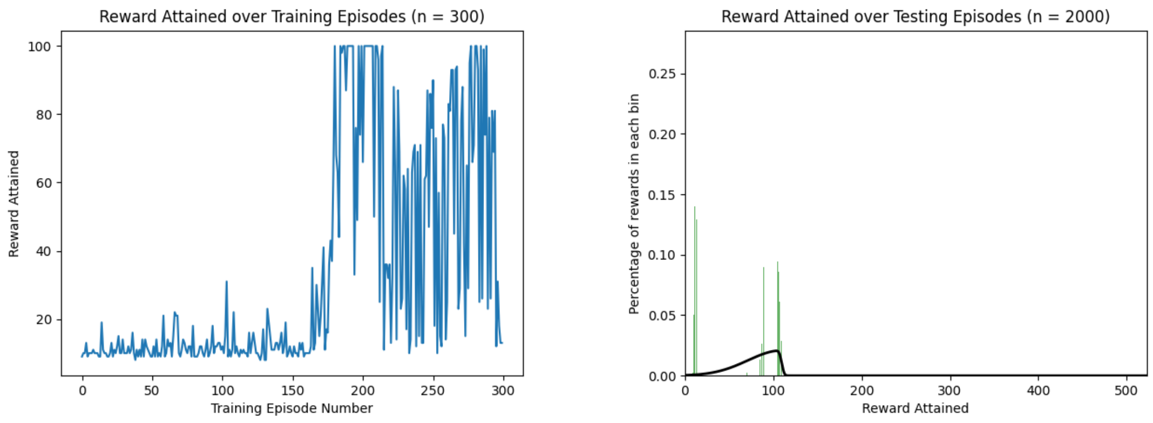


Figure 21. The reward during testing is highly inconsistent, either over 100 or near zero.

With this many layers, the agent might be overfitting to situations it encounters during training.

Adding Dropout

Our last optimization is to try adding a dropout layer (p = 0.2) to the single hidden layer agent, in case it is also overfitting. The mean testing reward skyrockets to 264.1 ± 89.9.

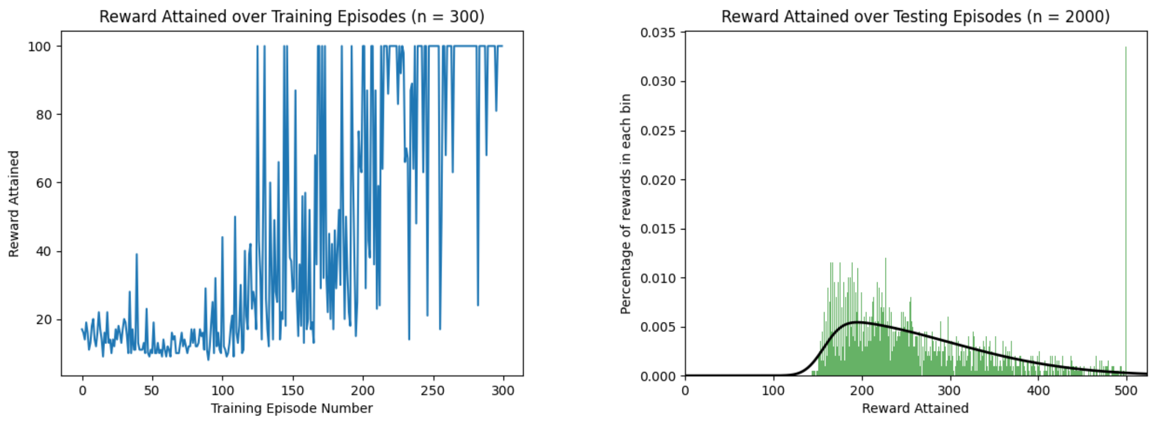
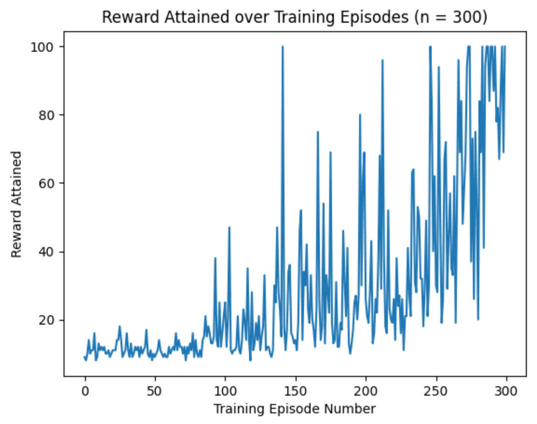


Figure 22. The agent achieves a perfect reward about 3.5% of the time.

After adding another dropout layer, the testing reward jumps further to 277.3 ± 109.7.



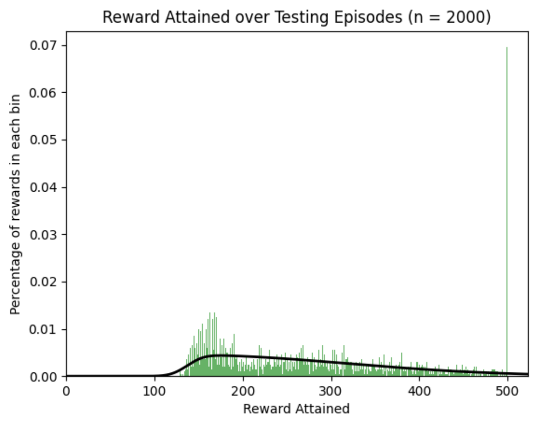


Figure 23. This agent achieves a perfect reward over 7% of the time.

The agent with dropout layers takes longer to run, but it performs much better. This suggests that since our problem is not as complicated as, say, Atari Breakout or Tetris, DQN is prone to overfitting on the Cart-Pole.

Comparison to Q-Learning

We end our discussion of deep Q-networks by comparing their performance to Q-learning. So that the comparison is fair, we fix the number of training episodes and the maximum episode length. All the other hyperparameters are chosen as the ones that resulted in the highest average reward on testing episodes in our earlier explorations.

Since the Q-Learning agent was optimized for 2000 episodes and a maximum episode length of 200, we first compare the two models in that setting.

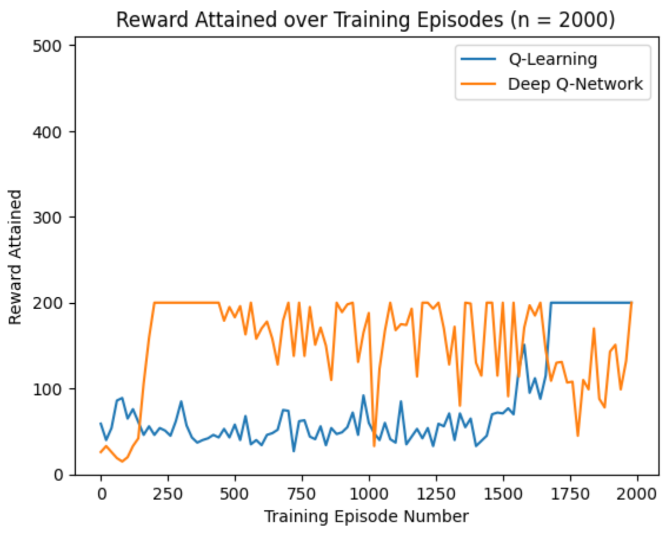


Figure 24. The DQN rapidly outperforms traditional Q-learning and begins hitting the maximum training episode length after just 250 episodes.

The DQN appears to train a lot faster than Q-learning. It also outperforms Q-learning on testing episodes. Over 2000 testing episodes, DQN achieves an average reward of 244.3 while Q-learning achieves an average reward of 151.2.

Next, we try comparing their performance after just 500 training episodes and a maximum episode length of 500. DQN was optimized for this setting. The only hyperparameter that changes is that for Q-learning, epsilon decay starts after 250 iterations instead of 1000.

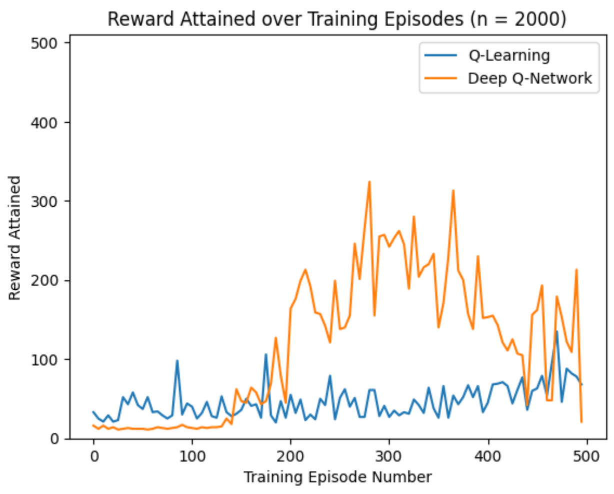


Figure 25. Once again DQN appears to outperform traditional Q-learning which barely improves in so few training episodes.

After 500 training episodes, DQN achieves a mean testing reward of 86.7. Q-learning achieves a mean testing reward of only 43.4. While both still perform better than the baseline agents, DQN clearly outperforms again.

Recall that after 20,000 training iterations, Q-learning is also nearly perfect at keeping the Cart-Pole vertical. So both agents have the capacity to learn the simulation, but DQN appears to learn faster.

Conclusions and Future Extensions

Through this project, we achieved our goals of learning more about RL for physical systems, using OpenAI Gym, and deep Q-networks. Although the Cart-Pole simulation is fairly simple, it provided a rich playground to explore how hyperparameters affect Q-Learning and DQNs.

Our analysis is not 100% complete. For example, we need to go back and take a look at how epsilon affects DQNs. However, we created RL agents that perform much better than the baseline agents. And we discovered that DQNs provide a significant improvement over normal Q-learning. We also would like to work on changing the experience replay buffer in the future.

If we had more time, we would explore improvements on DQNS like Double DQNs. It would also be fun to add user interaction, i.e. create a game where users use left/right keys to keep the stick up longer than our AI (with easy/medium/hard levels based on which agent you play against).

Appendix

Appendix A: Discretizing the State Space for Q-Learning

The pole angle and pole velocity are more important for keeping the pole upright. So we split the cart position and velocity into 30 bins each, and the pole angle and velocity into 50 bins each. Since the cart position is bounded in [-4.8,4.8] and pole angle is bounded in [-22.5 degrees, 22.5 degrees], we bin them by splitting the range into 30 and 50 equal-sized bins. The cart velocity and pole velocity are unbounded, but over 10000 timesteps, the cart velocity was almost always within [-2,2] and the pole velocity within [-3,3]. So we bin them by splitting the range into 28 and 48 equal-sized bins, leaving two bins for high and low outliers.

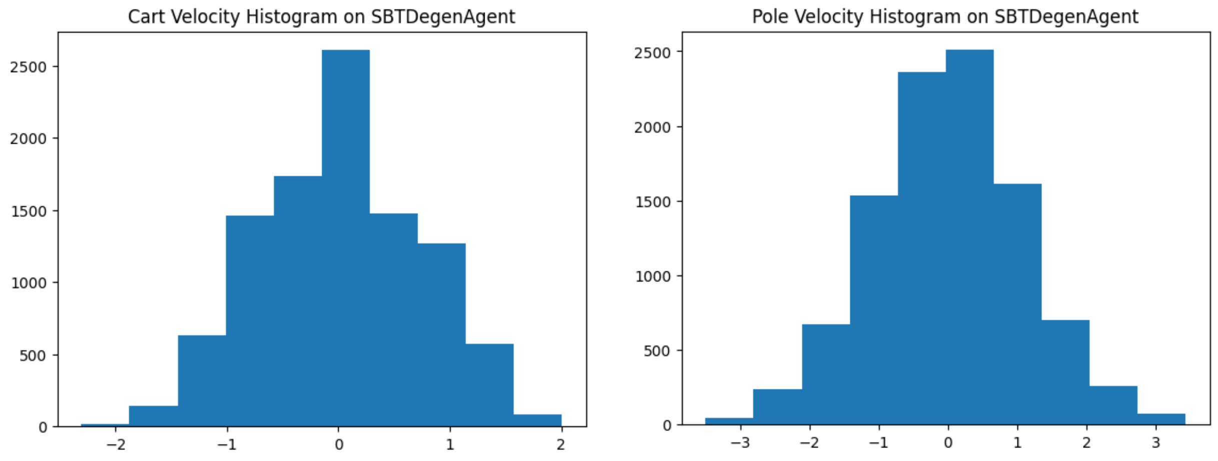


Figure 26. The histograms above depict the distribution of cart velocities and pole velocities over 10,000 timesteps. Although the observations are theoretically unbounded, they don't become very large in practice.

Appendix B: Network Description

Our Q-function initially used a network with three layers and ReLU activations to add nonlinearity. It was based on an existing implementation of DoubleDQN.[5] The Adam optimizer was used to update network weights.

[5] https://github.com/pfnet/pfrl/blob/master/examples/quickstart/quickstart.ipynb

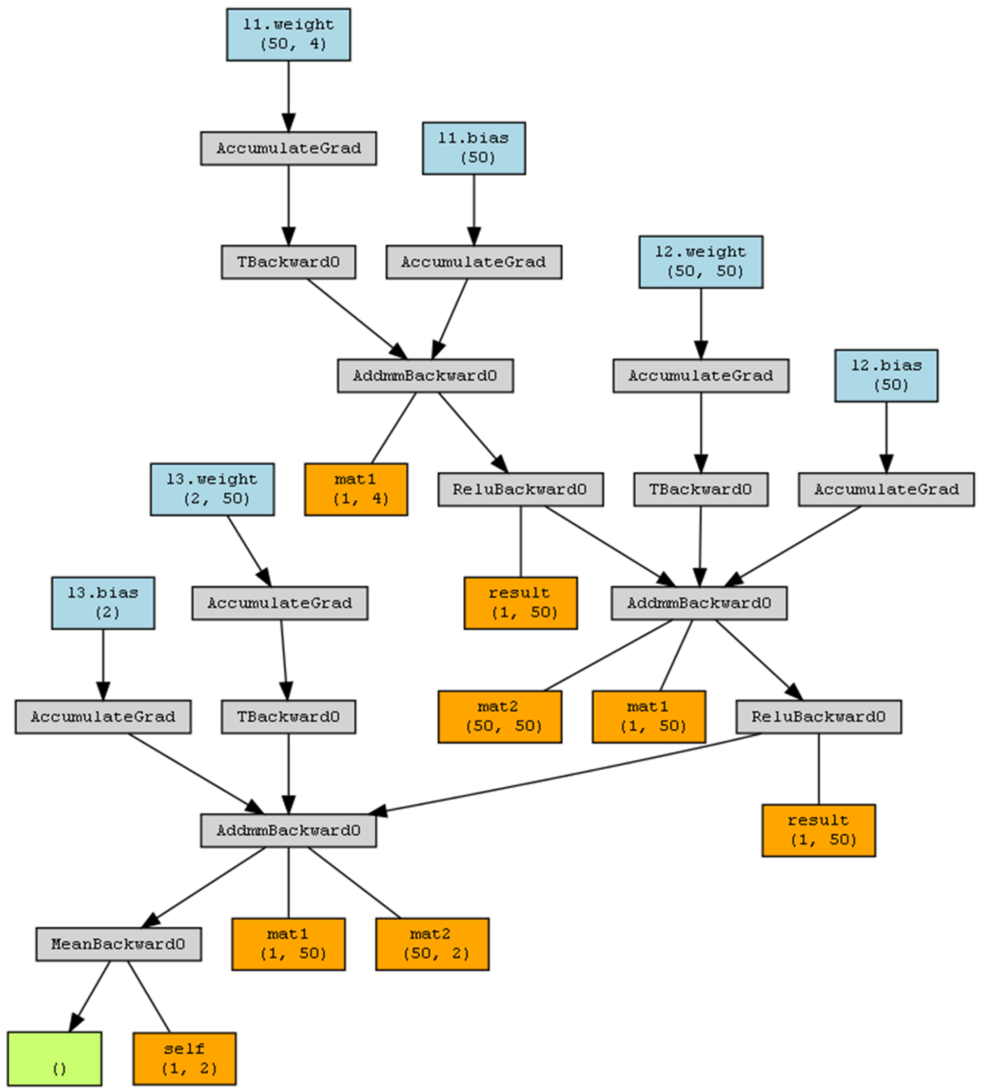


Figure 27. This is a visualization of the 3-layer DQN network, created in the PyTorchViz library.[6]

Appendix C: Hyperparameter Optimization for DQN

The mean and standard deviation of rewards were observed while changing the learning rate, number of episodes, and timesteps in an episode. For learning rate, the values of 0.1 and 0.01 were tested. The number of episodes was varied between 100, 200, 300, and 500, while the timesteps in an episode were changed to 100, 200, 300, and 500 as well. These variations allowed for a comprehensive analysis of the effect of each parameter on the rewards received by the agent. The results of this analysis can be used to optimize the training process and improve the performance of the agent to see what combination of hyperparameters provide what performance. This analysis was done by keeping the maximum timesteps of 200 for evaluation.

[6] https://github.com/szagoruyko/pytorchviz

Table 14. Hyperparameters Results for evaluation rewards

Evaluation Rewards									
Mean									
Learning Rate: 0.1					Learning Rate: 0.01				
Length↓ Episodes →	100	200	300	500	Length↓ Episodes →	100	200	300	500
100	41.6	9.3	66.1	9.3	100	9.4	159.0	198.6	94.9
200	9.3	9.4	9.4	21.9	200	9.4	73.4	198.0	119.2
300	9.3	9.5	62.4	70.6	300	9.4	62.5	195.1	114.1
500	9.5	9.3	54.1	191.5	500	16.1	195.6	179.3	170.5
Standard Deviation									
Learning Rate: 0.1					Learning Rate: 0.01				
Length↓ Episodes →	100	200	300	500	Length↓ Episodes →	100	200	300	500
100	18.4	0.7	29.0	0.8	100	0.8	13.8	5.1	3.7
200	0.7	0.7	0.7	3.4	200	0.7	7.3	4.1	5.6
300	0.7	0.9	21.3	5.4	300	0.7	19.4	8.8	6.9
500	0.9	0.7	16.0	16.9	500	2.8	8.4	15.5	10.7

Table 15. Hyperparameters Results for training rewards

Training Rewards									
Mean									
Learning Rate: 0.1					Learning Rate: 0.01				
Length↓ Episodes →	100	200	300	500	Length↓ Episodes →	100	200	300	500
100	17.5	12.5	21.5	18.3	100	12.0	36.9	50.4	59.2
200	11.8	13.4	22.4	20.7	200	12.2	31.8	96.2	78.8
300	10.8	12.7	12.9	30.6	300	11.6	17.8	101.1	72.2
500	11.6	12.5	25.7	34.0	500	13.9	63.8	81.2	91.7
Standard Deviation									
Learning Rate: 0.1					Learning Rate: 0.01				

Length↓ Episodes →	100	200	300	500	Length↓ Episodes →	100	200	300	500
100	13.0	5.1	17.0	13.6	100	4.7	38.2	38.6	38.0
200	2.7	5.9	19.7	15.9	200	4.2	38.0	79.3	68.2
300	2.0	7.0	6.8	26.6	300	3.2	14.2	88.2	67.6
500	2.9	8.1	19.9	43.0	500	9.2	84.0	78.6	79.4

Observations -

- Based on the learning rate
    - For a learning rate of 0.1: Generally, it can be seen that as both the length per episode and the number of episodes increase, so does the reward for a maximum reward of 200. However, increasing the number of episodes for a low length per episode, and increasing the length of episodes for a smaller number of episodes leads to less reward. Best training and evaluation mean reward is obtained for 500 episodes with a length of 500 each episode.\
    - For a learning rate of 0.01: Generally speaking the performance is better for a learning rate of 0.01 than it is for a learning rate of 0.1. This was observed for pretty much most cases except for a high number of episodes (500). However, no specific trend was observed for a learning rate of 0.01, as was observed for a learning rate of 0.1. The best training and evaluation mean reward is obtained for 300 episodes with a length of 100 per episode. This result is very interesting, and it differs from the highest number of episodes and length yields the highest mean reward that we observed for a learning rate of 0.1.
    - For a lower learning rate (0.01), the training took more time than for a higher learning rate (0.1).
- There was no clear trend observed between the training and testing mean and standard deviation. For some of the values, the training performance was better, while for others, the evaluation performance was better.
- It was interesting to see the standard deviation values exceeding the mean values for the averaged rewards for some specific cases. This can be attributed to the sharp rise and decline of reward values as the episodes progressed.
- It was clear that the standard deviation during training is overall much higher than it is during evaluation.