Landing on the Moon with Reinforcement Learning

Deep Q-Learning on OpenAI LunarLander Game

Cailyn Craven
cailyn.craven@colorado.edu
Github: https://github.com/cailyn-craven/OpenAIFinal

1. Overview of Project:

Reinforcement Learning (RL) is a type of artificial intelligence where agents take actions in an environment to maximize cumulative rewards. RL overlaps with many disciplines including psychology, neuroscience, game theory, and control theory.[1] Many classic RL algorithms like ***Temporal Difference Learning*** date back to the 1980s or earlier.[2]

In the past decade, RL research has soared given the popularity of deep learning techniques and the ability to pair the two. Using non-linear neural networks as function approximators for Deep Reinforcement Learning required some research breakthroughs since initial Naive Deep Q Learning attempts struggled with the ***deadly triad.***

The ***deadly triad*** includes the following problems: *Convergence is not guaranteed when using a function approximator with a non-linear model. * If a single network is simultaneously used to evaluate the maximum action and choose the maximum action while being updated at every step, it has the effect of chasing a moving target. The model being trained doesn't actually have a gradient. *Highly correlated samples. [3]

In the 2015 *Nature* paper "Human-level control through deep reinforcement learning" Mnih et al. described how they utilized a deep-Q-network agent (DQN) that reached human level performance on Atari games. To overcome the deadly triad, the researchers utilized a ***replay buffer*** to give the agent a memory. States are randomly sampled in a different order before using gradient descent so samples are no longer correlated. To solve the problem of a moving target Q-value making the regression target unstable, the implementation uses a separate target network and prediction network.[4] From Atari games, DeepMind continued to take on harder challenges. Reinforcement Learning featured prominently in the AlphaGo program that defeated Go champion Lee Sedol in 2016.[5]

One of the primary purposes of this project was to use ***OpenAI Gym,*** a toolkit for developing and comparing reinforcement learning algorithms.[6] To understand Reinforcement Learning better, I started with a simple implementation of the classic ***Temporal Difference Algorithm*** to solve the FrozenLake-v0 environment.

Realizing the limitations of a tabular method, I turned my attention to Deep Reinforcement

[1] "Reinforcement Learning," Wikipedia, https://en.wikipedia.org/wiki/Reinforcement_learning
[2] "Temporal Difference Learning," Wikipedia, https://en.wikipedia.org/wiki/Temporal_difference_learning.
[3] Kim, Geena. "Deep Reinforcement Learning Intro," Lecture at University of Colorado Boulder, Fall 2020.
[4] Mnih et al. "Human-level control through Deep Reinforcement Learning," *Nature,* v. 518, 2015, https://applied.cs.colorado.edu/pluginfile.php/15569/mod_resource/content/1/MnihEtAlHassibis15NatureControlDeepRL.pdf
[5]**.** Silver et al. "Mastering the game of Go without human knowledge,**"** *Nature* Vol 550, October 2017.

[6] "OpenAI Gym Documentation, https://gym.openai.com/

Learning. Phil Tabor provides an interesting walkthrough of a ***Naive DQN*** that illustrates the problems that arise from the deadly triad in naive implementations of a Deep Q-Learning algorithm for the Cartpole-v0 environment.[7] After seeing firsthand some of the challenges of utilizing Reinforcement Learning and Deep Learning together, I turned to a ***Deep Q-Learning*** implementation capable of helping an agent learn on how to land on the moon in the LunarLander-v2 environment.

Approach:

2. Environments and Game Rules

Over the course of the project, I touched on three different environments:

- For the basic ***Temporal Difference Algorithm,*** I utilized the ***FrozenLake-v0*** environment. FrozenLake is a simple text-based ***gridworld*** environment where the agent starts off in the top left corner and needs to make it down to the bottom right corner to retrieve a frisbee. The agent is on a frozen lake so sometimes it will stochastically slide in a direction it didn't intend to go. The agent receives a reward of 1 for making it to the bottom right corner. Other steps have a reward of 0. The game terminates if the agent falls into a hole on the ice. The 4x4 grid means the ***state space*** is 16, and the agent has four possible ***actions:*** left, right, up or down. [8]
- To better understand the deadly triad, I utilized ***Cartpole-v0.*** This was the toy problem utilized in the notebook introducing this project so it was important to base the main project on another environment. Therefore, I will just touch on this one briefly. In this ***classic control*** problem, a pole is attached by a join to a cart moving along a track. The pole starts upright, and the agent's goal is to prevent it from falling over. To control the system, apply a force of +1 or -1 to the cart. The agent receives a reward of +1 for every timestep the pole remains upright. The game ends when the pole reaches more than 15 degrees from vertical or the cart has traveled 2.4 units away from the center.[9]
- The primary environment for this project was ***LunarLander-v2,*** a continuous control task in the Box2D simulator. The simulation is based on a 1979 Atari Game. In this simulation, the goal is to land a rocket on the moon. We would expect that landing a real ship on the moon would require knowing lots of physics and control theory. This project utilizes RL to see if the agent can learn how to land on the moon without knowing anything about dynamics. The agent trains to learn how to land efficiently (and most importantly avoid crashing!)

  The problem of landing can be formalized as an ***MDP.*** The environment has a landing pad which is always at coordinates (0,0). The ***state*** is composed of ***8 variables***: position x, position y, velocity x, velocity y, angle, angular velocity, and sensors for whether each of the two legs are touching the ground (the sensors have value 0.0 for not touching, or 1.0 for touching). Six of the eight state variables are ***continuous*** with values ranging between negative infinity and infinity. The values are float32.

[7] Tabor, Phil. "Modern Reinforcement Learning: Deep Q Learning in PyTorch," https://www.udemy.com/course/deep-q-learning-from-paper-to-code/.

[8] FrozenLake-v0, OpenAI Gym Documentation, https://gym.openai.com/envs/FrozenLake-v0/

[9] Cartpole-v0, OpenAI Gym Documentation, https://gym.openai.com/envs/CartPole-v0/

The ***4 discrete actions*** are: do nothing, fire left orientation engine, fire main engine, or fire the right orientation engine.

Reward: The reward is an important part of an MDP. For this game, the agent receives a reward between 100 and 140 points for moving from the top of the screen to the landing pad at zero speed. The agent loses the reward if it subsequently moves away from the landing pad. The episode finishes with a large reward of +100 for landing successfully or -100 for crashing. A leg contacting the ground receives a reward of +10. Firing the main engine loses 0.3 points per frame. In this version, the ship has infinite fuel. The game is considered solved if an agent averages 200 points over the past 100 games.[10]

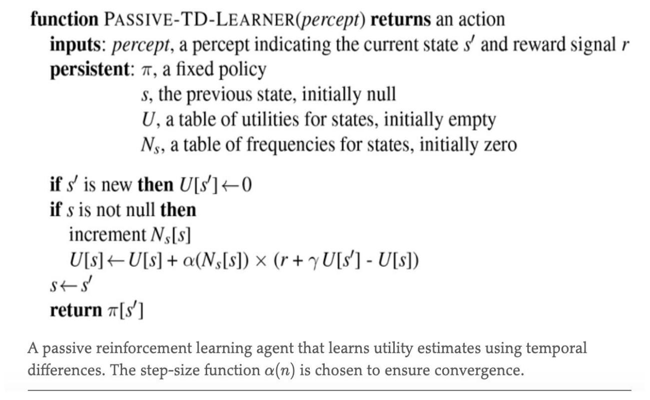
3. Model Choices

Problem Solving:

I structured the project as a progression of looking at Temporal Difference, a classic reinforcement learning algorithm, then trying out a Naive Implementation of Deep Q Learning, and then an implementation of Deep Q Learning so that I could really understand how algorithms could address the limitations of other algorithms to make them suited to solving problems in certain environments.

For the FrozenLake environment, I utilized **Temporal Difference** which keeps a running moving average, bootstrapping an estimate of the value function from the current estimate of the value function.[11]

Artificial Intelligence: A Modern Approach has the following pseudocode for the TD Learning Algorithm:



```
function PASSIVE-TD-LEARNER(percept) returns an action
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r$
  persistent: $\pi$, a fixed policy
              $s$, the previous state, initially null
              $U$, a table of utilities for states, initially empty
              $N_s$, a table of frequencies for states, initially zero

  if $s'$ is new then $U[s'] \leftarrow 0$
  if $s$ is not null then
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s]) \times (r + \gamma U[s'] - U[s])$
  $s \leftarrow s'$
  return $\pi[s']$
```

A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence.

[12]

A **tabular learning method** uses a table to keep track of the states and actions an agent has visited. Such an approach was well suited to the FrozenLake environment because it has a small number of states and actions so it is possible to represent the state action paris as a table with the columns as the actions and the states as the rows.

I utilized an ***Epsilon Greedy action selection*** strategy which balances ***exploration-exploitation*** by looking up the Q-value for each action and most of the time taking the ***greedy max action*** but sometimes taking a random action with epsilon probability.

This classic algorithm was easy to implement in OpenAI Gym, but when I moved on to the LunarLander environment, the classic Temporal Difference algorithm wasn't enough for the problem. Unfortunately, the value function for the LunarLander environment can't be represented with a table because of the continuous state space variables.

Given the limitations of tabular methods, I utilized the following process to select an algorithm

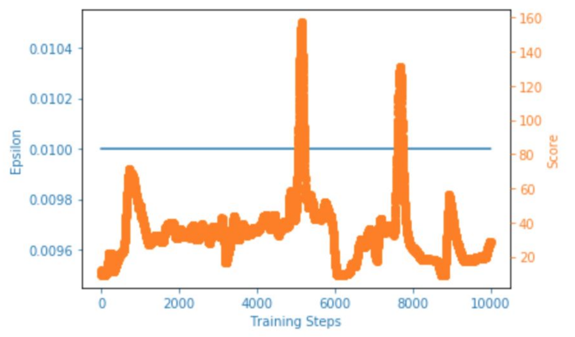
[10] LunarLander-v2, OpenAI Gym Documentation, https://gym.openai.com/envs/LunarLander-v2/

[11] "Temporal Difference Learning," Wikipedia, https://en.wikipedia.org/wiki/Temporal_difference_learning

[12] Russell and Norvig, *Artificial Intelligence: A Modern Approach,* 4th edition, 2020.

better suited to the LunarLander environment. The ship starts in a low orbit and descends until it lands or crashes. Then each landing attempt repeats independently of the previous one. This means that the algorithm should be well suited to ***episodic tasks.*** Q-Learning, SARSA, or Expected SARSA with a ***neural network function approximator*** are good choices for a **control problem** like this one. For this project, I explored the Q-Learning algorithm.

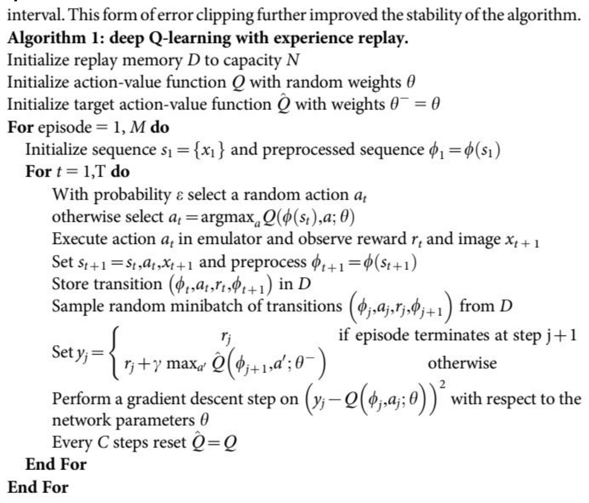
For researchers, it wasn't immediately clear how to pair Reinforcement Learning and Deep Learning. The deadly triad was challenging to overcome, and I wanted to make sure that I understood the challenges. Phil Tabor gives a very interesting tutorial where he does a Naive Implementation of Q-Learning for the Cartpole environment to show how it fails.[13] This isn't the primary part of my project and I coded along with Tabor to complete it, but I included a copy of the notebook on GitHub in the NaiveDeepQLearning file because the results were interesting.



As epsilon decreases the agent's performance improves until it hits a peak as would be expected. Then after reaching a peak, the rewards come crashing back down. This occurs several times.

Reasons for the Naive Deep Q algorithm failing include the following: The agent discards its past experiences without learning from them. As epsilon decreases, the agent is more likely to get stuck in a local minimum. The same network is being used to evaluate the max action and choose the max action so the model is chasing a moving target.

Perhaps if we ran this long enough, the agent would actually learn, but Cartpole only has a state space of size 4. Most Atari games have much higher input dimensions, and even for the 8 input dimensions for the LunarLander simulation, we would not expect this Naive Q-Learning approach to work. Therefore, to implement the lessons from the Mnih et al. 2015 paper and avoid the ***deadly triad*** for a DQN, it is important to include a ***replay buffer*** and ***two networks:*** a target network and a prediction network. This is the Deep Q-Learning Algorithm from the Mnih et al. paper that the project implements.



interval. This form of error clipping further improved the stability of the algorithm.

Algorithm 1: deep Q-learning with experience replay.

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
     With probability $\varepsilon$ select a random action $a_t$
     otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
     Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
     Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
     Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
     Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
     Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
     Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
     network parameters $\theta$
     Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

14

[13] Tabor, Phil. "Modern Reinforcement Learning: Deep Q Learning in PyTorch," https://www.udemy.com/course/deep-q-learning-from-paper-to-code/.

[14] Tabor, Phil. "Modern Reinforcement Learning: Deep Q Learning in PyTorch," https://www.udemy.com/course/deep-q-learning-from-paper-to-code/.

Metaparameter Choices:

For machine learning algorithms, hyperparameter optimization is critical. For the DQN algorithm, design choices for implementing the agent include selecting the ***function approximator,*** deciding how to ***update action values,*** and how to do ***exploration.***

Function Approximator: For a function approximator it is often best to start simple. One simple option would be a tile coder that represents features in a continuous state space. Unfortunately, for a tile coder, the number of features grows exponentially with the input dimensions, and in this case, the state space has 8 dimensions. This means that in this case, a ***neural network*** is a good choice for the function approximator.

Activation Function: If we use a sigmoidal function like tanh, we may run into ***vanishing gradient issues.*** **ReLUs** are a popular choice that make sense to use for this problem.

Training the Network: To use ***stochastic gradient descent,*** a number of hyperparameters would need to be tuned to be effective. Due to this, I explored other options besides vanilla SGD. One option would be ***adagrad.*** Adagrad decays values down to zero which is problematic for non-stationary learning. ***RMSProp*** utilizes information about the curvature of the plots to improve the descent step. ***ADAM*** is a good choice because it combines the curvature information from RMSProp with ***momentum*** which helps speed up learning.

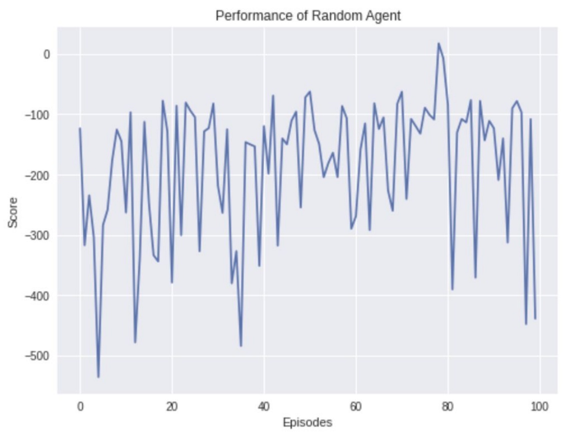
Exploration Strategy: One option would be to use ***optimistic initial values.*** This is a good option for a linear function approximator with non-negative values, but does not work well with neural networks. In this case, I chose to implement ***Epsilon Greedy.***[15]

Tests, Experiments and Results

Testing

My primary strategy to test whether the agent was learning was to print out the average rewards over the past 100 episodes while the agent was training and then plot the average score over the past 100 episodes vs. number of episodes. The environment is considered solved if an agent averages 200 points over the past 100 episodes so this testing strategy helps evaluate whether an agent is demonstrating learning.

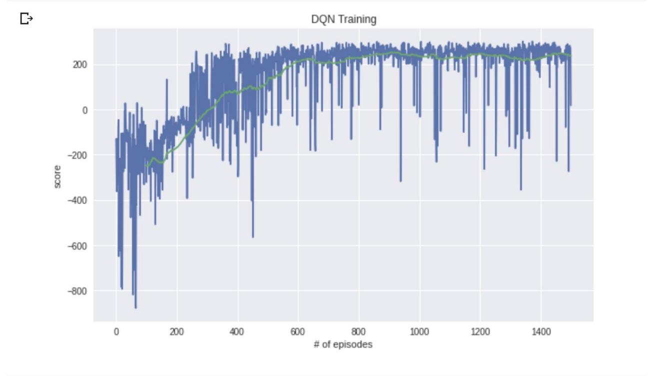
I started every model run by plotting the performance of a completely random agent. As expected, the random agent performed poorly and achieved a large negative score.



On the other hand, a plot like with an upward trend in scores shows that the agent is learning:

[15] This section draws from a lecture in the Coursera course: "A Complete Reinforcement Learning System Capstone", University of Alberta and Alberta Machine Intelligence Institute, Coursera, https://www.coursera.org/learn/complete-reinforcement-learning-system/home/welcome

```
plt.show()
```



Watching the demo video clips also can be useful for debugging because it is helpful to have a visual representation of what the agent is doing.

Experiments

To improve the agent's performance ***hyperparameter tuning*** is critical. Indeed, in the Mnih et al. 2015 article, hyperparameter tuning was critical to the agent's ability to perform well on Atari games. Given the timing constraints and my access to GPU resources, these experiments just scratch the surface of a thorough hyperparameter tuning approach.[16]

****Experiment with Neural Network Size***

Purpose:

For the LunarLander problem, a neural network with a single hidden layer is sufficiently powerful to represent the value function. More neurons mean more representational power, but they also mean a larger number of parameters that must be learned.

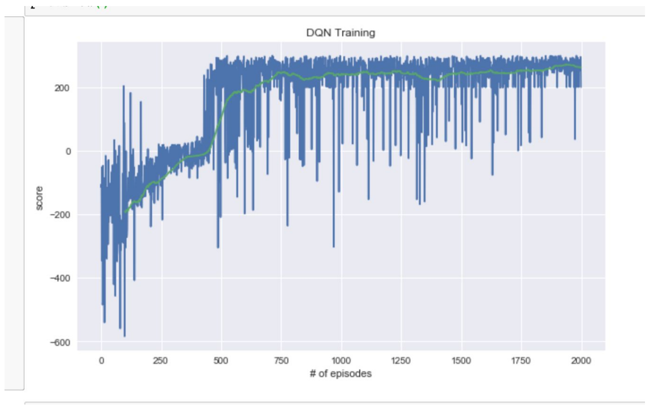
Methods for Neural Network Size Experiment:

To experiment with neural network size, I ran three sets of 2000 episodes with networks that had hidden layers with input/output dimensions 32/16, 32/64, and 256/128. I kept all other hyperparameters fixed. For each set of 2,000 episodes, I recorded the average reward for each hidden layer size.

Results of Neural Network Size Experiment:

The ***network architecture*** with a fully-connected input layer with 8 units and an output of 256 units followed by a fully connected hidden layer with input size 256 units and output size of 128 units followed by an output layer with an input unit size of 128 units and then output size of 4 units (corresponding to the number of actions), performed best with an average reward of 247.43 across the three 2000 episode trials.

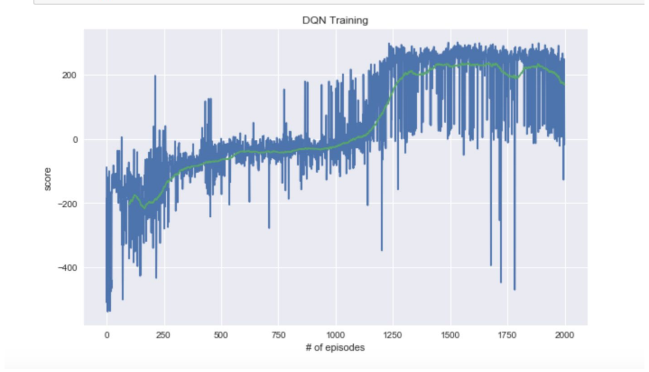
Here is a plot of the agent training with the largest neural network in the experiment:



Compared to the smallest neural network in the experiment:

[16] Mnih et al. "Human-level control through Deep Reinforcement Learning," *Nature,* v. 518, 2015, https://applied.cs.colorado.edu/pluginfile.php/15569/mod_resource/content/1/MnihEtAlHassibis15NatureControlDeepRL.pdf

```
plt.show()
```



Performance Comparison Table:

Neural Network Hidden Layer Size (Input/Output Dims)	Average Rewards for 3 trials of 2000 episodes
12/16	197.7
12/64	231.13
256/128	247.43

Problem Solving: Given that solving the LunarLander environment requires an average score of 200 across the past 100 episodes, I realized that the smallest neural network with a hidden layer with input/output dimensions of 32/16 provided insufficient representation. With this architecture, the agent had an average reward score of 200.76 across the three trials. The lowest average score was 197.7 which doesn't successfully solve the environment.

Summary and Interpretation:

To ensure that the agent will learn how to solve the environment, it is important to use more neurons in the hidden layer. Indeed, utilizing the largest number of neurons tested allowed the agent to achieve average reward scores of 200 over the past 100 episodes after between 500-700 episodes instead of over a thousand episodes like the middle and smallest number of neurons.

Experiment with Replay Buffer Size

Purpose: As the Naive Deep-Q Learning implementation showed, without a ***replay buffer, correlated elements*** made the neural network unstable. Mnih et al. 2015 demonstrated how having a replay buffer was an important factor for pairing RL and Deep Learning. The larger the replay buffer size, the less likely correlated elements will be sampled so the neural network will likely be more stable. However, a large experience buffer also requires a large amount of memory and slows down the training time.[17]

A replay buffer fills up with an agent's memories and then overwrites old experiences with new experiences. With a small buffer, the agent will care most about the things it has experienced recently. This could be limiting with regards to the agent's performance. However, having a really large buffer could also have downsides if the agent has recent promising trajectories. On the Stack Exchange thread, "How large should the replay buffer be?" one commenter gives a good analogy to how a large replay buffer is like a college student having all of the books for their entire academic career on the bookshelf. If the student recently decided to pursue a programming career but needs to sample from all of their books including their first-grade books that is a disadvantage. [18]

Methods for Replay Buffer Experiment:

To experiment with the replay buffer size, I ran three sets of 2000 episodes with a 1000, 20k, and 100k replay buffer size keeping all other hyperparameters fixed.

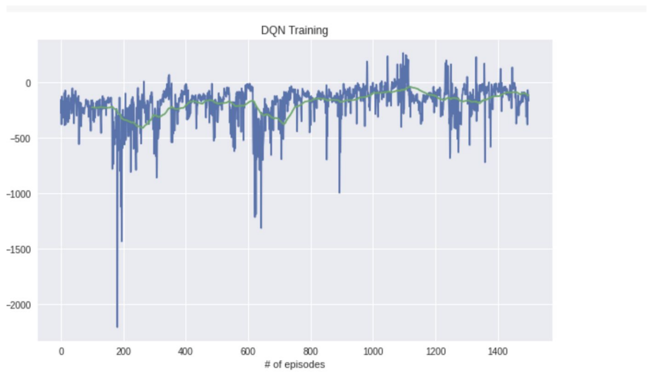
Results of Replay Buffer Size Experiment

With a 100k replay buffer, the average score over three trials of 2000 episodes was 247.43.

[17] Mnih et al. "Human-level control through Deep Reinforcement Learning," *Nature,* v. 518, 2015, https://applied.cs.colorado.edu/pluginfile.php/15569/mod_resource/content/1/MnihEtAlHassibis15NatureControlDeepRL.pdf

[18] "How large should the replay buffer be?", Stack Exchange Post, https://ai.stackexchange.com/questions/11640/how-large-should-the-replay-buffer-be

Without any replay buffer, the Naive Deep Q implementation for CartPole was unstable. In this case, using a small buffer of size 1,000, the ship didn't learn how to land within 2000 episodes and achieved a negative average reward.



The 20k buffer size achieved average scores of 240 over the 2000 episode trials. This was only slightly lower average performance than the 100k buffer size which had an average score 0f 251.47. However, with the 20k buffer, the agent didn't solve the environment until about 1,000 episodes whereas the agent solved the environment in about 800 average episodes for the 100k buffer. I therefore kept the buffer size at 100k.

Performance Comparison Table:

Buffer Size	**Average Reward After 3 Trials of 2000 Episodes Each**
20k	240.18
100k	251.47

Problem Solving/Interpretation: This experiment affirmed the importance of having an adequately sized replay buffer. It was important to see that the environment couldn't be solved with a small buffer of size 1,000. Other researchers have also found significant performance drop offs for buffers that are too large. It would be interesting to continue increasing the buffer size to see where that occurs for this agent.[19]

For future work, I could also optimize the batch size. One problem that I did have to consider during the implementation was figuring out how to sample before the agent has had enough experiences to fill the buffer. Sampling from the smaller buffer increases the risk of correlation. It seems like somewhat of a waste to not learn until the buffer is full, but ultimately this was the approach that I utilized.

4. Conclusion

Discussion:

This project was a thoroughly rewarding introduction to the OpenAI Gym toolkit and the PyTorch machine learning library. I used a simple TD Algorithm with an Epsilon Greedy Action Selection Strategy to solve the FrozenLake environment. Then I moved on to an environment that couldn't be solved with a tabular approach. After gaining a better understanding of strategies to overcome the deadly triad, I implemented a Deep Q-learning agent with a replay buffer and two networks: a target network and a prediction network. With some tuning to the size of the neural network and the replay buffer, this agent solved the LunarLander environment in under 1000 episodes and achieved an average reward around 250 for 2000 episodes.

Some of my biggest debugging issues came from needing to brush off my linear algebra and make sure that tensors were the right dimensions. I also spent a good deal of time

[19] Zang and Sutton, "A Deeper Look at Experience Replay," arXiv:1712.01275, 2017, https://arxiv.org/abs/1712.01275.

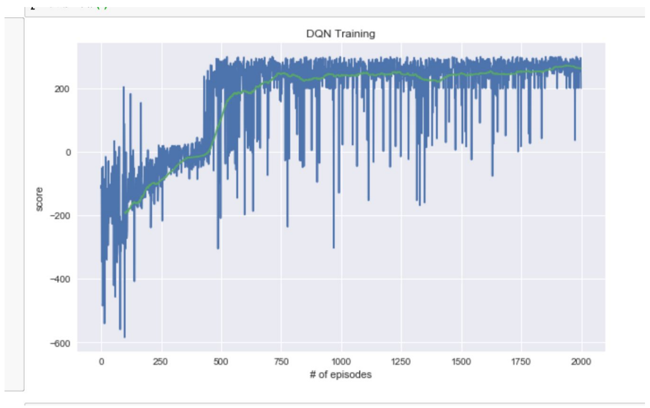
dealing with PyTorch's pickiness about data types. One of my primary reasons for sticking with PyTorch was the GPU support. With the ability to load the dataset and model parameters into GPU memory, we have better parallelism for computing matrix multiplications, and it also has automatic differentiation.[20]

Suggestions for Future Work: In the University of Alberta's Reinforcement Learning Capstone Course on Coursera, the lecturer advises that **Sarsa** would be a better approach to solving the LunarLander environment than Q Learning because learning Sarsa learns an epsilon soft policy which is a better choice than learning a deterministic policy like Q Learning.[21]

It is my understanding that a soft policy has "some usually small but finite, probability of selecting any possible action" [22] A deterministic policy means that for every state the agent has a clear defined action to take.[23] I would like to understand this reasoning on algorithm selection better, and it seems like the Sutton and Barto Reinforcement Learning textbook or other source like that would be a good resource to explore. The Coursera lecturers advise that Expected Sarsa is typically more robust than Sarsa so that would be the algorithm that they would advise to solve this particular problem.[24]

For future work, my plans would be to compare the performance of an Expected Sarsa implementation to a Deep Q Learning implementation for the LunarLander environment and see which performed better with regards to average scores and average number of episodes required to solve the environment.

Another comparison that I would like to make is switching from using an Epsilon Greedy action selection strategy to Softmax. Even when an agent achieves a relatively high score solving the environment, the agent will still have an average over the past 100 sessions that is close to zero or even negative.



Epsilon Greedy is equally likely to explore an action with a highly negative value as a less bad action. On the other hand, for ***Softmax,*** the probability of selecting an action is proportional to the value of the action so the agent is less likely to explore actions that are really bad. I would be interested to explore whether the training graph for a Softmax action selection

[20] "Linear Algebra for Deep Learning," Sebastian Raschka, Stat 479, Deep Learning course at the University of Wisconsin, https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L04_linalg-dl_slides.pdf.

[21] "A Complete Reinforcement Learning System Capstone", University of Alberta and Alberta Machine Intelligence Institute, Coursera, https://www.coursera.org/learn/complete-reinforcement-learning-system/home/welcome

[22] "What are soft policies in reinforcement learning?" StackExchange, https://stats.stackexchange.com/questions/342379/what-are-soft-policies-in-reinforcement-learning

[23] "A Complete Reinforcement Learning System," University of Alberta and Alberta Machine Intelligence Institute.

[24] "What is the difference between a stochastic and deterministic policy?" Stack Exchange, https://ai.stackexchange.com/questions/12274/what-is-the-difference-between-a-stochastic-and-a-deterministic-policy

strategy had less low dips than one utilizing Epsilon Greedy.[25]

This implementation could also benefit from thorough hyperparameter tuning. When it wasn't possible to run experiments, I tried to use rules of thumb and default values. It would be useful to employ a larger parameter sweeping procedure, creating parameter sensitivity curves and test ranges of a number of different parameters.

References

*Kim, Geena. "Deep Reinforcement Learning Intro," Lecture at University of Colorado Boulder, Fall 2020.

*Mnih et al. "Human-level control through Deep Reinforcement Learning," *Nature,* v. 518, 2015, https://applied.cs.colorado.edu/pluginfile.php/15569/mod_resource/content/1/MnihEtAlHassibis15NatureControlDeepRL.pdf

*OpenAI Gym Documentation, https://gym.openai.com/envs/

*Raschka, Sebastian. "Linear Algebra for Deep Learning," Stat 479, Deep Learning course at the University of Wisconsin, https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L04_linalg-dl_slides.pdf

*University of Alberta and Alberta Machine Intelligenc Institute, "A Complete Reinforcement Learning System Capstone", Coursera, https://www.coursera.org/learn/complete-reinforcement-learning-system/home/welcome

"Reinforcement Learning," Wikipedia, https://en.wikipedia.org/wiki/Reinforcement_learning

*Russell and Norvig, *Artificial Intelligence: A Modern Approach,* 4th edition, 2020.

*Silver et al. "Mastering the game of Go without human knowledge," *Nature* Vol 550, October 2017.

*Tabor, Phil. "Modern Reinforcement Learning: Deep Q Learning in PyTorch," https://www.udemy.com/course/deep-q-learning-from-paper-to-code/.

*"Temporal Difference Learning," Wikipedia, https://en.wikipedia.org/wiki/Temporal_difference_learning.

*"What are soft policies in reinforcement learning?" StackExchange, https://stats.stackexchange.com/questions/342379/what-are-soft-policies-in-reinforcement-learning

"What is the difference between a stochastic and deterministic policy?" Stack Exchange, https://ai.stackexchange.com/questions/12274/what-is-the-difference-between-a-stochastic-and-a-deterministic-policy

*Zang and Sutton, "A Deeper Look at Experience Replay," arXiv:1712.01275, 2017, https://arxiv.org/abs/1712.01275.

The code for this project is available at: https://github.com/cailyn-craven/OpenAIFinal

[25] "A Complete Reinforcement Learning System," University of Alberta and Alberta Machine Intelligence Institute.