Deep Q-Learning for Lunar Lander Using Pytorch and OpenAI Gym

Name: Samuel Perry Email: shperry@crimson.ua.edu

***Abstract* — Reinforcement learning has been a widely used approach to solve many problems with well-defined environments and reward functions, such as simple video game controls. This paper uses OpenAI Gym to implement the Q- Learning algorithm, a widely used reinforcement learning technique, for the popular game Lunar Lander. This paper implements the Q-Learning algorithm in a neural network architecture using Pytorch, thus creating a Deep Q-Network (DQN). The results demonstrate the promising ability of a DQN to solve the Lunar Lander game without prior human knowledge despite the complexity of the in-game physics.**

Keywords — Deep Q-Network, Reinforcement Learning, OpenAI Gym

I. Introduction

Reinforcement learning has been widely used to solve many problems that possess well-defined environments. It allows researchers to create policies and agents with little-to-no available expert data. When very little data exists surrounding an event or region of research, traditional machine learning models may not perform well or generalize well for the problems. In situations where little data exists surrounding an event or area of research, reinforcement learning offers the possibility of self-play, allowing models to generate their own data and learn optimal policies in a self-supervised manner. This capability has made reinforcement learning an important tool that can be adapted to solve many problems in various fields. Reinforcement learning has the ability to learn through trial and error, using feedback from the environment to guide the learning process. By optimizing the trade-off between exploration and exploitation, reinforcement learning algorithms can learn optimal policies in complex environments, even without complete knowledge of system dynamics.

Fig. 1. Lunar Lander Environment from OpenAI Gym

One of the most popular uses of reinforcement learning is in creating control agents for simple video games. In video games, there is typically a score or an objective that can be easily visualized and the environment is already created. Because video games generally have these well-defined environments and reward functions, reinforcement learning problems can be more easily defined for these types of environments. An agent can learn the optimal policy to maximize score and explore the entirety of an environment. This allows the agent to learn the optimal policy and make decisions in a wide range of scenarios.

OpenAI Gym has defined many environments for games that can be solved using reinforcement learning. Many different algorithms can be used and applied in the environments from OpenAI Gym, which makes it an ideal library for implementing reinforcement learning. With an understanding of reinforcement learning algorithms, researchers can solve environments using rewards that OpenAI has provided in the library and create models that can optimally play games without the need for human intervention.

II. Experimental Environment

A. OpenAI Gym

In order to apply reinforcement learning techniques to a given problem, it is essential to define the environment in which the problem is situated. The process of environment definition typically involves the specification of relevant parameters such as the state and action spaces, which are crucial for the successful implementation of reinforcement learning algorithms. To this end, I utilized OpenAI Gym, a freely available software package that is specifically designed for research in reinforcement learning [1]. This software offers numerous pre-built and user-built environments that simulate a variety of games and problems, each with pre-defined parameters, which simplifies the task of programming a fully functional environment. This, in turn, enables researchers to focus on the development and performance evaluation of their reinforcement learning algorithms in a more streamlined manner.

This study focuses on developing a reinforcement learning solution to the popular game Lunar Lander. This game is based around attempting to land a spacecraft onto the surface of the moon. The objective of the game is relatively simple, but due to the game's complex physics-based dynamics, this is a formidable task. With the complexity of the game dynamics as well as the simplicity of the objective, Lunar Lander is an ideal proving ground for a reinforcement learning solution.

B. Lunar Lander Environment

The Lunar Lander environment library from OpenAI Gym contains a working Lunar Lander game that can be rendered and

played, as well as pre-defined parameters that can be found in the documentation [2].

The action space is comprised of 4 discrete actions represented by integers: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine.

The observation space is an 8-dimensional vector that contains the lander's x and y coordinates, x and y linear velocities, the lander's angle with respect to the default angle, the lander's angular velocity, and two boolean variables that account for both legs of the lander and whether contact is currently made with the ground.

The lander starts at the highest y value of the environment in the center of the platform. There is a randomly generated force applied to the lander for randomizing the actions needed. Gravity is always acting on the lander and can be changed in the environment if need be. The game has ended if the lander's body comes into contact with the moon (crashed), the lander leaves the bounds of the x coordinates (x coordinate is greater than 1), or the lander has fallen asleep according to the Box2D specifications for the environment (it no longer moves and has not collided with another entity).

For the reinforcement learning algorithm to function, a reward needs to be defined for the environment. The Lunar Lander environment has both positive and negative rewards for the agent to learn the correct pattern of play. An agent is penalized when it moves away from the landing pad in the center, given a -100 reward if it crashes, and given -0.03 reward for every engine that is firing in the current frame. An agent is rewarded +10 for each leg making contact with the ground, +100 points for landing safely, and +200 points for landing safely on the landing pad.

III. Related Work

This section looks at several related works that pertain to implementing reinforcement learning using OpenAI gym as well as current reinforcement learning developments that have been made using OpenAI Gym.

A. Reinforcement Learning using OpenAI Gym

Using [3], a reader can learn the basics of using OpenAI Gym to implement reinforcement learning. It explains how to use the popular machine learning library Pytorch to solve reinforcement learning problems with Atari games that are available in the library. As this paper implements a DQN using Pytorch, [3] is an important resource to learn many basic implementations of reinforcement learning algorithms in Pytorch as well as using the OpenAI Gym environment.

B. Q-Learning in OpenAI Gym

Another common area where reinforcement learning can be successfully applied is robotics. In robotics, one of the most common pieces of software used is the Robot Operating System (ROS) and the Gazebo simulator. In order to aid in reinforcement learning for robotics, [4] created an extension of OpenAI Gym that allows ROS and Gazebo simulations to be tested and run in the OpenAI environment. This allows robotics researchers to have a replicable environment to test and control reinforcement learning techniques used in robotics.

Additionally, this paper covers both Q-Learning and SARSA reinforcement learning algorithms in the new extension created in their work. The paper offers successful ways to implement Q-Learning and SARSA algorithms and compares the results of these differing algorithms in the environment.

C. Quantum Reinforcement Learning using Lunar Lander

Classical reinforcement learning has been widely used to solve many different areas and problems with successful results. The speed and development of classical reinforcement learning has widely slowed, as there is very little need or ability to push the current algorithms further. One suggestion for improving classical reinforcement learning comes from quantum computing. Quantum reinforcement learning is separate from classical reinforcement learning, as it uses algorithms and hardware that are specifically designed for use by quantum models that improve speed and accuracy of the algorithms.

[5] shows the results of using quantum reinforcement learning with OpenAI Gym environments. More specifically, it uses the Lunar Lander game to demonstrate promising results of quantum reinforcement learning as compared to classical reinforcement learning. This is the first successful quantum agent that can complete the task of Lunar Lander on small quantum computing hardware. The model uses a single-qubit-based variational quantum circuit without entangling gates and a classical neural network to select actions and successfully play Lunar Lander. This paper is extremely important for future development of reinforcement learning, as it was also the first agent to successfully run on IBM quantum machine, a task that was previously unattainable. The developments from this paper show promise in updating and growing the field of reinforcement learning as our computational power grows.

IV. Methods

This section describes the methods chosen in this study as well as the rationale for selecting the methods.

A. Q-learning

One of the early breakthroughs for estimating an optimal policy is the Q-learning algorithm, a value-based reinforcement learning algorithm which is defined by the following equation from [6]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

In the Q-learning approach, the action-value function that is learned directly approximates the optimal action-value function. The action-value function of the agent is updated directly through a trial-and-error adjacent system where the observed reward at the next state directly affects the value of the state-action pair at the current state. This leads the approximated action-value function to converge to optimality given sufficient sampling episodes.

In modern reinforcement learning, the optimal policy is approximated using a neural network, although the basic algorithm for Q-learning is the same. When the agent uses a neural network to converge to the optimal policy, we call this Deep Q-learning. A Deep Q-Network (DQN) is used to maximize rewards from a given environment that an agent operates within. With predefined rewards within an environment, a DQN explores and exploits the environment to discover optimal play through repetition and play.

I chose Q-learning because it is a Temporal-Difference Learning (TD) technique that has seen great success in many different scenarios. TD techniques combine ideas from both Monte Carlo (MC) Methods as well as dynamic programming (DP) methods, which allows a researcher to get benefits of both MC and DP. Additionally, using a deep learning approach with a DQN allows for more rapid and accurate approximation of the optimal policy for playing Lunar Lander and saves storage and computation. Finally, many implementations of reinforcement learning in Lunar Lander and other games in the OpenAI Gym tested Q-learning, meaning that Q-learning is a widely used approach among researchers. It was these main reasons that lead me to use Deep Q-learning when searching for the optimal policy for Lunar Lander.

B. *Architecture*

The architecture that was tested in this study is two linear neural networks with four layers: an input layer, two hidden layers of 128 nodes each, and an output layer. One neural network represents the policy approximator that will be used for decision making and the other represents the target policy approximator that we will be attempting to approximate. The layout is shown in figure 2. The neural network was built using Pytorch and is similar in structure to the tutorials available at [7,8] with modifications to reflect a more successful Lunar Lander agent. Additionally, the initial tests with a one hidden layer architecture are explored in the *Implementation and Results* section. This model had the exact same architecture as the current model except one hidden layer was removed for training.

The input layer takes in the 8-dimensional observation vector that represents the x coordinate, y coordinate, x velocity, y velocity, angle (relative to starting position), angular velocity, left leg boolean, and right leg boolean.

The input layer is then fully connected to the first hidden layer, which consists of 128 nodes. The first hidden layer uses the leaky ReLU activation function. During testing, the leaky ReLU model performed better than the model with ReLU, but due to computational constraints, only one test could be run. The first hidden layer is fully connected to the identical second hidden layer. Models with two hidden layers generally performed better than the models with only one hidden layer with small increases in time of training. The final model uses two fully connected hidden layers of 128 nodes with leaky ReLU activation functions due to the limited testing possible.

The second hidden layer is fully connected to the last layer which, is the output layer. Because the output layer decides on the action to be taken, it consists of four nodes, one for each of the possible actions: do nothing, activate left engine, activate right engine, and activate main engine.

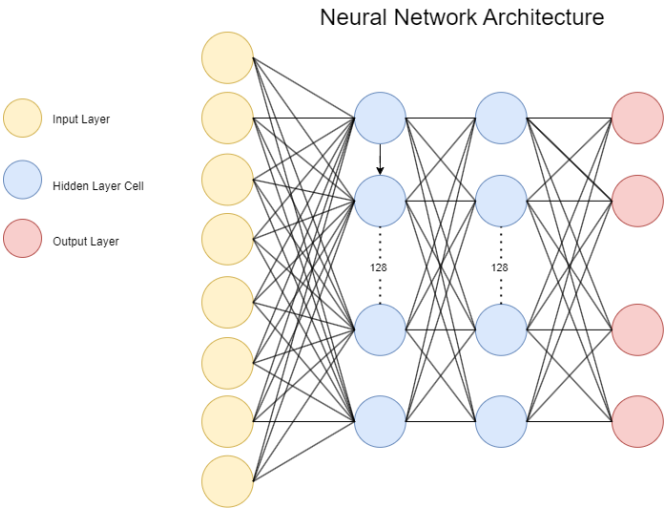

Fig. 2. Architecture of the DQN used in the study with input nodes (yellow), hidden nodes (blue), and output nodes (red).

C. *Training*

In training, I tested many different hyperparameters and selected them based on the performance of the model on the Lunar Lander game. The performance was determined by the level of reward received per epoch as well as the number of epochs to reach the highest reward.

In training, the batch size is set to 128, as the training on this batch size was relatively quick and performed very similarly to other batch sizes.

The policy network was trained on the batches with the final state masked, whereas the target network was trained on the full episodes. This ensures the target network has the full reward and values to work with and approximate the values of the next state while the policy network will need to learn these values through training.

The discount factor for the rewards is set to 0.99. The agent receives the largest rewards when it makes contact with the ground, either in a crash or successfully landing. Additionally, the agent will receive a reward every frame until termination based on engine use and distance from the center pad. The discount factor needs to be relatively large to account for the largest of rewards only manifesting at the end of the episode. With larger reward discount factors, the future rewards will be more heavily weighted. The smaller the discount factor, the more the immediate reward matters. Because the agent samples so quickly and the larger rewards are not immediate, this study uses a larger discount factor for training.

The learning rate is set at a constant 0.0001 for the entirety of the training. This aligns with a large quantity of papers and model architectures that were discovered when looking at other resources similar to this study. Additionally, the static learning rate produced successful results, and as such, a dynamic learning rate was not considered as this would add unnecessary overhead.

One issue that is common in deep Q-learning is that Q-learning can be unstable since the target value is calculated using the network that is being updated, causing the updates to be prone to divergence. A solution to this instability is using “soft”

target updates where the target network is not updated every training step [9]. This allows the target network to slowly converge and helps relieve the update divergence that can be common in Q-learning. The "soft" target update is done through τ, where $\tau << 1$. In this study, τ mimics the study done in [9] and sets τ to be 0.001, where updates to the target network are done much slower. This slows learning but greatly increases the stability of the model and leads to convergence.

Finally, one key topic in reinforcement learning is the trade-off between exploration and exploitation. This is the trade-off that allows for reinforcement learning to be such a powerful tool. This trade-off dictates how often a model will abide by the value function to make the "correct" decision and how often it will explore states that appear to be worse in the current value function. To achieve this, the model uses a decaying exploration rate that is large at the beginning of training, but decays to be smaller when the model has more closely approximated the correct result. According to prior research, the exploration rate at the beginning of the training should be high to ensure that the model explores all possible states. However, as the agent becomes more knowledgeable and learns the environment and reward function better, the exploration rate should lower as the states that it views as worse, generally are worse according to the optimal solution. To achieve this, the exploration rate was set to 0.9, meaning that at the beginning, 90% of actions would be ignored for random actions. As the training continues, the exploration rate lowers to 0.05 so as to keep some exploration, but closely follows the policy derived from the previous steps of training.

The model was run for 200 epochs during regular training which took approximately 30 minutes on a 16 GB of RAM laptop. This was the only computational power used to test and develop the model.

The Adam optimizer was the optimizer used in this study. Adam is a popular optimization function that uses stochastic gradient descent to optimize objective functions. Adam comes with the Pytorch package and is easy to use. Additionally, it is computationally efficient, has little memory requirements, requires little tuning, and can deal with problems caused by noisy or sparse gradients [10].

Each epoch, an action is selected from the policy network given the current state of the environment. The agent then takes a step through the environment using the chosen action, and the observation, reward, and termination checks are recorded. For each batch, a list of state-action values is computed using the policy network that is training. Next, using the target network, we compute the value of the next state to be used in the expected state-action values. We use this computed next state values, multiply it by the discounting factor, and add the computed reward for the state. Next, the loss is calculated and backpropagated through the policy net to update the parameters. The Smooth L1 Loss is used in computing the loss of the policy net as it is less sensitive to outliers and potentially prevents the exploding gradient problems [11]. Additionally, this was a common loss used among projects implementing reinforcement learning in the OpenAI Gym environments. Finally, the policy net is updated, and the target net is updated using the "soft" update policy until the model is done training.

V. Implementation and Results

Before beginning implementation of a reinforcement learning technique, I first had to install the necessary requirements for the project. First, I installed OpenAI Gym and attempted to run an example script for the game Pong on a local machine. Possibly due to licensing, the script would not run, as OpenAI Gym had not been installed correctly. To remedy this, I manually downloaded OpenAI Gym and attempted to run the script. The script worked with the manually downloaded library and development work could then begin.

Another issue that limited the testing and training step of the model was the lack of computational power. The only owned machine that could run and train the model was a personal computer with 16 GB of RAM. This is plenty of RAM for model training and testing when fully allocating all 16 GB to the training. However, with many different ongoing projects, computing power could not be completely devoted to training and testing. As such, there could be more hyperparameters and model structures to test, and in the future, this will be a great step to continue the project.

With the issues resolved, the model could be trained, tested, and given a performance evaluation. The two models I chose to test during training had the exact same hyperparameters during training, but the architectures were different. The first model I tested was a one hidden layer architecture. The performance is shown below in figure 3. The blue line represents the reward at each epoch and the orange line is the average over the last 100 epochs. From the OpenAI website, an agent generally receives around a 100-140 reward over an episode where it has successfully landed on the pad.

This model performed well, but there are large fluctuations that can be seen as the model is tested. This shows that the one-layer model did not converge to a specific solution and did not find the most optimal solution. Additionally, the model did not converge to a stable solution in the 200 epochs of training and the performance varied greatly from epoch to epoch. Towards the end of the training, there was a slight trend upwards. This means that the model had not finished training and could have been optimized even further if given more time. Although the

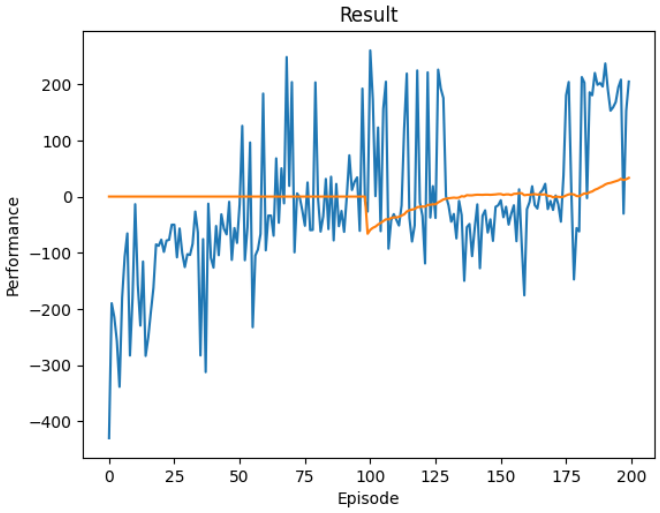

Fig. 3. Performance of the model in Reward vs Episode format. The blue represents performance at each episode, the orange is a moving average of 100 episodes.

model appeared to be learning, it does not seem the model has converged to an optimal solution for the Lunar Lander game.

The model that is talked about in the training and methods section is the two-layer model. This model is the best performing model of the ones that were tested and had the best convergence. This can be seen in figure 4 below.

In comparison to the one-layer model, the two-layer model performs significantly better and converges much better. This convergence led to more consistent play after the training had completed, and the model performed better than expected. Both models perform better than the expected reward from OpenAI of around 100-140, but the two-layer model receives rewards almost 100 points higher than the first model. The model learned much quicker than the previous one, reaching similar levels of performance at around the 25-episode mark. Additionally, the model performed much more consistently on the performance metrics, meaning it had more closely converged compared to the one-layer model that was originally tested. From a visual perspective, this model was much improved over the one-layer model, and it learned the environment and how to play the game much better and faster than the one-layer model.

Despite the improved performance, it is clear to see that the model has not fully converged and could be improved upon. Although not as large as in the one-layer model, major fluctuations in performance around the model still exist. In the middle, around 50-125 episodes, the model improves these fluctuations and seems to have learned the environment well. However, the model then sees these large fluctuations again before ultimately settling closer to the 200-300 reward range. Through more training, the fluctuations will most likely go away, although more steps may be taken in the future to guarantee convergence of the model.

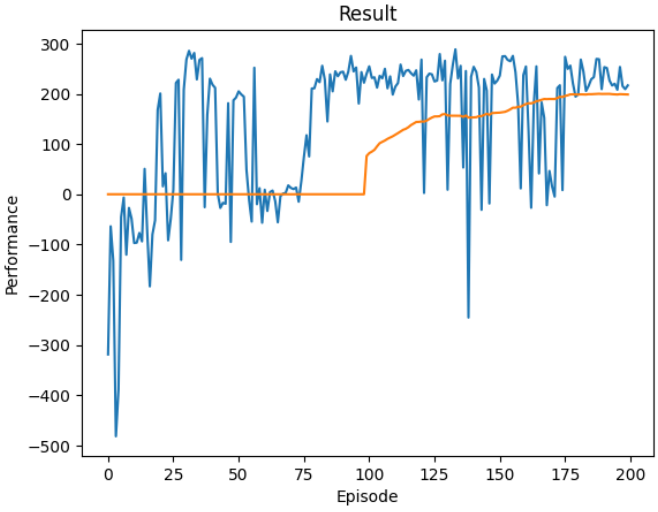

Fig. 4. Performance of the two-layer model in Reward vs Episode format. The blue represents performance at each episode, the orange is a moving average of 100 episodes.

VI. Discussion and Conclusion

The results of the study were surprising, and I did not expect the model to perform as well as it did. I was pleased with the result and did not expect the model to learn such a complex game so quickly. Visually, the model performed incredibly well and learned the game very rapidly. After approximately 25 episodes, the model seemed to outperform my capabilities of the Lunar Lander game, and with more training, the model became even better.

From this project, I learned how to use an environment and define a problem in the frame of reinforcement learning. Additionally, I learned how to adapt problems and resources to be framed as a reinforcement learning problem. Finally, I learned a lot about hyperparameter tuning and using Pytorch to apply a deep learning solution to a reinforcement learning problem.

Moving forward, I will use the reinforcement learning mindset to solve problems in other areas of research and my career. Being able to frame problems as reinforcement problems and apply a solution is invaluable in the coming future, as these skills will allow me to be useful in research environments.

Finally, as my goal is to work with artificial intelligence in industry, I believe the lessons from this class will be useful to me for many years to come. I can take what I have learned, define a problem the correct way, and apply it in my job to find the best artificial intelligence solution with the best approach.

Acknowledgment

Code: https://github.com/shperry03/DQN-Lunar-Lander
Video: https://youtu.be/F1Qm8TmDW84

References

[1] G. Brockman et al., “OpenAI Gym,” CoRR, vol. abs/1606.01540, 2016, [Online]. Available: http://arxiv.org/abs/1606.01540

[2] O. Klimov, “Lunar Lander” (v2), 2022, [Online]. Available: https://www.gymlibrary.dev/environments/box2d/lunar_lander/

[3] P. Palanisamy, Hands-On Intelligent Agents with OpenAI Gym: Your guide to developing AI agents using deep reinforcement learning. Packt Publishing, 2018. [Online]. Available: https://books.google.com/books?id=BAlnDwAAQBAJK.

[4] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. 2017.

[5] J.-Y. Hsiao, Y. Du, W.-Y. Chiang, M.-H. Hsieh, and H.-S. Goan, Unentangled quantum reinforcement learning agents in the OpenAI Gym. 2022.

[6] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, Second. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[7] A. Paszke, M. Towers, “Reinforcement Learning (DQN) Tutorial.” [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[8] Y. Feng, S. Subramanian, H. Wang, S. Guo, “Train a Mario-Playing RL Agent.” [Online]. Available: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html

[9] T. P. Lillicrap et al., Continuous control with deep reinforcement learning. 2019.

[10] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization. 2017.

[11] Pytorch, “SmoothL1Loss.” [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html#torch.nn.SmoothL1Loss