Solving the Lunar Lander Environment with DDQN

Evan Jones
Georgia Institute of Technology College of Computing
Git Hash:0655da2c3dad163b322d39e982210243962dab69
Georgia Institute of Technology
Atlanta, GA USA
evanjones@gatech.edu

I. Introduction to Lunar Lander Environment

For this project we will be attempting to solve Open AI's Lunar Lander V2 environment. The environment tests our agent's ability to avert disaster and successfully land a falling spaceship on the ground within a specified landing area under a 2D model of Newtonian physics engine. We will consider the environment solved when an agent has achieved an average score of greater than 200 over the past 100 previous episodes. For additional details regarding the specifics of the environment please see Open AI's accompanying *documentation*. With our environment introduction out of the way lets move onto the discussing the foundation of our agent Q learning.

II. Tabular Q-Learning

Introduced formally by Christopher Watkins in 1989, and derived from the foundational bellman equation, Q learning was one of the first widely successful model-free approaches to learning guaranteeing convergence to an optimal policy [4]. In the method's initial formulation (tabular Q learning) a table is initialized with arbitrary values and size SxA. Thus, each entry within this table represents one of the possible state action value combinations possible in a given environment. We then allow the agent to interact and explore the environment updating its values of state actions pairs with corresponding observed rewards upon reaching them.

Following repeated application of this equation throughout enough environmental interactions and certain conditions of exploration, we are guaranteed to converge on the optimal state action value pairs. This further allow us to easily find the optimal policy by acting greedily with respect to these optimal state action value pairs. However, Q learning itself is not flexible enough in this tabular form for us to apply as a standalone learning method in more complex environments. In tabular form we are fundamentally limited in representational power by how much physical memory is available to store all combinations of the state action space not to mention processing power in seeking and updating this table. However, taking lessons from the field of supervised learning, researchers have begun to use functional approximation paired with Q learning updates as a means to represent complex and vast spaces more efficiently and practically than in the traditional Q table. In the next section we discuss a few of such approximators and their relevant properties.

III. Functional Approximators

The advent of neural networks usage in reinforcement learning has vastly expanded the ability to represent complex spaces through nonlinear approximation. As proven by Kratsios [6] even a single nonlinear layer is capable of approximating any function. Furthermore, by stacking multiple layers of such nonlinear functions we further improve our ability to generalize even further. Although applying such approximations voids our convergence properties, several methods have been proven to be especially effective in practice despite this more specifically non-linear functional approximation via artificial neural networks. Given that our Lunar Lander Environment has a potential state action space that far exceeds our physical memory capacity, we choose to utilize this method of nonlinear functional approximation combined with Q learning to address our problem. In the next section we briefly discuss the DDQN models which will be the framework applied to solving this environment.

IV. DDQN

One of the most notable successes of this approach to reinforcement learning comes from Google Deep Mind's 2015 publication on DQN's [2] (DDQN's predecessor). The idea behind DQN put simply, is utilizing the vast representational power of deep neural networks to represent an approximation of the optimal action-value/Q function. This idea of using these two methods to complement one another was initially proposed by Riedmiller in his paper on Neural fitted Q networks [1]. However, the original implementation of NFQ was extremely difficult to scale practically due to several shortcomings. These shortcomings highlight the primary areas of concern for using neural networks with Q learning. Unlike in supervised learning where we aim to have independent samples to train on, in reinforcement learning we have correlation among our training observations temporally as each new state action is often dependent on prior actions taken by the agent. Additionally, in supervised learning, we typically assume that the function we are attempting to approximate is stationary which allows us to use non-linear methods such as neural networks to at least approach local minima. That is not the case in reinforcement learning. As we discussed earlier, each Q value update is changing our state action function toward the optimal function in the tabular case, but in non-linear approximation case this does not hold as we are only

approximating the optimal function. Additionally, we have the problem that updates to neural networks will propagate thought the entire network, and could alter portions of the network beyond a single state action pair making learning more unstable.

In order to avoid such issues Riedmiller proposed fitting networks on the entire training set de-novo and iterating over several hundreds of epochs until convergence. Although this resulted in more stability, it remains an infeasible approach for large neural networks such as one needed to solve our task efficiently. The choice of using entire training sets also prevents the benefits of online learning. It is with these issues with NFQ in mind, that Deep Mind created DQN and subsequently DDQN. Instead of using the entire training set for single batch fitting and repeated presentations training, Deep mind decided on three critical extensions of NFQ to remedy these problems while further preserving the underlying strengths of neural nets, and the ability to learn online. First, instead of having a single training set Deep mind decided to implement an experience replay buffer. This was a FIFO queue that stored agent experiences as they happened, and overwrote the oldest experiences when the buffer reached capacity. Secondly, storing experiences in this way the agent was now able to sample small mini-batches from this data buffer more often and incrementally within each time step. Furthermore, by having the buffer constantly filled with new experiences evaluated at the current policy this incoming new data would replace stale data making the buffer more representative of the agents recent progress. Finally, they added a second neural network known as the target network which would be essentially a copy of the original network with frozen weights that would be periodically updated to the weights of the network being trained actively online usually in an episodic fashion. By using a frozen target network to evaluate the accuracy of our online network in the learning process we further reduce the problem of non-stationarity by having fixed network targets to evaluate against. Through this process of freezing training and updating, Deep Mind was able to greatly improve upon each of the areas problematic under NFQ.

DDQN has its foundations in a paper originally published as an extension to Q learning known as Double Q learning by Hasselt [5]. The premise of the paper was that standard Q learning could be enhanced by further separating how the algorithm chooses the next action in greedy cases of action. It has been well known that standard Q learning has problems with the overestimation of state action values which can lead to less than optimal implementation. We could speed up learning if we were able to effectively reduce the amount by which our function over estimates these values thus making learning steps more direct. Under double Q learning the author suggests that we should utilize two Q tables which would randomly switch roles as being the provider of the argmax action or the able from which this argmax was pulled to represent the max q value. Deep mind took inspiration from this research and adapted DQN to take advantage of this observation [3]. However, instead of using several neural networks to break apart the argmax action choice within the q function they decided to take advantage of their already existing target networks. Instead of evaluating the max action with the online network they would evaluate the argmax of the target network state action values then select the action within the online network with that corresponding index. Given that DDQN represents only a slight extension over DQN with massive benefits to learning speed, and less divergence potential it was decided to use DDQN as our base framework for solving the lunar lander environment. In the next section, we discuss the general details concerning this projects specific implementation of the DDQN agent.

V. DDQN Implementation

As a whole the implementation of the agent can be broken down into three primary parts which interact with one another to accomplish our goal of solving the lunar lander problem. Firstly, we implemented a standalone class of function to handle the scheduling of epsilon or exploration decay as the agent progresses through the environment. Secondly, we implemented and efficient means of storing and sampling our experience tuples received from the environment with FIFO updating and fixed capacity. For this we utilized a double ended queue of fixed size. This allowed overwriting of old experience while also permitting efficient append and sampling functionality. Third, we needed to implement the heart of our agent which is the online and target neural networks. To accomplish this task we utilized Pytorch an open source deep learning framework that allows us to quickly develop and train neural networks. Further details on the model architecture are discussed in subsequent sections. Finally, and most importantly we implemented the general agent which would handle all of the logic behind interacting with the environment, buffer, and neural network. This is also the area this will handle our Q learning iteration logic and training and optimization steps. In order to implement the DDQN agent we relied upon a great deal of the implementation details supplied in Deep Minds DQN and DDQN papers [3] [2] so we will not duplicate the code here.

It is notable that as happens in many replication studies we encountered several pitfalls in the transition from pseudo code to real working code. Firstly, we attempted to implement the neural network of the model using Tensorflow the other major deep learning framework currently popular. However, despite eventual successful implementation, we decided to start from scratch again with Pytorch due to extremely slow evaluation and training speed. Although more efficient, Pytorch seems to have a few less safeguards when it comes to broadcasting tensor operations and explicitly tracing gradients than Tensorflow. Discovering these issues and solving them was especially cumbersome, but essential to a working model. The next implementation issue centered around dealing with certain environment timeout behaviors. In the case of an environment timeout due to length we needed to ensure that such an observation was not marked as terminal. This small change

often overlooked can be a commonly overlooked aspect in implementation that would prove catastrophic during training.

Finally, the last primary pitfall that affected the implementation of the agent had to do with the program we chose to use for logging our experimental results Tensorboard. However, we soon noticed that our models were not terminating as expected, and the data output from the application was discontinuous from a graphical perspective with overlapping and inconsistent plots. After significant time trying to find a bug within the agent implementation, we finally traced the issue back to a python issue with time keeping precision on windows operating systems. Thus data labeled with this non-unique and imprecise timestamps caused mangled data ordering. The issue was first addressed using a monotonic nanosecond level clock instead of the default time module. With these pitfalls in mind, it is evident that the replication of any complex agent such as this requires significant attention to detail and a vast amount of precision and patience. Now that our general implementation has been explained, we move onto the specifics of practical implementation by discussing the relevant hyper parameter configurations to give our agent the best chance of succeeding.

VI. Hyper Parameter Tuning & Selection

One of the reasons we decided to solve the lander problem with a DDQN was due to the flexibility afforded through the usage of nonlinear functional approximation, and generalization ability of neural networks. However, this flexibility also comes at the cost of tuning several hyper-parameters which allow are approximation to better fit our specific learning task. A non-exhaustive list of such parameters includes number of layers, batch size, size of layers, layer activation functions, optimizer choice, and many more. Furthermore, this list only contains our network hyper parameters, we still have additional tuning parameters on the Q learning side of things such as gamma and exploration.

Obviously finding the proper values for our problem is going to be a difficult task especially considering the large amount of time it take to train a single working agent. If we were to take a grid search approach this would take far too long, instead we will use certain heuristics to narrow the range of hyper parameters we are actively tuning. In order to do this I tried several arbitrary initalizations of various hyper-parameters most of which were similar to values used in DQN Nature publication. After some iteration I found that networks with 3 layers, epsilon exponential decay over one hundred thousand steps from one to .05, Rectified linear unit activation, RMSProp optimizer, hard target network updates at the episodic level, and Mean Squared Error loss resulted in a general model framework capable of learning under varying hyper parameter initializations. In order to compare hyper parameter tunings we need to have results of actual agent learning so it is for this reason we will use this general model framework as a control in the experiments with hyper parameters. I chose to experiment with three primary hyper parameters batch size, hidden layer size, and learning rate. My reasons for choosing these three was that during the semi

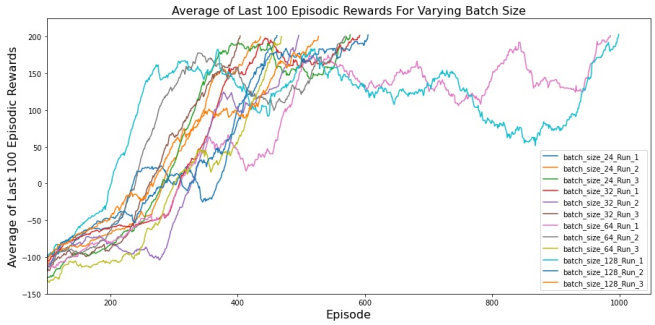

Figure 1. Average 100 Previous Rewards with Varying Batch Size

educated search for feasible models these three parameters seemed both flexible enough so that even through a range of values learning still occurs. In each of the following experiments we will evaluate our base model on three runs for each choice in a range of values for that given hyper parameter. Due to training time we will not consider the interaction effect of various hyper parameters with regard to one another as this would require an order of more model iterations that is not feasibly trainable within the time allotted for the project. Each of these runs will continue until we reach our termination conditions of greater than 200 in average reward over the past 100 consecutive episodes or the maximum episode cap of 5000, whichever comes first. Within each of these runs we will provide average reward over past 100 episodes as well as episodic reward which will be averaged over the 3 runs (for a given parameter value for visualization purposes). Below we begin the first of these experiments with batch size.

VII. Experiment 1: Batch Size

In this experiment we will evaluate our agent over three independent runs for each of the following batch size values 24,32,64,128, and 256. The following range of values was chosen based off of common heuristics in neural networks, and reasonable limits. Each of these batch sizes values determined how many experience tuples we will feed into each training iteration of our network. By feeding smaller batches into the network updates will be more subdued and consistent, but learning will be slower as each training iteration sees less of the replay buffer. However, large batch size will cause larger updates and see more of the replay buffer, but this comes at the cost of updates being too large which can be catastrophic for networks in a reinforcement learning environment. Additionally, given that our model will train at each time step, large batches will start to over sample our replay buffer which has a capacity of only 20,000 experience tuples. In Figure 3, we present the results of the various batch size effect on time to agent solving the problem and in Figure 4 we present the 3 run averaged episodic performance. Firstly, you may notice that the batch sizes for 256 are not displayed on either plot, this is for scaling reasons as this batch size converged in none of the three runs. This behavior alone should tell us that the step size of 256 does exactly what we expected, updates are

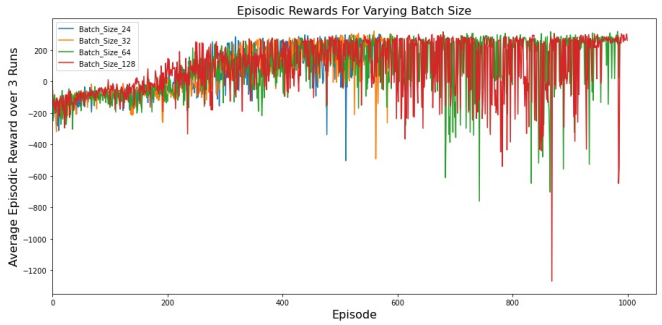

Figure 2. Episodic Rewards with Varying Batch Size

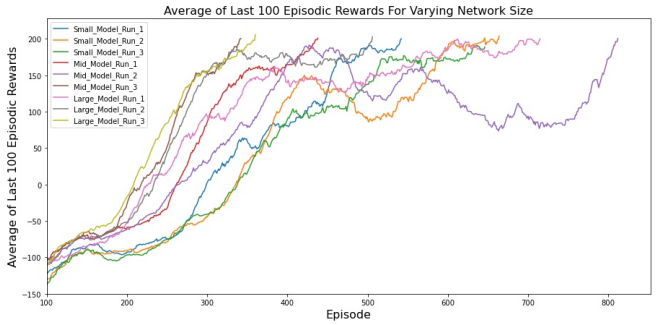

Figure 3. Average 100 Previous Rewards with Varying Network Size

too large and this causes the network to become erratic. In fact, the variance and performance of the 256 batch size was the worst agent by far across all tested parameters even the 100 episode moving average had to much variance to display.

Moving onto the valid values of batch size, we can see that although very noisy there seems to be some evidence that batch sizes of 32 provide the best results. Interestingly enough this is the same value recommended in most deep reinforcement learning literature. It also falls near the lower mid-range of our tested parameters. As we can see the tails of this range 24 and 128 have the two worst performing instances. Intuitionally, this makes sense as we would expect too small of updates with 24 and too large of updates with 128 both of which result in instability in some case or prolonged iteration when compared with the more reasonable choices of 32 and 64. Next, we move onto experiment two where we will try various sizes of hidden layers within our network.

VIII. Experiment 2: Hidden Layer Size

When it comes to neural networks the two primary areas that determine a networks ability to solve a task boil down to how large the network is as in the number of hidden units per layer, and how deep the network is in number of layers. Both of these two aspects of model architecture directly determine how much representational power our network has. However, in order to train our model in a reasonable amount of time we need a network deep and large enough to represent our problem effectively, but shallow and small enough to remain computational tractable. In order to test the impact of such model architecture alterations on our agent's learning performance, we decided to hold the number of layers constant at three which was a depth computationally feasible, and sufficiently expressive. We did however select three different values of hidden layer sizes which were combined into three models with overlapping ranges named small, medium, and large having respective hidden units of (64,32,24), (128,64,32), and (256,128,64).

As we can see in Figure 3 and Figure 4 each of our models was able to reach convergence within finite time. However it seems that our larger network on average seems to have some slightly stronger performance this could indicate that the problem itself or some specific dynamic within the

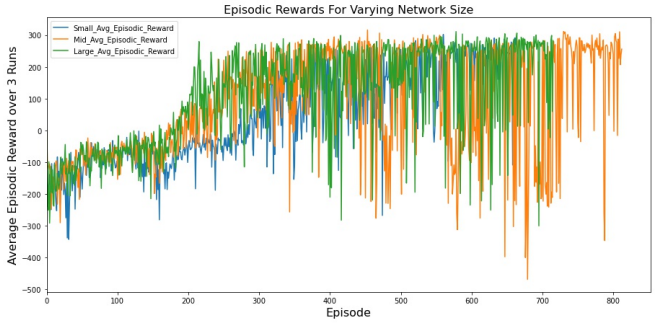

Figure 4. Episodic Rewards with Varying Network Size

environment is more easily captured with a larger number of hidden layer units. This makes sense as larger networks should have more representational power when compared to their smaller counterparts. Although not noted in the graphs, through the process of deciding on a hidden layer range we discovered that moving in either direction of size increment to our current parameter values makes the agent degrade significantly on the performance side. These runs are no included in the graphs as they lack convergence and would thus cause severe scaling issues, but this reinforces our intuition about network size relative to problem difficulty and how there is usually an operable range of hidden layer sizes beyond which we are unable to either effectively represent the entire problem or too large so as to drown out and over complicate the model. This is a key insight for anyone with interest in the field of deep reinforcement learning to keep in mind during the design of network architecture. Next, we move onto experiment three where we will analyze the changes to the learning rate parameter.

IX. Experiment 3: Learning Rate

For our final experiment we will be evaluating one of the most crucial parameters to having an agent that learns effectively when using a DDQN agent. The learning rate parameter determines how quickly and by what magnitude our agent will learn from our incoming target gradients. Under a low learning rate our network will be slower to absorb and adjust to a given mini-batch. The benefit of small learning rate becomes evident when considering some of the problems

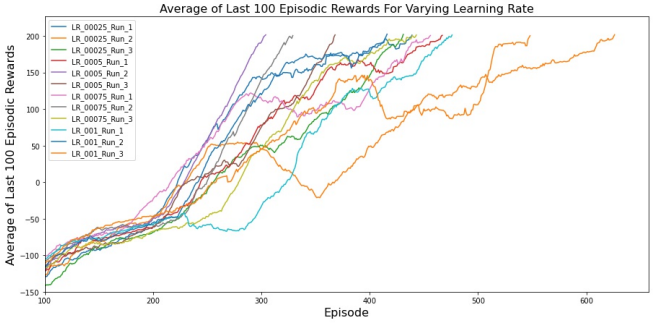

Figure 5. Average 100 Previous Rewards with Varying Learning Rate

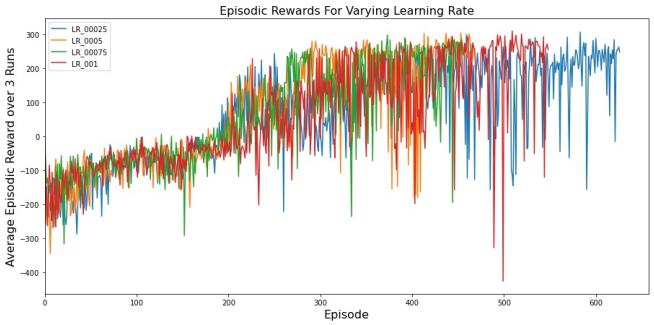

Figure 6. Episodic Rewards with Varying Learning Rate

of non-stationary targets discussed in section 3. By having a small learning rate we are less likely to make a large step away from this moving target function. Under larger learning rates, we will see learning happening quicker as the agent more quickly updates from gradients. However, in the presence of non stationary targets this could prove disastrous if our agent were to make a large update step in a direction away from our adjusted target. Reinforcement learning with deep neural networks can be difficult enough with non-stationary issues so the last thing we would want is to compound this problem by having an overly aggressive learning rate.

For this experiment we will be doing three runs for each of the following learning rates 0.00025,0.0005, 0.0001, 0.0025, 0.0075 ,and 0.01. The results of these experiments are depicted in Figures 5 and 6. As we can see from the plots, Our agent seems mirror the trend seem with the other parameters where a nice middle performs the best. That being said, I was quite surprised that a learning rate as small as 0.00025 was able to learn effectively at all due to steps being too small with only sporadically high rewards. Obviously my expectations were wrong it seems we would have to go much smaller to encounter this problem which happens around a learning rate of 0.00005 where we aren't able to observe convergence within the time frame. It is for that reason that the value is not plotted for scale reasons. Similarly, it can be seen that the values greater than 0.00025 are also not shown within the plots this is because again we were unable to see any convergence within the 5000 episode limit. It appears that for these higher values of learning rate as we expected the learning steps are too large in magnitude and cause the agent to not converge to a proper solution and in some cases ended up demonstrating very poor performance later on in the test run. This confirms our assumptions that there is a reasonable medium where we can both learn stably yet quickly enough where we make consistent progression towards our solution. As demonstrated in the figure 5 we can clearly see that we arrived at the value of 0.0005 as our optimal value of learning rate for this particular agent. Although, it is again notable that there is a fair amount of noise in the data given the limited number of runs which ideally would be expanded to better approximate average performance. However, despite these limitations it seems fairly reasonable to select as our best parameter.

X. Discussion & Closing Thoughts

As we can plainly see from our experiments and their related discussion the choice of hyper parameters in DQN/DDQN agents can be critical to having an agent that is capable of solving the problem at all. It is widely known that hyper parameter choices will vary greatly in different model architectures, with respect to one another, and with respect to the learning task. We learned and showed how extremely high or low ranged hyper parameters usually causes learning to fail. However, these experiments should provide some general heuristic ranges of hyper parameter values that are generally feasible and should be a reasonable starting point on a wide variety of other problems. Hopefully given more time future work could've elaborate on these additional parameters to create a proper complete evaluation of the agent under this environment most notable epsilon as well as hyper parameters interactions. Additionally exploration into dueling approaches was one such thing that was implemented, but left out of this report due to length constraints. Collectively, these results demonstrate the great strides that have been made in deep reinforcement learning both in the ability of hyper parameters to improve models as well as fundamental model extensions to improve learning. Both of these approaches are necessary to train effective agents especially as we use AI to tackle more complex problems beyond the simple lunar lander environment.

References

[1] Riedmiller M. "Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method". In: ECML 2005, LNAI 3720, pp. 317-328, 2005. New York, NY: Springer-Verlag Berlin Heidelburg, 2005

[2] Voolodymyr M. Kavukcuoglu K. Silver D. et. al. "Human-level Control Through Deep Reinforcement Learning" In: Nature Vol 518 pp. 529-533, February 26 2015. London, GBR: Macmillian Publishers Limited February 2015.

[3] Van Hasselt H. Guez A. Silver D."Deep Reinforcement Learning with Double Q-Learning" Association for the Advancement of Artificial Intelligence. London, GBR: Google Deep Mind December 2015.

[4] Watkins C. "Q-Learning" Machine Learning, 8, 279-292. Boston: MA. Kluwer Academic Publishers April 1992.

[5] Hasselt, Hado. "Double Q-learning." Advances in neural information processing systems 23 (2010): 2613-2621.

[6] Kratsios, Anastasis, and Ievgen Bilokopytov. "Non-Euclidean Universal Approximation." arXiv preprint arXiv:2006.02341 (2020).