Lunar-Lander Optimal Control

Sandesh Tiwari

Githash- 5742a7ab04cc1a3d5003ee4dbba366592791810c

Abstract— In this paper, we present a few experiments done while training a reinforcement learning agent using a deep Q-network to land the lunar surface of the OpenAI gym's LunarLander-v2 environment. We also discuss how the agent performed using different hyperparameters settings.

1 INTORDUCTION

In this article, we discuss the experiments performed while training a deep Q-network (DQN) to land on the lunar surface of the OpenAI gym's LunarLander-v2 environment (Mnih, V., Kavukcuoglu, K., Silver, D. et al., 2015). The state-space in the LunarLander-v2 environment is represented by an eight-dimensional state vector with six continuous variables and two discrete variables. The first two variables represent the x-axis and y-axis position of the lander respectively, the third and fourth variables represent the horizontal and vertical speed of the lander respectively, the fifth and sixth variables represent the angular speed for the lander, and finally, the last two discrete variables represent binary values to check if the lander's left and right leg are touching the ground. This brings us to the rationale for using a DQN instead of regular Q-Learning (Sutton, Richord, and Barto, Andrew G. 2018). To represent continuous variables for a regular Q-Learning, which uses tables to store state-action pair values, gets infeasible because the state space is continuous, and it is impossible to store everything in a table. On the other hand, a DQN uses a Deep Neural Network (Goodfellow, Ian, et al. 2016) as a function approximator to approximate values/action-values using a given state representation and one only must maintain the weight vector for the neural network instead of maintaining all possible values for the state variables. The state variables are fed to the input layer of the neural network and it performs some computation and gives the output. The output for our experiments is a list of all the possible action's softmax probabilities. For our environment, there are four possible actions: do nothing, fire left-oriented engine, fire right-oriented engine, and fire the main engine.

The goal of a lander is to land on the landing pad located at (0,0). If the lander moves away or moves closed towards the lander it penalized accordingly. If the lander crashes it receives a -100 reward, while if it successfully rests it receives a +100. Also, if the lander's legs meet the surface it receives a +10 reward. Each firing of the main engine receives a -0.3 reward and each firing of the side engines for orientation receives a -0.03 reward. Landing on the landing pad is the goal but landing on either side of the pad is also possible, although the surfaces may not be even like on the landing pad so has a higher chance of unstable landing which may lead to the use of engines to balance or may even lead to a crash. The lander can use an infinite amount of fuel but will be penalized for use of the engines as mentioned earlier. This problem is considered solved if a lander gets an average reward of +200 for 100 episodes.

2 DQN AND NEURAL NETWORK OVERVIEW

Neural networks are used in DQN because they are very good at approximating complex functions. With continuous values for input variables, like in our experiments, it is very difficult to handcraft a function that will correctly approximate. The main goal of neural networks is to minimize a cost function. A cost function is a measure by which we can tell how different the prediction and the actual values are. The predictions are generally known beforehand and are usually static. The global/local minimum of a cost function is computed using an algorithm called Stochastic Gradient Descent (SGD) or it's variants (Goodfellow, Ian, et al. 2016). A neural network can consist of several layers, which use non-linear activation functions to pass data to the next layer. To find the minimum point in the cost function, the weights of each layer is adjusted using, the partial derivatives of the cost function with respect to the weights, an algorithm called backpropagation (Goodfellow, Ian, et al. 2016).

There can be several hyperparameters for a neural network. For our purpose, we have used a regular feed-forward neural network architecture with two hidden layers and an output layer. Each hidden layer consists of 256 neurons. The first layer is an eight-element vector representing the eight variables from the state representation. The output layer consists of four neurons which represent the probability distribution over all four actions. A learning rate for a neural network can be thought of as a step size that is taken in each weight update step. We go into more detail about different learning rates in a future section.

For a reinforcement learning problem, the prediction is time-delayed, so we use a *replay memory* to store samples from an agent's experience in the form $e_t = (s_t, a_t, r_t, s_{t+1})$ for each experience and use the data from the memory to make learning stable. Sampling from a memory makes the training data very un-correlated and aids learning (Mnih, V., Kavukcuoglu, K., Silver, D. et al. 2015). In our experiments, we use a memory size of 100000. For each update, we use a batch size of 32, i.e. we update weights using 32 batches simultaneously. Using a batch size accelerates learning this number can be increased but using a too large batch size might slower the learning speed. For training, a sample is drawn uniformly from the memory. DQN also uses two neural networks, one to generate target and one to generate predictions. This further stabilized training

because we update the target network after every C step. We used the Mean-Squared error as our loss function. For our optimizer, we use Adam, which is considered a better variant of SGD (Kingma, P. D, & Ba, L. J., 2017). Figure 1 shows the expected loss function for DQN (Mnih, V., Kavukcuoglu, K., Silver, D. et al. 2015).

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)}\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

Figure 1- Loss function of a DQN.

In Figure 1, r is the reward for the step, γ is the discount factor, Q is the function, θs are the weights for functions, and U(D) is the uniform distribution for replay memory D.

3 TRAINING THE AGENT

Figure 2 shows the results for our first experiment. The x-axis represents the episodes and the y-axis represents the reward for each episode.

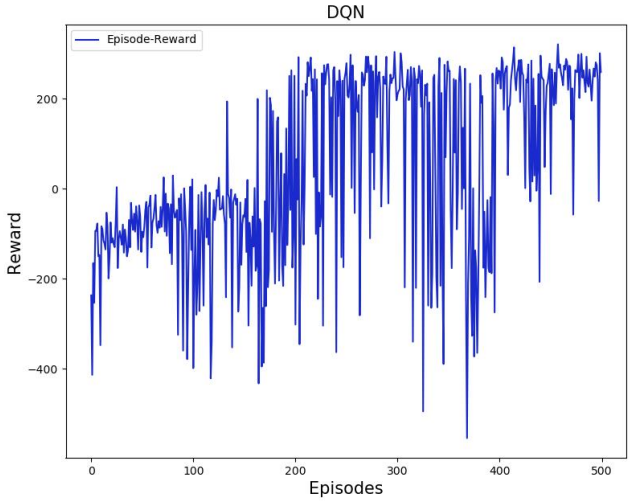

Figure 2- Graph of episodes vs. reward for the training process

For this experiment, we used different hyperparameters by an informal search and testing. More detailed analyses are done in the following sections. But for Figure 1 we set the learning rate for the neural network α to 0.001 and used two hidden layers with 256 units each. For the reinforcement learning hyperparameters of the experiment we set the discount factor γ to 0.99, initial exploration ε to 1.0, ε-decay to 0.00005, minimum ε to 0.01, the replay memory size to 100000, and the network update interval C, i.e. the interval after which the target network in updated, to 100. A batch size of 32 is used to train the neural network. Also, we trained the agent for 500 episodes because it seems to converge at around this range.

As we can see that the agent starts very bad as it is taking random action as ε is very high. As more episode's elapsed, the agent acts more greedily. The learning rate,0.001, for the neural network takes small steps while trying to minimize the loss function from Figure 1. We can also observe that even after 300 episodes some episodes have very low rewards, this is because the agent has a minimum ε value of 0.01 which ensures that the agent selects random actions at least 1% of the time. This also causes the graph to be noisy. The network update interval variable, C is updated after 100 timesteps in an episode. The number of episodes was choose experimenting a few times and picking a number when the last 100 episodes average is over 200. The number of hidden layers is set only to two for the neural networks because more hidden layers might introduce a very common problem of vanishing gradients (Hanin, B. 2018). One hidden layer also worked well but two hidden layer networks were generalizing better. Also, a neural network with only a few units in the hidden layers also might not learn the function so we used 256 units for the layers.

Even with this noisy graph, we can see the trend in the graph and that rewards received in the episodes increase. This shows that the agent is initially exploring more than exploiting and as it explores less it is exploiting the learned values which become better than exploring in later episodes.

4 ALPHA (LEARNING RATE) EXPERIMENT

For our next set of experiments, we tested three different values of α, the learning rate for the neural network. As mentioned earlier, the learning rate is the step size in the cost function. A good learning rate finds local/global minimum but a larger α might overshoot the minimum and might bounce around the minimum. Contrary, if we select a smaller α, we might only take small steps and hence will take longer to find the minimum. Figure 3 shows a graph comparing three learning rates. The x-axis represents the episodes and the y-axis represents the average reward for the last 100 episodes.

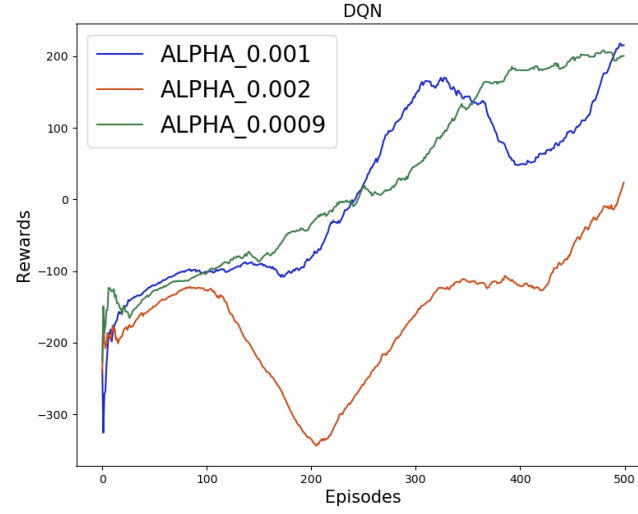

Figure 3- Moving averages for the last 100 episodes using different α values.

For convenience we set all the other parameters fixed to compare the varying α values. For all the learning rates we can see that because of the exploration factor ε, the agents do not differ much in the early episodes, but their differences can be observed more clearly in the later episodes. For α 0.002, we can see that the agent does not perform well even after 500 episodes. This is because the learning rate is too high and the update is probably jumping around a local minimum in the cost function and cannot get to the local minimum. The moving average rewards mostly are

below zero and the agent does not perform well. For α 0.001 the agent does learn better than in the previous case. The moving average increases slowly towards 200, which is considered a good score. The dip around episode 400 is most likely due to the exploration hyperparameter ε because it is never less than 0.01 and there is always a 1% chance the agent will explore. For α 0.0009, after a few initial episodes of higher exploration, the moving average increases without major dips along the way. This is because the learning rate is so low that the weight updates are very small, and they are not jumping around the local/global minimum but is moving towards the minimum instead. With this learning rate, the agent will improve if trained for even more episodes because of the update size, but it might not be the best choice because with α 0.001 we see that we reach the 200-reward average in fewer episodes.

5 EPSILON DECAY EXPERIMENT

Epsilon decay is a very important hyperparameter as it determines how long an agent explores new actions for an input state vector and when an agent starts exploiting more. For our experiments, we reduce the ε-decay value from the ε decay parameter for each episode until the ε is set to 0.01. Figure 4 shows a graph for three agents with three ε-decay values.

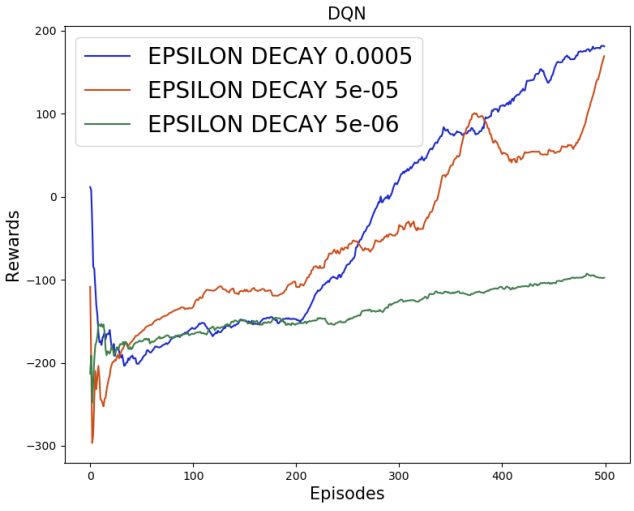

Figure 4 – A comparison of different ε-decay hyperparameters.

From the figure, we can see that exploitation is also important after high exploration in the initial episodes. With the decay set to 5e-06, we can see that the curve does not rise so quickly. This is because the agent is exploring most of the time and does not approximate the action probability distribution from the updated weights. For an ε-decay set to 5e-05, we see that the agent does much better. This is because the agent exploits it's learned weights sooner. Finally, with ε-decay set to 0.0005, the agent performs even better. The curve rises quicker than the previous two values. Also, we can see that for the first 100 episodes, all agents earn similar rewards. Then the 0.0005 ε-decay agent starts performing well and after about 200 episodes, the 5e-05 ε-decay agent also starts performing better. It shows that random exploration is not always helpful, and the agent needs to exploit after some exploration. The agent with ε-decay set to 0.0005 was able to achieve the average reward of 200 over the last few episodes.

6 GAMMA/DISCOUNT EXPERIMENT

The discount factor γ is also an important hyperparameter. As we can see in Figure 1 that γ multiples with the greedy value from the next state-action pair. γ is a discounting factor, so it is very important to set it below 1 because we do not want to give future state values more weight than the current reward. Figure 5 shows the results from our experiment with three different γ values.

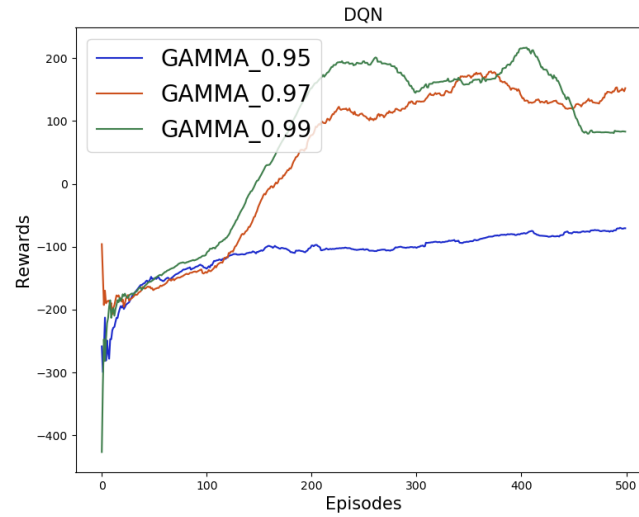

Figure 5- Comparison of different values for the discounting factor of the future value.

From the figure, we can see that when γ is 0.95 the agent does not perform well. This is because the agent is not weighing the next state-action enough so the loss function is not able to represent a good function and so the weight updates are also not very good. For γ set to 0.97, the agent does much better than the previous setting. Even just a slight increase in the weight of the next state approximation made the loss function a much better representation of the actual loss. The curve rises very quickly after about 100 episodes. Similarly, when γ is set to 0.99 the agent again performs very well. This is also because the loss function becomes a better loss function. For all three cases, we can see that agents perform similarly until the first 100 episodes. This is because the weights of the function approximator, neural network, are not updated enough to correctly find an approximate the loss measure. This is a result of the random initialization of the weights for the neural network. Once enough updates are made, to the network, using a good discount value the agent learns better approximate of the loss function.

From this comparison, we can imply that a higher discount factor-like 0.99 is a good weight for the future state-value estimate. The agent needs to be farsighted to perform better and to make the loss function closer to the actual loss function.

5 TRAINED MODEL ANALYSIS

Now we will turn our attention to a trained agent for analyses. For this agent we have selected the following hyperparameters: α is 0.001, γ is 0.99, ε-decay is 0.0005, and the rest of the hyperparameters like the number of layers, the number of units, memory size, and batch size are also the same. The α, γ, and ε-decay values were selected based on all the previous experiments presented in this paper. Figure 6 presents a graph with a reward for each of the 100 episodes the agent performed.

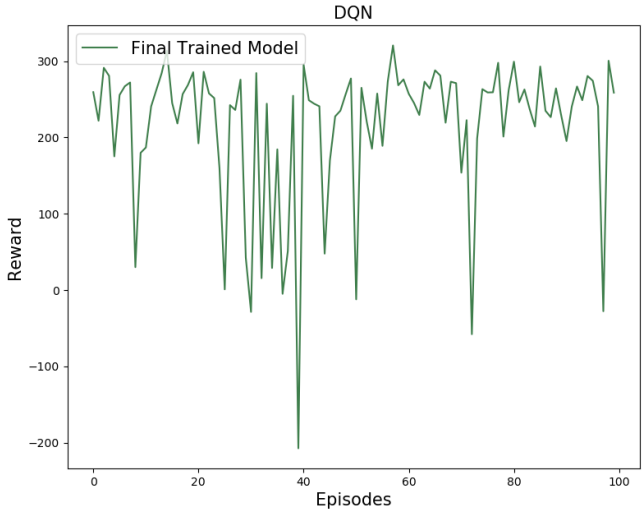

Figure 6 – Rewards per episode for a trained model.

As we can see in the figure that the trained agent performed very well with an average of 204.32 for 100 episodes. But even the trained agent performed badly in a few of the episodes. This is because our DQN is a function approximator and to correctly approximate every state representation to the correct action is very difficult, so some approximation leads the agent to take wrong actions which instead resulted in negative rewards. On the contrary, the function approximator performed very well for some episodes and it even led to some episodes with scores around 300. The agent could perform better if trained for more episodes because the neural network will become a better function approximator with more experience. But for our purpose, an average score of 200 over 100 episodes is considered a good agent.

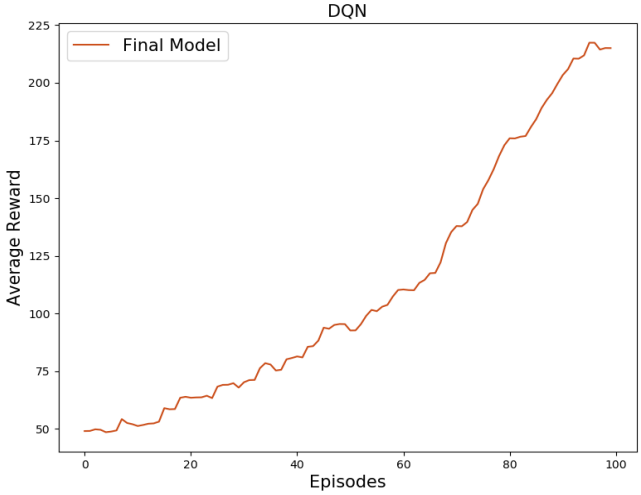

Figure 7 – Average rewards for a trained agent

To further illustrate that performance of the trained model we can see the average reward increasing in Figure 7. As we can see in the figure, the average reward for the episodes increased over the episodes and by 100 episodes the average is well over 200. The graph is not smooth because the agent did not perform well for some episodes. The graph demonstrates that our agent was able to solve the lunar landing task.

6 THE TRAINING ALGORITHM

In this section, we will go briefly over the algorithm we used to generate the graphs in the previous sections.

Algorithm: Lunar-Lander DQN with experience replay.
Initialize environment e, and rewards array
Initialize agent with two neural networks
For episode 1 to M **do**
    Reset e and observe state s_t, set termination t to False, and episode reward to zero
    While t is False **do**
        With probability ε select random a_t else select action $a_t = argmax_a\, Q\,(s_t, a, \theta)$
        Execute a_t and observer r, s_{t+1}, and t
        Store tuple $(s_t, a_t, r, s_{t+1}, t)$ to memory
        Make the agent learn from memory
    End While
End For

The final step in **While** block in the above algorithm is done inside the agent and is as follows.

Algorithm for the agent to learn:
If memory less than batch size b then return
Initialize counter C
Sample random minibatch b transition (s_t, a_t, r_t, s_{t+1}) uniformly
If episode terminates $y_t = r_t$
Else $y_t = r_t + \gamma\, max_{a'}\, Q'\,(s_{t+1}, a', \theta')$
Perform gradient descent on $(y_t - Q\,(s_{t+1}, a, \theta))^2$ with respect to θ
Every C step reset $Q' = Q$ and set C to zero
End

For the algorithms presented above, Q and Q′ are the predicting and target networks respectively. Also note that when the agent is initialized, the two neural networks are also initialized along with the batch size and other hyperparameters. Additionally, from the first step above we can see that if the current memory is less than the batch size we do not learn, but only after the memory has enough transitions stored. Furthermore, setting the memory size to 100000 helped a lot because the agent was sampling from a bigger domain, so the samples were less correlated. Initially, we tried using a smaller memory size of 500 but this did not yield better performance. This was because the agent needs to sample from more experiences to get more uncorrelated sequences of transitions.

The above-mentioned algorithm is very similar to the algorithm presented in the original DeepMind paper (Mnih, V., Kavukcuoglu, K., Silver, D. et al., 2015), but one major difference is that we are using a feedforward neural network instead of a Convolutional Neural Network (CNN). This is because we are not dealing with images and the

environment returns continuous values instead of screenshots from the environment.

7 PITFALLS AND PROBLEMS ENCOUNTERED

There were a few problems we encountered while building the landing agent. First was that when we trained the agent with only 200 episodes, the agent did not learn to land but only to hover just above the surface. This was because it was trying to prevent a crash, which had yielded high negative rewards from experience. Also, up to 200 episodes the agent does not have many experiences with successful landing, so the weights of the networks are not tuned properly to have smooth landings. Second, tuning the right number of layers and units for the neural network was a time-consuming and arduous. Initially, we used one hidden layer with only 32 units which were not able to learn the function for our lander. We then added another hidden layer with the same number of units and got some better results, but still, the agent wasn't performing so well. This indicated that the function we are trying to build is very complex, so we increased the number of hidden units to 256 each and got our final agent. All these changes helped the lander to learn the function better. Also, initially, we tried using a single neural network but the agent did not learn very well and the average reward for 500 episodes was not good, so we added a target network to stabilize training so that the target is only moving after every C steps as mentioned in the algorithm presented above.

8 FURTHER IMPROVEMENTS

The final agent performed well and was able to get an average reward of over 200 in 100 episodes but there can be some improvements that can help get the agent to perform even better. One of the main improvements we would make is to reshape the reward based on the distance to the landing pad. Although the environment may have already implemented a type of reward shaping, we would use potential-based reward shaping to see if it helps the performance (Ng, Y., A., Harada, D., & Russel, S., 1999). Another experiment could be done by adding more hidden layers to the neural network and tuning other hyperparameters accordingly. This would try to learn more complex functions for the agent. If given more time, we would also experiment with the minimum ε value to allow the agent to exploit more in the later episodes. Finally, in the future, one can experiment with the batch size by increasing it and increasing the episodes accordingly. These changes are significant enough to modify the performance of an agent so experimenting with them might yield more results for analyses.

9 CONCLUSION

The agent for the LunarLander-v2 environment required significant hyperparameter tuning to perform well. The experiments and figures presented above can be extended to tune other hyperparameters and find even better results. But from our experiments, we can imply that a good learning rate is needed for the neural network to find a local minimum. The agent also needs to weigh future values more to create a good loss function. Also, the agent needs to exploit after exploring for initial episodes and a good balance is required for the agent to perform better. Finally, even the final agent does not perform well in some of the episodes. This is because our problem is a continuous variable problem where the state values are continuous, and it is very difficult to approximate a function that performs well on all possible states. Suggestions mentioned in the above section would be a good starting place to further improve an agent.

REFERENCES

[1] Goodfellow, Ian, et al. *Deep Learning*. 2016.

[2] Hanin, B (2018). Which Neural Net Architectures Give Rise to Exploding and Vanishing Gradients? http://papers.nips.cc/paper/7339-which-neural-net-architectures-give-rise-to-exploding-and-vanishing-gradients.pdf

[3] Kingma, P. D, & Ba, L. J. (2017). Adam: A Method for Stochastic Optimization. https://arxiv.org/pdf/1412.6980.pdf

[4] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015). https://doi.org/10.1038/nature14236

[5] Ng, Y., A., Harada, D., & Russel, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/NgHaradaRussell-shaping-ICML1999.pdf

[6] Sutton, Richord, and Barto, Andrew G. *Reinforcement Learning An Introduction second edition*. 2018.