Function Approximation – Implementation of DQN on the Lunar Lander Problem

CS 7642 – Reinforcement Learning and Decision Making

Hash: 29856e646e1cf9e24794d62b8ee6b9dc0dfe8665

***Abstract*—This is the report for the second project of the reinforcement learning and decision-making class. Here, I implement Deep Q-Network on the Lunar Lander Problem.**

I. Introduction

Value iteration and policy iteration are used for policy prediction when the transition matrix and the rewards are known. Model-free methods such as Q-Learning and Sarsa are good for policy control when the space is discretized. However, consider playing backgammon which has 10^{20} states, Computer GO which has 10^{170} states or consider a continuous state, then these methods are not practical. For example, Q-value can potentially have an entry for each state and action. The amount of experience and computer power required to solve backgammon through Q-learning is immense [1,2].

Function approximation replaces these methods by using a value function or an action-value Q function as a function of not only of state, or state-actions pairs, but by also using weight parameters. These new functions are inspired by the previously studied methods such as Monte-Carlo simulations, Temporal-Difference (TD) or Q-Learning. However, they convert these methods from tabulation to functions.

The input-output system for function-approximation could be one of the following: State vector is entered as input, value function is outputted. State-action pair matrix is flattened and entered as input, q function is outputted as a scalar. State is entered as input, q function with the length of action space is outputted. Weights are used in the hidden layers for tuning.

Some of the common function approximators can be linear regression, polynomial regression, neural networks, decision trees, k-nearest neighbor methods or Fourier bases. In this paper, I do neural network implementation.

Deep learning implementations for reinforcement learning use models such as Sarsa or Q-learning as target function and they use NN's to mimic reinforcement learning algorithms. The advantage is that the input (state space) can be continuous, which eliminates the tabulation process.

Deep learning methods for reinforcement learning problems can be tricky because of several reasons. One big issue is convergence as convergence may not be guaranteed particularly for non-linear function approximators such neural networks. Another problem is that deep learning requires a lot of data that are labelled and independent from each other. Reinforcement learning must come up with its own guess of the labels. Therefore, the labels are not ready before training and the labels are continuously updated during training as well. Further, data are actually not independent for most of the reinforcement learning cases. Lastly, the data distribution changes for reinforcement learning throughout the training, and hence exploration-exploitation needs to be balanced.

Sarsa and Q-learning are not guaranteed to converge for non-linear approximation. Sarsa is guaranteed to converge for linear approximation functions while Q-learning is not. Gradient Q-learning, on the other hand, is guaranteed to converge for linear approximation functions. Similarly, Gradient TD has better convergence behavior than TD [1,2].

Gradient descent step for a neural net perceptron is:

$$\Delta \boldsymbol{w} = -\frac{1}{2} \propto \nabla_{\boldsymbol{w}}(J) \tag{1}$$

where $\boldsymbol{w}$ [1] is a vector of weights, $\propto$ is the learning rate and J is the objective function to be minimized. $\nabla_{\boldsymbol{w}}(J)$ is the partial derivates of J with respect to $\boldsymbol{w}$, and it directs the weights towards a smaller error. Here J is

$$J = \mathrm{E}\left[\left(Q_\pi(\boldsymbol{s},\boldsymbol{a}) - \hat{Q}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{w})\right)^2\right] \tag{2}$$

where Q_π is the true value-action function and $\hat{Q}$ is the approximation function. Q function could be replaced with the v function as well if we have the model i.e. transition matrix and rewards. Once the function approximation is trained, Bellman's equation can be used to find the optimum policy. However, here I only show the Q function implementation. Combining equations 1 and 2 gives

$$\Delta \boldsymbol{w} = -\frac{1}{2} \propto \nabla_{\boldsymbol{w}} \left(\mathrm{E}\left[\left(Q_\pi(\boldsymbol{s},\boldsymbol{a}) - \hat{Q}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{w})\right)^2\right]\right).$$

Now, I take the partial derivative of J with respect to $\boldsymbol{w}$. Note that the true (target) function is independent of the weights.

$$\Delta \boldsymbol{w} = \propto J \nabla_{\boldsymbol{w}} \hat{Q}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{w}) . \tag{3}$$

For a single step, the expectation term disappears:

$$\Delta \boldsymbol{w} = \propto (Q_\pi(\boldsymbol{s},\boldsymbol{a}) - \hat{Q}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{w})) \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{w}) . \tag{4}$$

The true policy (target) function can be estimated as one of the following. It can be the expected cumulative reward calculated through Monte Carlo simulations. It can be Temporal Difference learning based estimation. For example, TD(0) estimates the target function as $Q_\pi(\boldsymbol{s},\boldsymbol{a}) = r_{t+1} + \gamma max_a \hat{Q}(\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1},\boldsymbol{w})$. This can also be extended to TD(λ). Even though the target function is expressed in terms of the approximation function as well, its derivative with respect to

[1] Throughout this paper, vectors are expressed in bold.

weights is assumed to be zero because we want the input $\hat{Q}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{w})$ to approximate the target. We do not want target and input to meet in the middle of the way. Target estimate is more like the future estimate that we want to reach.

The approximation function expression could be many things. For example, function approximation can be linear.

$$\hat{v}(\boldsymbol{s}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x} \tag{5}$$

where $\boldsymbol{x}$ is the state vector. Alternatively, a non-linear neural network with many layers and neurons can be used for function approximation.

Deep Q-Networks (DQN) implement neural networks to approximate the *Q* function [3,4]. DQN's use experience replay which means that previous experiences are stored in $\boldsymbol{D}$. Every row of $\boldsymbol{D}$ is an experience $\boldsymbol{e}$ that contains [$\boldsymbol{s}$,a,$\boldsymbol{s}'$,r]. As the training takes place, random batches of $\boldsymbol{e}$ are drawn from $\boldsymbol{D}$. Further, DQN's have fixed targets which means their derivatives are not taken. Experience replay and having fixed targets stabilize the training process. They use the following as the objection function.

$$J = \mathrm{E}[(r_{t+1} + \gamma max_a \hat{Q}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}, \boldsymbol{w}^-) - \hat{Q}(\boldsymbol{s}_t, a_t, \boldsymbol{w}))^2] \tag{6}$$

The neural network takes the state vector as input and outputs an approximation function for each action. Therefore, the output has the size of the action space. $r_{t+1} + \gamma max_a \hat{Q}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}, \boldsymbol{w}^-)$ is the target value for that experience. The "-" superscript above $\boldsymbol{w}$ mean that target weights are fixed. Remember the experience was randomly selected from $\boldsymbol{D}$. Even though same neural network can be used to estimate the target and the input values, it is also possible to update the target function slower than then input function.

II. LUNAR LANDER

Lunar Lander is a deterministic problem in OpenAI Gym. The steps are deterministic. The purpose of the game is to land the rocket smoothly. There are 8 states, 6 of which are continuous. Continuous states are x location, y location, x velocity, y velocity, rocket angle, rocket angle velocity. There are two binary states: x_{leg} and y_{leg}, which shows whether the left or the right side of the rocket touches the ground [7].

There are four possible actions: do nothing, fire the left engine, fire the main engine, fire the right engine. Every step has a small negative reward. Crash has a large negative reward and successful landing has large positive reward. Next, we implement DQN to train the rocket for successful landing.

III. IMPLEMENTATION

I created a Pytorch NN where the states are the input. There are several dense layers of neurons and an output of size 4, one *q* for each action. Each hidden layer a linear activation function followed by a relu function. Number of layers and neurons are tuned along with several other hyperparameters.

The following procedure is implemented for training. Seed is fixed for numpy, gym, torch and random api's. An empty $\boldsymbol{D}$ is created. A Pytorch based NN is initialized. Lunar Lander gym environment is initialized. The following pseudocode is implemented.

```
Start episode loop for N episodes:
   Reset the gym environment.
   Total rewards -> 0
   Start step loop until episode is completed:
      Make a step based on epsilon-greedy search.
      Store s, a, s', r in D.
      Randomly pick 64 experiences from D.
      Cycle if step % 4 != 0
      Cycle if experience is less than 64.
      Set the optimizer to Adam.
      Set the loss function to mse loss.
      Update the weights based on gradient.
      s <- s'
```

Epsilon-greedy search takes a random step if a randomly generated number between 0 and 1 is below ε. ε is initially very high, close to 1 as we want more exploration at the beginning. Later ε is updated as $\varepsilon = \max(\varepsilon_{min}, \varepsilon * \varepsilon_{decay})$ for each step. ε is initially 1, ε_{min} is a small value between 0.05 and 0.1, ε_{decay} is between 0.995 and 0.999. Taking a random step allows for exploration and it is called off-policy as it is not moving according to the policy. Full-greedy search would be on-policy.

We do not want to train the algorithm greedily every step. This slows down the training. Therefore, we collect some experiences and retrain every 4 steps. I set the optimizer to Adam as it is a very popular optimizer. The loss function is mean squared error as shown in Eq. 6. Target function reduces to r_{t+1} if the state is terminal. Pytorch is used to set up and train the DQN.

IV. PROBLEMS FACED

Lack of knowledge of Pytorch was a major issue. Pytorch requires inputs in torch tensor format and it outputs in torch tensor format whereas OpenAI Gym inputs and outputs in Numpy format. The datatypes need to be converted all the time and they need to be put in correct shape. Pytorch backward step requires the gradient information for the input values. These values were lost before and it needed a fix. I also took help from [5,6] to understand Pytorch, OpenAI and the Lunar Lander problem.

Another problem that I faced was that I used np.random.random(0) to generate the epsilon value for greedy-search. This function was returning an empty array. The training was running without giving an error. However, it was never taking a random step. Therefore, no exploration was happening. It took a while to notice this mistake.

Another important problem that I had was to see slight improvements from -200 rewards to close to zero values, but then diverge back to -200 or even worse values. I have noticed my training was not very successful even for more of the successful cases. For example, I have seen cases where the 100 rolling mean reaches above 200 and then diverge back to negative. I think the instability that I had was mostly due to not using a fixed target. Later, I have found out that the neural network used for target should be lagging the neural network used for the input [1]. For future implementation, I would fix target NN and then copy the input NN into target NN every few steps. This may bring more stability to training.

Weight tuning was very critical. There are many hyperparameters to tune. For example, I noticed larger batch size brings more stability, but it also slows the run. Initially, I was setting γ equal to 0.9 which was too low. If there are 1000 steps till termination, the final reward is multiplied with 0.9^{1000}. Later, I fixed the γ value to 0.99 which has improved the results a lot. I also used a hyperparameter tuning tool called Optuna. I stopped Optuna and reentered the tuning parameter ranges several times when I noticed that a particular range for a particular parameter is more successful. For example, when I noticed I get better results with fewer number of layers, larger number of neurons and larger size of experience replay, I changed the ranges for the tuned hyperparameters. The final ranges I put were, 1-3 layers, 50-150 neurons, 10^{-4}-10^{-3} learning rate, 0.05-0.1 ε_{min}, 0.995-0.999 ε decay and 100000-1000000 experience replay vector size. I also noticed it prefers small ε_{min} , probably because further exploration results in bad moves closer to the end of the training.

Further, I carried out vectorization for the loops in order to speed up the calculations. The vectorization caused loss of information in datatypes which had to be fixed.

V. RESULTS

The final model has the following hyperparameters: input layer with 8 neurons, 1 hidden layer with 199 neurons, output layer with 4 neurons. The hidden layer has linear activation function followed by relu. Table 1 shows the other hyperparameters.

Table.1 Summary of Final Hyperparameters

Learning rate	0.0001856
ε_{min}	0.07
ε_{decay}	0.995
Len($\boldsymbol{D}$)	197144
Batch Size	64
Initial ε	1.
γ	0.99

Figure 1 shows the training total rewards per episode (in blue) and its 100 rolling mean (in orange). Initially, the rewards are close to -200. After several 100s of episodes, total rewards reach 200-250. After 1000 episodes, total rewards starts to diverge probably because I did not use fixed target network. For testing, I saved and loaded the final model. I simulated the Lunar Lander problem for 100 episodes without training and by taking full-greedy steps. Figure 2 shows the testing total rewards per episode and its rolling mean. The average of the testing rewards is 235 for 100 episodes.

Figure 3 shows training performance for varying γ's. γ is a value between 0 and 1. It is used to punish future rewards because a reward received after 1000000 steps is not very valuable. The lower the γ, the more punishment on futures rewards. I picked γ as the first hyperparameter to discuss because initially I picked 0.9 as γ and the model was not converging. Figure 3 shows that training rewards converge to -100-0. However, only 0.98 value converge to above 200. This is because a low γ punishes the final reward extremely making the reward negligible. Therefore, the agent was not able to learn how to collect the best rewards. I eventually picked 0.99 for γ. A γ of one treats all the rewards the same which caused the training rewards diverge to -800. Table 2 shows that computational time for γ = 1 training is very small. This is worth investigating why $\gamma = 1$ is so problematic.

Figure 4 shows the testing performance for different γ values. Figure 4 mimics Figure 3 in that γ=1 gives very bad results and only $\gamma = 0.98$ gives good results. Realizing this was very important for me which led me to pick high γ value.

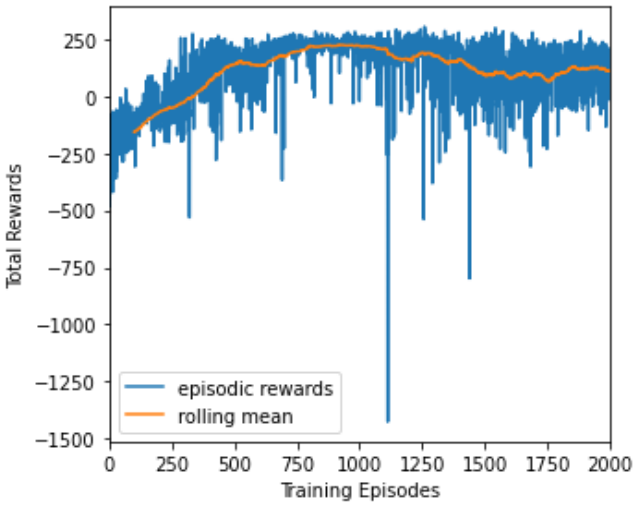

Figure 1. Total rewards per training episode for the final model.

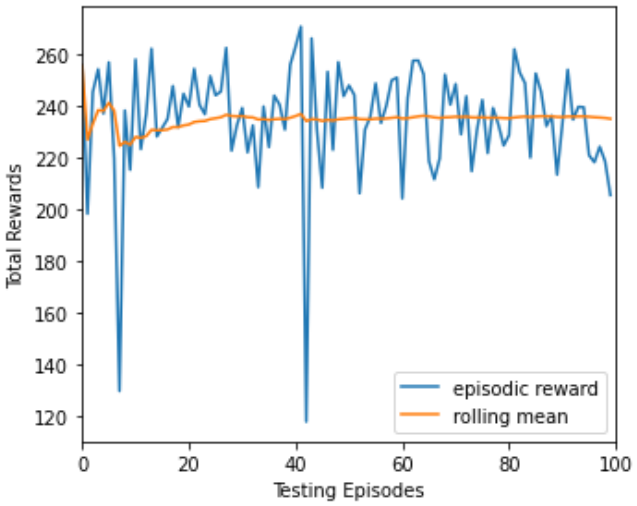

Figure 2. Total rewards per testing episode for the final model.

During Optuna tuning, I noticed that 1 layer gives the best training results. Therefore, I wanted to vary the number of layers to see their effects on training. Figure 5 and 6 show training and testing total rewards, respectively. 1 layer gives the best performance both for training and testing. I don't think larger number of layers underperform because of overfitting as they are not able to perform well even in training. The reason they are not able to fit may have to do with the speed of backpropagation of derivative information. If backpropagation is too slow, it may cause late learning in the earlier layers. It is also possible that the derivatives will be compounded in the early layers which may cause the agent to diverge. Interestingly, layer 2 performs ok for training but performs very bad for testing which is a sign of overfitting. Table 2 shows that 3-layers NN trains fastest which is the

opposite of what I was expecting as more layers require more forward and backward calculations.

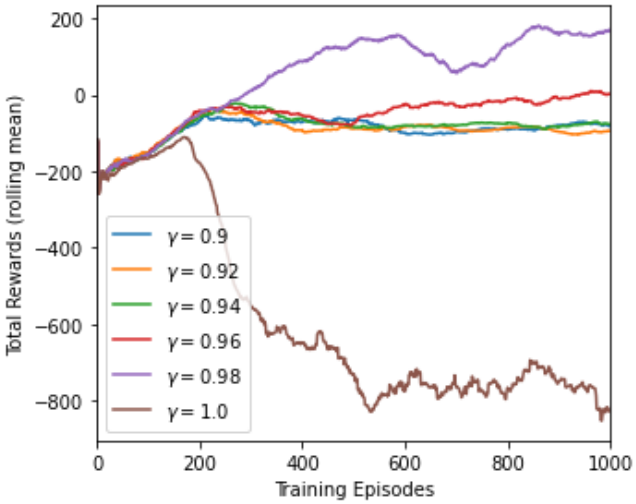

Figure 3. Effect of γ on training performance.

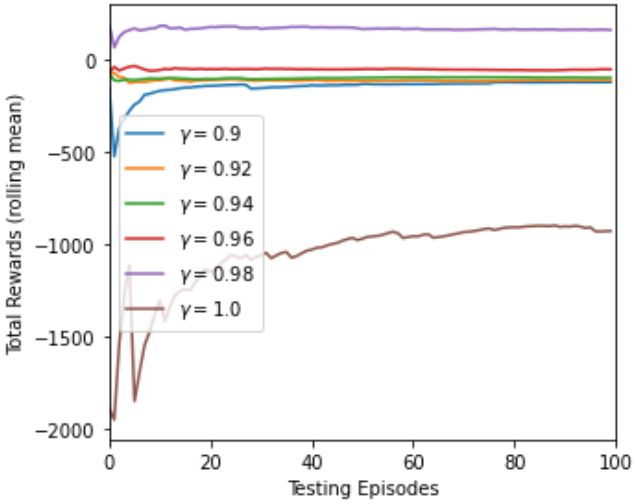

Figure 4. Testing episodes vs. total rewards for varying γ.

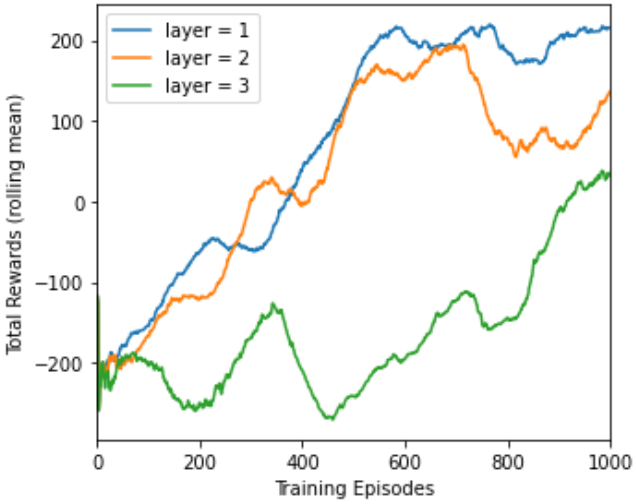

Figure 5. Training episodes vs total rewards for varying NN layers.

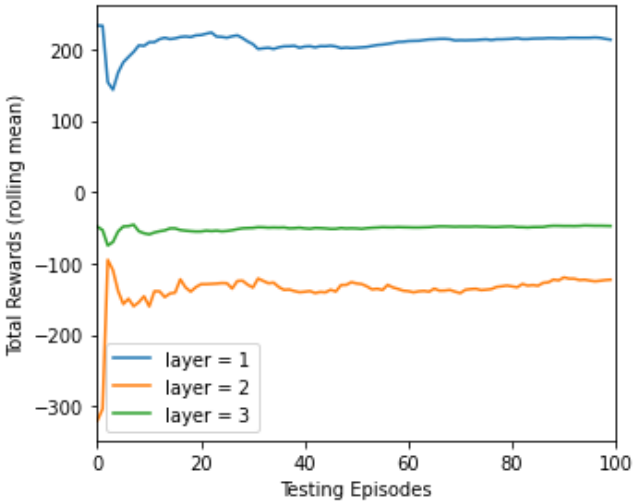

Figure 6. Testing episodes vs total rewards for varying NN layers.

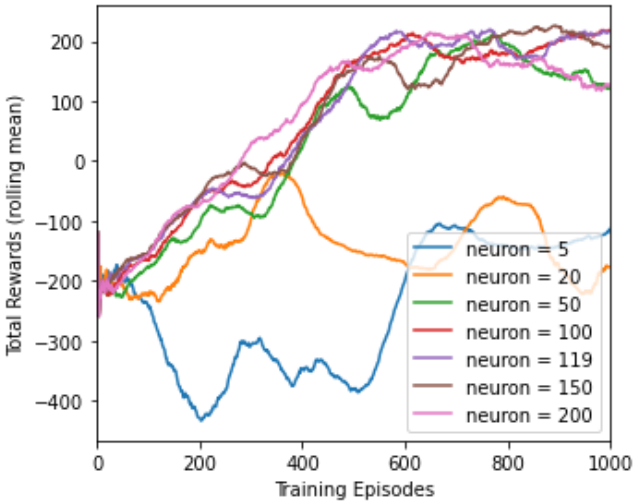

Figure 7. Training episodes vs total rewards for varying NN layers.

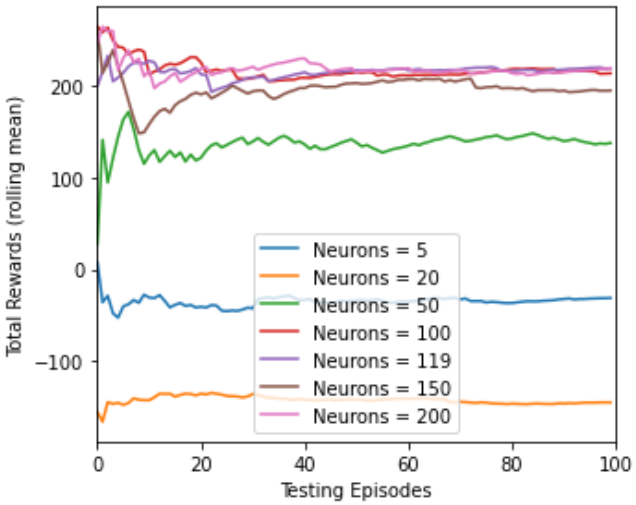

Figure 8. Testing episodes vs total rewards for varying NN layers.

Figures 7 and 8 show the training and testing performance for varying neuron numbers per hidden layer. Here, hidden layer number is fixed to 1 but the number of neurons is varied. During training, I had noticed that the algorithm prefers larger number of neurons. Therefore, I increased the number of

neurons every time. Figure 7 shows that 100-120 neurons are the sweet spot for best performance. My optimum model uses 119 neurons. 5, 20 and 50 neuron cases underfit i.e. they are not complex enough to learn the game. Similar behavior can be seen in Figure 8 as well.

Figures 9 and 10 show the training and testing performance for varying ε_{min} values. ε_{min} determine the lowest probability of random step during epsilon-greedy step. A large ε value causes more observation which avoids local optima. However, large ε_{min} will also cause too many random steps which will lead to making bad steps. Training rewards are very low for high epsilon values. The optimum values seem to be 0.01 and 0.05. However, testing shows that 0.07 value gives the best results. This is because I took full-greedy steps during testing and random steps did not cause a crash. Testing results show that only 0.07 gives good results which shows how difficult it was to find the right set of hyperparameter values.

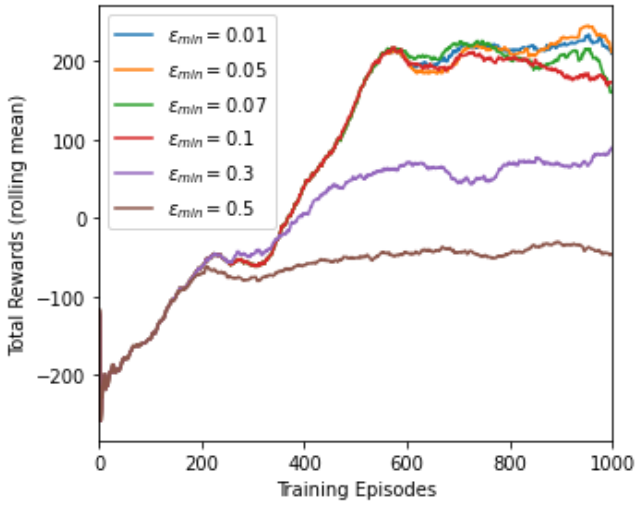

Figure 9. Training episodes vs total rewards for varying NN layers.

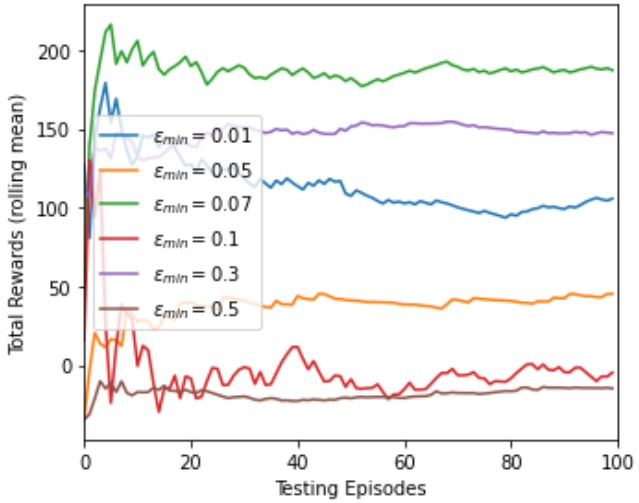

Figure 10. Testing episodes vs total rewards for varying NN layers.

Table 2. Hyperparameter values vs. Computational time.

GAMMA	TIME(S)	LAYERS	TIME(S)
0.9	1989.496	**1**	1100.554
0.92	1950.415	**2**	1026.57
0.94	2005.693	**3**	773.8944
0.96	2099.064		
0.98	1211.233		
1	177.6406		
NEURONS	**TIME(S)**	**EPSILON**	**(TIME(S)**
5	88.41862	**0.9**	948.7562
20	1121.122	**0.92**	934.7623
50	978.0838	**0.94**	952.4469
100	986.9365	**0.96**	1017.672
119	1046.324	**0.98**	1066.415
150	1236.338	**1**	471.4309
200	1158.082		

VI. RESULTS

In this paper, I went through function approximation and implemented DQN on the Lunar Lander problem. During my implementation, I have discovered that hyperparameter is of paramount importance to achieve good results for the following two reasons. First, Q-learning is not guaranteed convergence for reinforcement problems. Second, I did not implement DQN right because I did not fix the target neural network.

DQN introduces two new concepts to stabilize the NN training for reinforcement learning purposes. One is experience replay where we store the previous experiences and randomly pick previous experiences as target. This pulls the algorithm back to random experiences in case it starts to diverge. DQN also introduces fixed target neural network where target NN lags the input NN. Target NN should be updated every few steps.

While I used DQN as it is a popular method and it brings stability, it would be interesting to see how it performs compared linear approximation or other approximations. It would also be interesting to change the calculation of target function from TD(0) to other estimations.

REFERENCES

[1] https://www.youtube.com/watch?v=UoPei5o4fps&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&index=6, RL Course by David Silver

[2] Sutton, R. S. and Barto, A. G. 2020. Reinforcement Learning. Second Edition. MIT Press.

[3] Mnih, V., Kavukcuoglu, K. et. al. Playing Atari with Deep Reinforcement Learning. Deepmind Technologies, 2013.

[4] Mnih, V., Kavukcuoglu, K. et. Al. Human-Level Control Through Reinforcement Learning. Nature vol. 518, pp. 529-533, 2015.

[5] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html , Reinforcement Learning (DQN) Tutorial, Pytorch.

[6] https://shiva-verma.medium.com/solving-lunar-lander-openaigym-reinforcement-learning-785675066197

[7] https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py