Solving Lunar Lander with Function Approximation

Surya Teja Adluri

Hash Code – 3eb666cd473b9fe60fcc0f8d88f56afa9f6a0d1d

1. Introduction

This report analyzes the Reinforcement learning agent's performance that successfully lands the lunar lander in the Open AI Gym. Lunar Environment has continuous state space with an 8-dimensional tuple representing each state space, and it has a discrete action space of 4 possible actions. The environment is implemented in the Open AI Gym. In this project, agents learn from the rewards and only make actions in the environment. Hence, it is a model-free control problem. The agent's end goal is to land at the center (0,0), and the agent receives a reward for various actions from the environment. To solve this problem, tabular methods that involve TD learning methods are not efficient as the environment has a continuous state space and to represent and perform operations on a large continuous state space requires immense computational capabilities. Hence, a function that approximates the state space's value function is needed; this task can be accomplished by both Linear methods like supervised methods and non-linear methods like Artificial Neural Networks. This approximate value function is represented by a set of weights typically less than the possible states. Once a value function is generalized, different methods like Sarsa, Q-Learning, Expected Sarsa, etc., can be extended to generalization. As a result of generalization, when a single state is updated, the weights associated with the states are updated, which will affect many other states' values, which offers powerful learning but also offers less stability, however, with various modifications to the original algorithms and choosing the correct set of hyperparameters, powerful algorithms and agents can be built. In this report, function approximation is applied to solve the Lunar Lander problem.

2. Function Approximation

Function approximation in RL is like any other supervised learning algorithm, and it works by minimizing the error between actual values and target values by performing gradient descent. The differences are that training data is acquired incrementally during the agent's training, and unlike supervised learning, the target is not stationary. Both the actual values and target values are constantly moving. As a result, loss, which is calculated as the error between actual and target values in the Gradient descent algorithm, is not an accurate measure to assess an RL agent's performance during the training process. Hence, the convergence to a global optimum value function using Gradient Descent is not guaranteed. However, the global optimum value function is not necessary for obtaining the optimal policy. Various modifications to RL algorithms and the correct combination of specific hyperparameters during training prevent the algorithm from diverging and eventually obtains an optimum policy. In the following sections, a few RL algorithms are analyzed to solve the lunar lander problem better, and one of the best ones is chosen to implement the algorithm.

3. Choice of Algorithms

3.1 Linear vs. Non - Linear

First and foremost, function approximation can be either linear or non-linear methods. Linear methods are guaranteed to converge and are efficient because there is only one optimum in the linear case, and any method that is guaranteed to converge to near a local optimum is automatically guaranteed to converge to or near the global optimum. However, Linear methods need a careful selection of features that represent the whole state space which may need domain knowledge about the environment and are limited to no interactions between the features. In contrast, non-linear methods such as **Artificial Neural Networks** (ANN) with hidden layers can compute the abstract representations of raw input. Thus, eliminating the need to create the features beforehand to represent the state space. Hence, to solve the lunar lander problem, ANNs with hidden layers are chosen. There are two ways to approach the lunar lander problem with ANNs; by giving the entire state and action space as an input to the ANN and outputting only a single value of value function Q. Another approach is to pass only state space as an input and outputting Q value for each action. The latter approach is chosen because it is computationally straightforward; it reduces the number of forward passes required for each state-action pair. Also, this approach is convenient and similar to Q-learning, which is the RL algorithm of choice in this report.

3.2 ANN in Lunar Lander

Since Lunar Lander is a control problem, we estimate the value function in state-action pairs, i.e., we learn Q(S, A). To represent control function as a function approximation with respect to weights, we use the notation $\hat{q}(S_{t+1}, a, \mathbf{w}_t)$. We attempt to learn a policy from the value function by following the below equation (1). This is possible because the lunar lander has a discrete action space. Hence, argmax can be applied, and it yields the best action corresponding to that state.

$$A^*_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) \qquad (1)$$

In the above equation $\hat{q}(S_{t+1}, a, \mathbf{w}_t)$ is estimated using an ANN. Firstly **loss** is calculated after a forward pass through the neural network, based on the error between target and actual values. The Gradients are calculated using the **Backpropagation** algorithm, and the algorithm tries to minimize the loss by performing a Gradient descent step, as shown in equation (2).

$$\mathrm{w}_{t+1} \doteq \mathrm{w}_t + \alpha[U_t - \hat{q}(S_t, A_t, \mathrm{w}_t)]\nabla\hat{q}(S_t, A_t, \mathrm{w}_t) \qquad (2)$$

Where U_t is the target, choice of RL algorithm determines the target U_t which is discussed in the next section. Methods used in (2) are called Semi Gradient methods because U_t also involves w_t but gradient descent is only performed in the direction of $\nabla\hat{q}(S_t, A_t, \mathrm{w}_t)$. Hence, (2) is not a true gradient method.

3.3 RL Algorithm

This section analyses which RL algorithm is best for solving the Lunar Lander problem. First and foremost, the choice would be between on-policy and off-policy control. One example of on-policy is Episodic Semi – Gradient Sarsa (SB 20, Pg. 244); in this algorithm target U_t uses the same ANN $\hat{q}$. Then, similar to TD (0), the weight updates are calculated as shown in equation (3). The only difference between TD (0) and equation (3) is the target $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathrm{w}_t)$ is non-stationary.

$$\mathrm{w}_{t+1} \doteq \mathrm{w}_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathrm{w}_t) - \hat{q}(S_t, A_t, \mathrm{w}_t)]\nabla\hat{q}(S_t, A_t, \mathrm{w}_t) \qquad (3)$$

On-policy algorithms are simple to implement and computationally efficient. However, since the training data is incrementally collected in the lunar lander problem, there will be strong correlations between training examples, and the agent may learn the best actions based on these correlations, and that agent may perform worst when those correlations are not present. Hence, to effectively address this problem, certain modifications are made to the algorithm. One technique is to use experience reply (MinhEtAlHassibis 2015), which records <state, action, reward, next state> pairs and stores it in a buffer. Later during the update step, training examples are randomly sampled from this buffer, thus eliminating the sequential correlations between pairs of training examples. Since the learning depends on different state-action pairs, the algorithm is no longer an on-policy, hence used off-policy Algorithm (Q-learning) for solving lunar lander. Q - learning algorithm is modified to work efficiently with function approximation, which is explained in the subsequent sections.

4. DQN Implementation

As discussed in the previous section, the off-policy algorithm Q - learning has chosen, but function approximation with off-policy learning comes with various challenges. The main challenge is that the algorithm is not robust and has a very high divergence chance, mainly because of bootstrapping the target Q value function with the approximate estimate (3). Due to this, loss, which is calculated while training the agent with ANN, is not accurate. As a result, convergence is not guaranteed. Another Neural network (MinhEtAlHassibis 2015) is created to mitigate this issue, which is used only for the target action-value function Q' in Q-learning update (3). This ANN has the same number of layers and weights as the actual Q network, but the weight updates are not updated as frequently as the actual Q function. I.e., the updates are done at fixed intervals, this hyperparameter (target network update frequency), and its effects are analyzed in the later sections. This approach stabilizes the algorithm by eliminating an unwanted disadvantage of generalization, which is an update to one single weight parameter may cause updates to several similar states in the network. Note that this is generally an advantage of generalization, but in the RL context, since Q-learning update (3) uses the bootstrapping rather than actual value to learn, generalization is a disadvantage because an update to $\hat{q}(S_t, A_t, \mathrm{w}_t)$ may affect the $\hat{q}'(S_{t+1}, A_{t+1}, \mathrm{w}_t)$. However, using the separate network for target Q function(Q') and introducing a delay to update that network will eliminate such a scenario.

4.1 Algorithm for training the agent

```
1.  Initialize experience replay memory buffer D with a fixed capacity N.
2.  Initialize two separate ANNs with the same random weight parameters. One is the actual action-value Q function $\hat{q}$, and another is the target Q' function $\hat{q}'$. ANN uses two hidden and two input-output layers, with the weights (8,128), (128,64), (64,32), (32,4) at each layer.
3.  For each time step, per episode in the environment, select an action $A_t$ from state $S_t$ using the € - greedy policy.
4.  Execute the action $A_t$ and observe reward $R_{t+1}$ and next state $S_{t+1}$.
5.  Store this pair in the replay memory D as ( $S_t, A_t, R_{t+1}, S_{t+1}$).
6.  When buffer size(k) > buffer delay(r), sample a random mini-batch of size consists of ($S_t, A_t, R_{t+1}, S_{t+1}$) pairs from replay buffer.
7.  Calculate target $U_t = R_{t+1} + \gamma \max_{A'} \hat{q}'(S_{t+1}, \mathrm{A}', \mathrm{w}_{t'}) * (1 - done(S_{t+1}))$.
8.  Gather Q ($S_t, A_t$) from $\hat{q}(S_t, A_t, \mathrm{w}_t)$ Neural Network.
9.  Compute loss using Mean Square Loss (MSE) between values computed from (7) and (8).
10. Calculate gradients using the backpropagation algorithm.
11. Perform a Batch Optimizer step (Batch Gradient descent) based on equation (3) above.
12. For every C (target network update frequency) step, reset $\hat{q}$ and $\hat{q}'$ to same weight values at every layer in ANN.
13. Repeat the above steps until the agent hits the mean reward of 225 for 100 consecutive episodes.
```

Algorithm 1: Pseudo code for DQN Algorithm with Experience Replay

4.2 Hyperparameters

Hyperparameters in the above algorithm were selected by manual exploration; this was enough for an agent that solves the lunar lander problem. A grid search through the parameter space is recommended to extend this algorithm to more complex problems. Grid search is not used here due to computational requirements.

Learning rate(α) = 0.0005	Gamma(ϒ) = 0.99	€_ start = 1.0
Batch Size (K) = 32	Target Update Frequency(C) = 1500	€_ end = 0.1
Replay Buffer Size(N) = 70K	Buffer delay(r) = 7 * K	€_ decay = 0.99

The Q learning functions, which are initialized in step (2) above in Algorithm 1, have two hidden layers and two input and output layers. Each layer used RELU activation function except the output layer, and this non-linear activation function enables the neural network to learn complex abstraction within the Q function. Adam Optimizer is used to perform gradient descent steps, and MSE loss is used to calculate the loss (or) error. The learning rate should be kept sufficiently small as the algorithm learns from a moving target; hence, faster learning based on loss function is not accurate. More about the effects of various learning rates are discussed in the later sections. Gamma is kept to 0.99 because a low discount factor makes the agent act myopically, so the agent may never want to gain the reward by landing and keeps rotating in circles forever; hence, higher the value of a discount factor is needed as we are prioritizing the successful landing of the agent. Too high gamma may also cause divergence as the agent always try to land and keeps on crashing every time.

Batch size(K) at 32 was perfect for a replay memory buffer size (N) of 70000. Since the replay memory size is large, memory holds the training examples of various diverse possible state-action pairs; hence, small batch size prevents the algorithm from bias in error calculation. The advantages of maintaining a large-size memory buffer are analyzed in the later section. € decay is used for the value of € for an €-greedy policy in step (3) of the algorithm above. € value is started with 1.0, which ensures agent try different actions and slowly decreases as the learning goes ahead with the decay of 0.99*€ per episode. Epsilon decay ends at € value of 0.1, which ensures the agent to try random actions even when it reaches peak performance. Target update frequency(C) is another crucial hyperparameter, and its importance is already discussed in the previous section. Using a value of 1500 offered a significant performance improvement, C's effects are analyzed in the next section.

4.3 Training the agent - Results and Analysis

Fig 1. Performance of an DQN agent while training

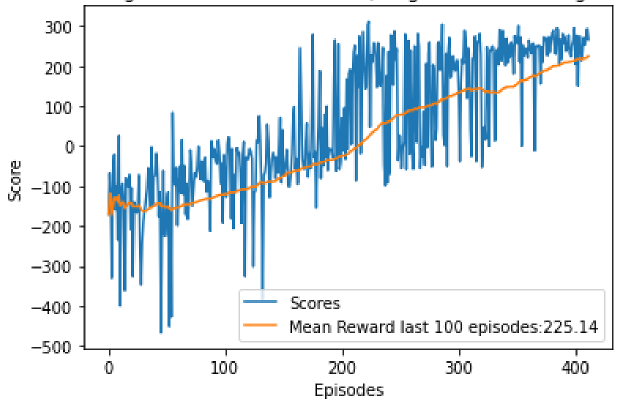
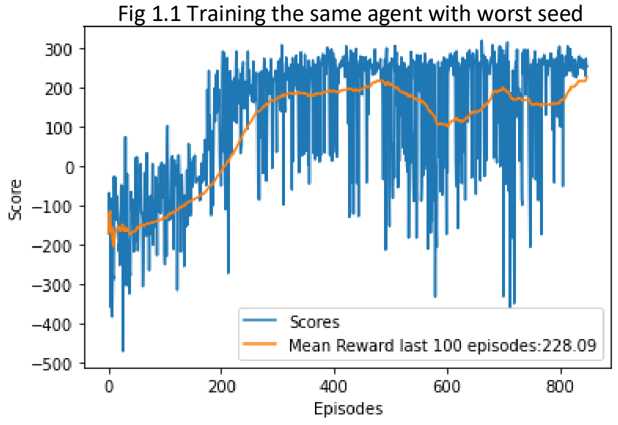


Fig 1.1 Training the same agent with worst seed



The agent is trained till it reaches an Average reward of 225 for the consecutive 100 episodes; when the agent reaches the desired reward, the learning process is stopped, and the trained ANN model with the action-value function $\hat{q}$ is saved to test the model's performance in the next section. The graph in fig1 shows the trend of reward for each episode using the blue plots and the mean reward for the last 100 episodes using the orange line. Initially, as the agent tries to explore different state-action pairs, the agent can make several crashes, so huge negative rewards are recorded until 70 episodes. Another possible reason for low scores is because learning is paused, and sampling from the memory buffer is not done until the buffer contains (7 * batch size) no. of examples. As a result, the initial actions are entirely random, which is essential because buffer at the end of the 70 episodes would store many states and actions. So, in around 70 episodes, the memory buffer has enough representative samples, and then its effect is clearly shown in the following episodes as large negative rewards are eliminated by learning from those experience samples. Few episodes recorded a significant error, but those are probably the state actions spaces that the agent never explored during the initial stages. Another critical observation here is once the never seen examples are recorded, the agent takes time to learn from them, which is evident from episodes 110 to 130 where agent do not immediately learn about these examples; hence these large rewards are only disappeared after around 130 episodes. Here this slow learning is deliberately introduced to prevent the algorithm from diverging. As discussed in the previous sections, slow learning is achieved by introducing a separate target network and a target network update parameter(C). Later the algorithm slowly learns the good actions by reducing the loss function, improving the performance as the number of episodes increases. The above learning pattern is not guaranteed to be similar in all the runs, as the agent's actions are random initially. However, one important observation is that even if the (states, actions) encountered by agents initially are not sufficient to learn from, the algorithm can learn whenever the new unexplored examples occur. To prove this scenario, an auxiliary experiment is conducted with the same algorithm but experimented with different seeds to repeat the worst-case situation, i.e., in a situation where random actions at the beginning are not sufficient to learn all the possible examples. The results are presented in fig 1.1. In this scenario, the agent explored a small set of actions (due to randomness) at the beginning that resulted in less negative rewards. When the agent reached 450 episodes, the mean reward of the last 100 episodes (orange line) also decreased rapidly. It almost went to 100 from 190, but the agent learned from those actions too. Around 600 episodes agent was able to correct those actions and eventually reached the desired reward. This phenomenon is also one of the disadvantages of using a separate target network as it causes slow learning, but in non-linear function approximation, the algorithm must prevent oscillations and divergence. The agent may take longer, but eventually, it is trained successfully. The difference between fig.1 and fig 1.1 clearly showed the importance of exploration vs. exploitation in the context of RL.

4.4 Testing the agent - Results and Analysis

Fig 2 Performance of trained agent from Fig 1

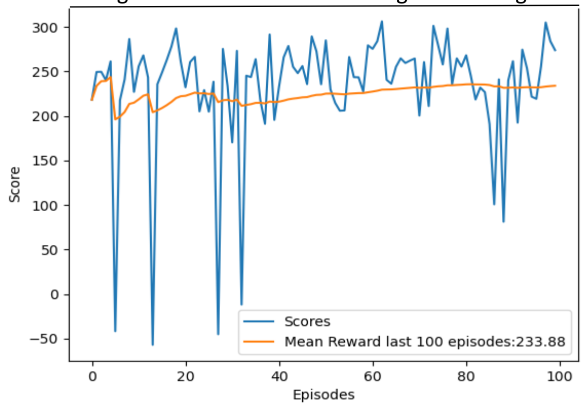
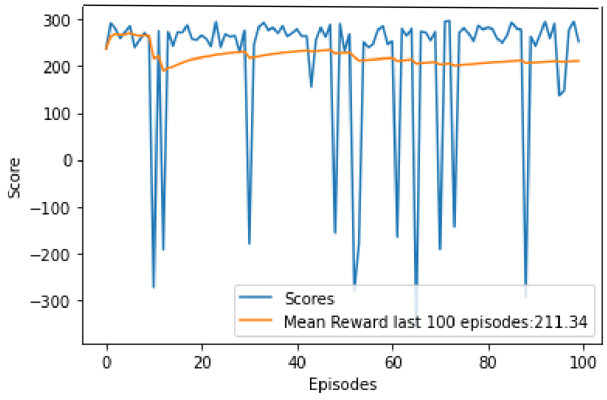


Fig 2.1 Performance of trained agent from fig 1.1



Both the trained agents from figures 1 and 1.1 from the above are tested in different environments. In fig2, the agent scored the highest average reward over 100 episodes, only a few episodes recorded a score less than 0, and the remaining episodes performed very well. Whereas the trained agent from fig 1.1 performed well overall (shown in 2.1), but the agent crashed in some episodes, recording huge negative rewards, but the agent successfully solved the lunar lander problem by recording an average of over 200 for 100 consecutive episodes. This result shows the importance of the initial exploration of the agent when the learning started. When the agent explores as many as states, in the beginning, buffer quickly populates with representative examples, and this allowed the agent to perform well in fig 2. However, using the techniques and set of hyperparameters described above while training the agent eliminated randomness, which is proved from Fig 2.1.

4.5 Pitfalls and resolutions

There were few challenges while training the agent, which led to the agent's bad performance during the testing. The main challenge was the agent not exploring all the possible training examples due to randomness in the environment; thus, the replay buffer does not have enough representative samples to learn. A new hyperparameter is introduced to resolve this issue - replay start size(r) set at seven times the minibatch size by manual exploration. Furthermore, the algorithm continues till the agent achieves a reward of 225 or more (previously, it was 200); this change allows the agent to continue learning to gain the maximum reward. Learning may be slow at this point but proved essential to gather large rewards during the testing. Also, € is not decreased below 0.1 to address the cases like figure 1.1, i.e., to make the agent try different actions at the end of the learning.

5. Effect of Hyperparameters

Fig 3. Training of the DQN agent for different replay buffersizes

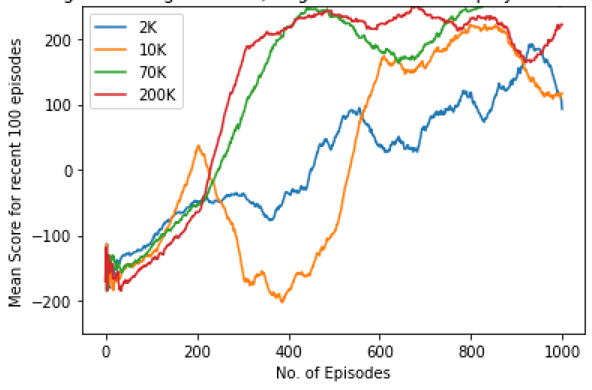


Fig 3. Shows the effect of **Replay memory buffer size** on the process of training the agent. When the batch size is too small (2k), the agent learned very slow, but it did not reach the reward of 200 in 1000 episodes. Also, there are many fluctuations in the graph, suggesting the lack of suitable training examples for the agent to learn about the environment. At this buffer size, the agent is slow to learn and has a very good chance to overfit as the agent learned from the minimal information spread through many episodes. When the buffer size increased to 10k agent learned very fast and reached the mean reward of 0. However, later, a sudden large dip in the average scores clearly shows that 10k is also not enough buffer size for the algorithm. Later, the algorithm recovered from the significant dip slowly when the buffer contains representative samples but never reached 200. When the buffer size increased more (70k), the agent acquired the target reward around the 400th episode, but when the agent made to run longer, the average reward is reduced slightly, but then it picked up and continuously increased (>250) till the 1000th episode. During the experimentation, it is discovered that this is a critical quality of an RL agent; it

should learn whenever there is a dip in the performance due to new states or generalization in the neural network. Agents who showed similar performance patterns while training performed well during testing in a new environment. When the buffer size increased to 200k, the agent now stores information of many recent episodes, this is a vast buffer size for this problem, but surprisingly it performed well and never showed any large dips in the performance. However, the algorithm might underfit because, at every step, only 32 out of 200k is sampled to calculate the loss. Hence huge buffer sizes are restrained from using in this problem.

Fig 4. Shows the effect of **learning rate** on the process of training the agent. The learning rate is used to reduce the neural network's loss while attempting to learn a function approximation. First of all, the learning rate should be minimal because the target function is bootstrapped, and quick jumps towards the target will cause divergence. It can be observed from the highest learning rate (0.1) in fig 4. Secondly, a low learning rate is also not sufficient because the agent takes different actions based on the network's values, and if the learning rate is too low, the agent may not choose correct actions as the network produces incorrect values, and the algorithm will get stuck. In the graph, it is evident for the lowest learning rate; the agent never reached more than -100 reward. The learning rate of 0.005 also proved to be large as the mean score continued to be diverging after 100. The algorithm is performed best when the learning rate is 0.0005; it is a nice balance between too low and too high values. Similar to fig.3, the mean reward decreased after reaching 400 episodes but improved later.

Fig 4. Training of the DQN agent with different learning rates

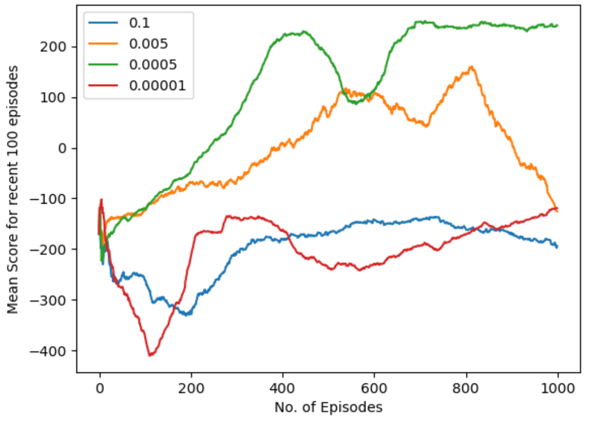


Fig 5. Training of the DQN agent with different target update frequencies

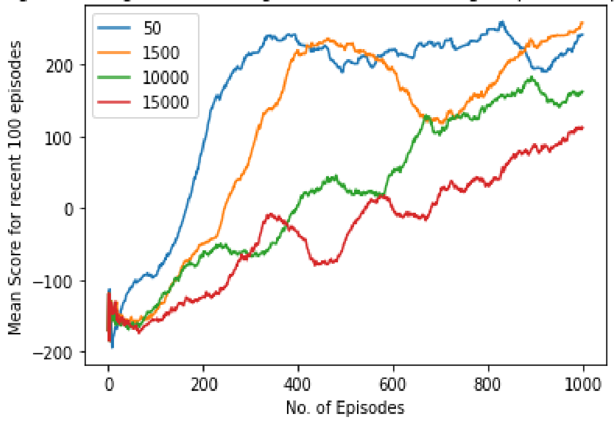


Fig 5. Shows the effect of **target network update frequency**(C), when the C is too high >10K, the agent is slow to learn because the target network contains old values, and loss computed will be minimal as the actual network almost moved towards the target network way before 10k update steps of gradient descent. Hence, many more episodes will be needed to converge the values of large update frequency. The primary purpose of introducing C is to make the learning process slower without causing divergence, as the learning rate is susceptible because of bootstrapping. Hence, even though the agent performed well with 50, 1500 is chosen to maintain a decent amount of delay in the target network updates, ensuring the agent explores different states.

6. Conclusion

This report shows that the Lunar-Lander problem can be successfully solved using a DQN algorithm with various modifications like experience replay buffer, target network update delays, and initial buffer delay. It also addresses the challenge of dealing with the Deadly Triad (SB 20 pg.264), i.e., when an algorithm contains function approximation, bootstrapping, and off-policy, the algorithm has the danger of instability and divergence. However, using separate ANN for target, introducing a target network update frequency(C) parameter, experience replay technique, and combining specific hyperparameters' values helped the algorithm handle the deadly triad. Epsilon decay strategy ensured the agent exploration higher initially, thus populating replay buffer with representative samples. However, due to randomness in the initial exploration and due to large continuous state space, the agent cannot explore enough samples to converge quickly; in those cases also, the algorithm performed well, as shown in figure 1.1. However, to improve the training performance in those cases, priority can be given to good examples in the replay buffer; good examples are which yield improvements in the mean reward; these can be found out during training. In the later stages of the training, examples can be sampled according to this priority. This technique eliminates the disadvantage of buffer sizes storing only recent examples, and with this technique, large buffer sizes can be safely used. Another important property for a learning agent to perform well is its exploration strategy, this algorithm used decaying epsilon strategy, but simulated annealing can also be used here to vary the € value according to the situation. Another technique is to try random restarts to prevent the algorithm from converging to the local optimum. All these techniques can be implemented to the DQN algorithm presented above, and these techniques guarantee performance improvements, but these are not implemented in this report due to time limitations.

7. References

[SB 20] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2nd Ed. MIT Press,2020.

[MinhEtAlHassibis 2015] Human-level control through deep reinforcement learning.
https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf