CS-7642 Project 2 Report – Lunar Lander

Puneeth Kumar Chana Reddy (preddy61@gatech.edu)
23966a93afffb927f86b8d93a81cc19c9976e317 (Git hash)

***Abstract*—In this paper, we discuss about model free control methods in Reinforcement learning, Value-function approximations using neural networks. We also specially focus on Deep Q-Network (DQN), a type of model free control algorithm used to solve OpenAI Gym's Lunar-Lander-v2 environment.**

I. Introduction

ALgorithms that purely sample from experience such as Monte Carlo Control, SARSA, Q-learning, Actor-Critic are "model free" Reinforcement learning algorithms. They rely on real samples from the environment to alter behaviour.
Q-learning was used to solve the Taxi environment in HW4 where-in the state space was discrete and hence representing a Q table using a simple 2D matrix was viable. In the Lunar Lander environment, the state space is continuous, hence maintaining a look up table is practically impossible. Therefore non-linear function approximations like neural networks are used to predict value estimates.

II. Model Free TD Control

Temporal Difference (TD) prediction methods generate the value estimates for the states. TD for the control problem categorizes algorithms into On-policy and Off-policy control. Off-Policy learning algorithms evaluate and improve a policy that is different from policy that is used for selecting the actions. On the other hand, the On-Policy learning algorithms evaluate and improve the same policy that is used for selecting the actions. Advantages of Off-Policy learning methods are

- **Continuous exploration**: Agent can learn optimal policy while following an exploratory policy.
- **Learning from demonstration** : Agent can learn from the other agent or human movements.
- **Re-using policies** : Agent can re-use experiences generated from old policies. This way the agent can use an old policy which generated a good reward instead of evaluating it again.

A. Q-learning

Q-learning is one of the TD based off-policy model free control algorithm and it is known to be one of the early breakthroughs in reinforcement learning
Q-learning seeks to find the best action to take given the current state. It is considered an off-policy learning algorithm, as it learns the value of the optimal policy independently of the agent's actions. The update rule for the Q-learning is shown below:

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

In the training process of Q-learning, it is not required to learn Q-values for all possible action-value pairs for the next state. It simply needs to learn the best ones generated from experience. For incremental reinforcement learning algorithms, according to Sutton's book most of the incremental algorithms are guaranteed to converge to optimality.

B. Value Function Approximation

Solving an MDP with a small state space can be done using a look up table, however when the state space in large enough then two problems are encountered, the first being running out of memory to build the gigantic look up table. The second problem is that, even though the memory is huge, exploring the entire space for estimating values would be very slow.
Value function approximation tries to build some function to estimate the true value by creating a compact representation of state space using smaller number of parameters.

$$\hat{v}(s, w) \approx v_\pi(s) \quad (2)$$

$$\hat{q}(s, a, w) \approx q_\pi(s, a) \quad (3)$$

where w is the weights of value function approximator used to estimate the value function across the entire state or state-action space.

Function approximations can be done in following ways

- **Linear Function Approximation**: A linear function approximator is a function that is linear in the weights and the inputs. The weights and states(inputs) only have linear mappings as shown below

  $$\hat{v}(s, w) = w^T x(s) = \sum_{i=1}^{d} w_i x_i(s) \quad (4)$$

  Linear function approximators have several nice properties like, For supervised learning, the weights can be calculated by a simple matrix inversion, without any gradient-descent or incremental learning.
  In linear function approximation, features needs to hand crafted which requires deep domain expertise. This is one of the disadvantages of linear function approximation over non-linear function approximations. Also, linear function approximators may not be useful

in cases where the input space is high dimensional.

- **Artificial Neural Networks** : Neural networks are simple, nonlinear function approximators. A weighted sum of the inputs is passed through a nonlinear function, typically a rectified linear activation function or ReLU. The output of the first layer is then passed to other layers to get outputs for several neurons. Neural networks often use error backpropagation algorithm to update the weights. This algorithm simply does gradient descent on the squared error in the outputs of the network.
  One of the advantages of neural networks is its capability of auto feature creation. The hidden layers along with the weight between neurons automatically create and learn features based on the experience for a given problem. This saves the efforts of hand-crafting the features.

C. Stochastic Gradient Descent & Loss Function

In most learning networks, error is calculated as the difference between the actual output and the predicted output.The function that is used to compute this error is known as Loss Function. One of the most widely used loss function is mean square error, which calculates the square of difference between actual value and predicted value.
In the context of reinforcement learning, we need the function approximation algorithm to be able to learn online and handle non-stationary target functions. The objective of such function approximation is to minimize the Mean Squared Value Error (MSE) on the value function which is defined as:

$$MSE = \sum_{s \in S} \mu(s)[v_\pi - \hat{v}(s, w)]^2 \tag{5}$$

where μ(s) represents the weights over state space (not all states are treated equally), $v_\pi(s)$ as the true value estimate for state s and $\hat{v}(s, w)$ is the parameterized function approximation for state s.

Stochastic Gradient Descent (SGD) is widely used in function approximation methods. Gradient descent is an iterative algorithm, that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function. SGD performs the weight updates after each experience by a very small amount (defined by learning rate α) in the direction that reduces the MSE most. SGD update formula as follows:

$$w_{t+1} = w_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t) \tag{6}$$

where $v_\pi(S_t)$ is the true value of state S_t given policy π. But in the TD control context, where the true value is unknown, therefore $v_\pi(S_t)$ is replaced with the TD target

$$R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) \tag{7}$$

III. Lunar Lander & Deep Q-Network

In this section we will discuss about the Lunar Lander problem set up. The states space, actions taken by the lander and the reward structure of the environment are discussed in detail. Later on we look at the Q-Network's shortcoming in solving the Lunar problem and how the Deep Q-Network has n edge over the Q-Network in this environment

A. Lunar Lander Set Up

Lunar Lander environment is provided by OpenAI gym. This environment deals with the problem of landing a lunar space lander on a landing pad at the target coordinates (0,0). The state space, actions and the reward structure are described below.

- **State Space:** The state space is continuous in this environment. At each time step the state is provided to the agent as 8 dimension tuple (x,y,v_x,v_y,θ,v_θ,leg_L,leg_R).The state variables x and y are the horizontal and vertical position, v_x and v_y are the horizontal and vertical speed, and θ and v_θ are the angle and angular speed of the lander. Finally, leg_L and leg_R are binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground.

- **Actions:** There are four possible actions namely [do_nothing,fire_left_engine,fire_main_engine,fire_right_engine]

- **Reward Structure:** Lander's base reward for moving towards the landing pad is decided by various factors. Lander is penalized if it moves away from the landing pad. An episode is terminated if the lander comes to rest or crashes, receiving -100 or +100 points additionally on top of the base reward. Each leg-ground contact is worth +10 points. Firing the main engine incurs a -0.3 point penalty and firing the orientation engines incur a -0.03 point penalty, for each occurrence.

As the state space and actions were discrete in the Q Network used in HW4, a look up table was used to hold the Q values for each action. On the other hand when the state space is continuous, as described in this problem, Q-Network's approach of using look up table is practically impossible as described in the earlier sections. In this problem setup, we are required to find the dependency of each parameter over other, making a simple linear function approximations practically not usable. Therefore a more complex non linear function approximations like Artificial neural networks(ANN) is used in this context.

B. Deep Q-Network (DQN)

In this sub section we will discuss about one of the first model free control algorithm with artificial neural networks called Deep Q-Network (DQN). This algorithm combines Q-Network with ANN to produce satisfactory results by using the features produced by deep neural network in the learning process. Furthermore, various hyper parameters involved in constructing DQN are studied. DQN used the following semi-gradient form of Q-learning to update the network's weights:

$$\Delta w = \alpha[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)] \nabla\hat{q}(S_t, A_t, w_t) \quad (8)$$

where w_t is the vector of the network's weights, A_t is the action selected at time step t, S_t and S_{t+1} are current and the next states respectively at time steps t and t + 1. Mnih et al. modified the basic Q-learning procedure using the **Experience Replay** and a duplicate Q-Network called **Target Q Network** to stabilize the weight updates.

C. DQN with experience replay Algorithm

Algorithm 1: DQN Algorithm

```
Input: Lunar Lander Gym Environment
Output: PyTorch ANN model with action-value
  function approximation
Initialize replay memory D to capacity N
Initialize update network frequency C
Initialize Q function with random weights $\theta$
Initialize target $\hat{Q}$ function with weights $\theta^- = \theta$
for episode 1 to max episodes do
    Reset Gym Env to get current state;
    Initialize score per episode;
    for step 1 to max steps do
        $\epsilon$-greedy to select the best action $a_t$;
        execute action $a_t$ and observe reward $r_t$, next
          state $s_{t+1}$;
        add experience ($s_t$, $a_t$, $r_t$, $s_{t+1}$) to replay
          memory D;
        if step == C then
            sample mini-batch from D
            use $\hat{Q}$ to generate target estimates $y_j$ on
              mini-batch
            use Q to calculate predicted estimates $p_j$
            perform SGD on the loss function $(y_j-p_j)^2$
              with respect to $\theta$
            perform soft target update
        end
    end
end
```

- **Experience Replay:** Q-learning with experience replay provides several advantages over the standard Q-learning. The process of storing and re-using experiences for many updates allows DQN to effectively squeeze more data from past experiences. Additionally, experience replay reduces the variance of the updates by randomly sampling the data to eliminate the correlation between the successive updates as they would be with standard Q-learning.

- **Target Q-Network:** The issue with Q-learning without target network is that at every step of training, the Q-network's values constantly shift, if we use these values to adjust our network values then the value estimations can easily be destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate this, the target network's weights are fixed, and only a small percentage of weights are updated frequently from the primary Q-network, this makes the training more stable. Transferring a small part of the weights frequently to the target Q-network instead of copying all weights at once is called a soft update as described in the below equation

$$\theta^- = \theta * \tau + \theta^- * (1 - \tau) \quad (9)$$

The problem with the copying all weights at once is that, all the weights are smoothly copied with the same speed from the primary Q-network, even though some of them are updated towards the wrong directions. This behavior increases the risk of generating sub optimal weights. Although reducing the overall update speed is not a right way to eliminate wrong updates, it would rather decrease the learning speed. To efficiently update the weights while keeping learning speed, a soft update method is used as described above.

By introducing the above changes the update equation changes to:

$$\Delta w = \alpha[R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)] \nabla\hat{q}(S_t, A_t, w_t) \quad (10)$$

where $\tilde{q}$ is the estimate from the duplicate network

IV. Experiments & Results

In this section we will discuss about the implementation details of the DQN algorithm.Results and graphs of the trained DQN agent. Lastly, we will discuss about the effect of three hyper parameters on the learning.

A. DQN Hyper Parameters

- Hidden Layer Nodes: **64 X 64**
- Experience Buffer Capacity: **100000**
- Batch Size: **32**
- Learning Rate: α**=0.001**
- Discount Factor: γ**=0.99**
- Epsilon Decay: **0.99**
- Network Update Frequency: **C=2**
- Soft Update Rate: τ**=0.001**

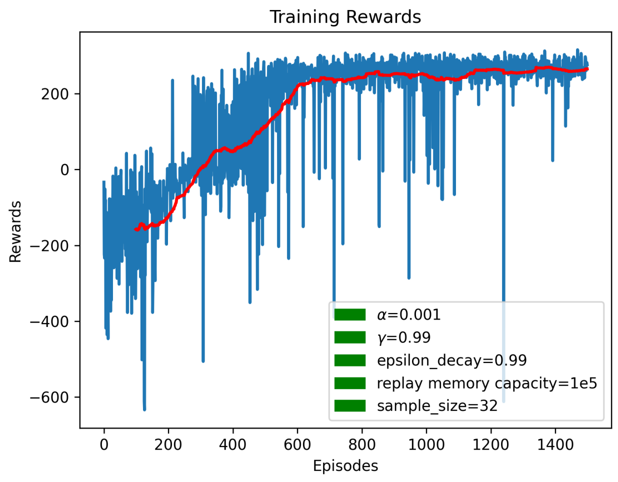

Fig. 1. The above figure shows rewards per episode over 1500 training episodes of the DQN agent.

As seen in the Figure 1 the rewards during the initial episode are negative because the agent is learning during this phase. Eventually, the agent starts to learn and the average reward starts to shoot up and becomes positive after 350 episodes. At around 520 episode the average rewards goes above 200. Furthermore, the agent is allowed to run through 1500 episodes to gain more experience. As we see the average rewards after 550 episodes stays more or less stable but we still see some occasional negative rewards for some episodes, this may be due to the fact the agent wouldn't have encountered such episodes. These instances will reduce if the agent is allowed to train for more episodes.

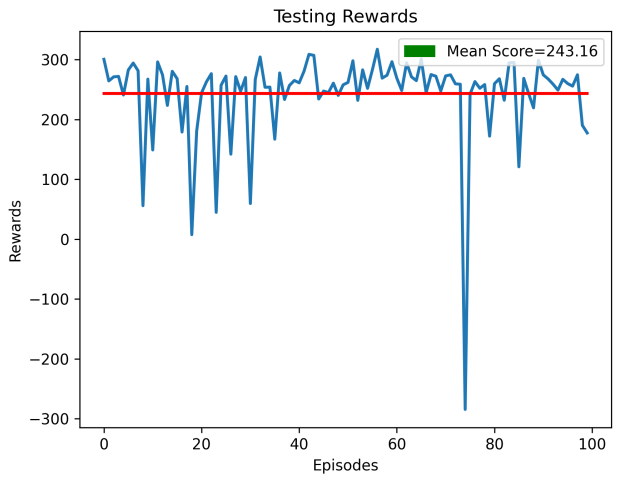

Fig. 2. The above figure shows rewards per episode over 100 consecutive testing episodes of the DQN agent.

As noted above, the trained agent produces a mean score of around 243 for 100 consecutive episodes. The occasional negative reward episodes may be attributed to insufficient training of the agent. By adding more stringent constraints like "no negative rewards for an episode" during the training process would eliminate such odd instances.

B. Effect of Hyper Parameters

In this section we will discuss about the effect of 3 hyper parameters namely Learning Rate (α), Discount Rate(γ) and Epsilon Decay on training process. The experiments are conducted for 1500 episodes with 1000 steps per episode for multiple values of each of the parameter.

- **Learning Rate** (α):The learning rate ranges between 0 and 1 and it determines how much the weights are updated in the direction to minimize the MSE (mean squared error) in each pass. Setting the low value means learning happens very slowly and setting a very high value causes the learning to happen quickly but can sometimes can never find the minimum MSE.
For validating the effect of various learning rates on the model performance different values of learning rates (1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}, 1e^{-1}) are chosen for this experiment. Best performance is observed for the middle value of the learning rate of 1e^{-3}. The green line in Figure 3 corresponds to this value and provides the maximum reward. As discussed above, the agent is unable to minimize the MSE at the higher learning rate causing reward values to diverge.

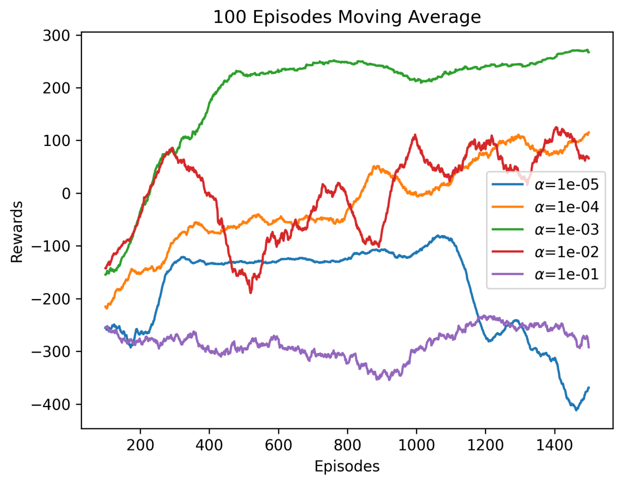

Fig. 3. The above figure shows rewards per episode over 1500 training episodes of the DQN agent for different learning rates

- **Discount Rate** (γ):The discount factor influences the accumulation of future rewards. A discount factor γ=0 will cause the agent to accumulate immediate reward, whereas a higher discount factor γ=0.99 will cause the agent to accumulate discounted future rewards.
For validating the effect of various discount rates on the model performance different values of discount rates (0.5, 0.623, 0.745, 0.867, 0.990) are chosen for this experiment. Best performance is observed for γ=0.99. The purple line in Figure 4 corresponds to this value and provides the maximum reward. In this project set up, to make a successful landing a large γ is needed to accumulate rewards for distant future, hence γ=0.99 accumulates maximum reward.

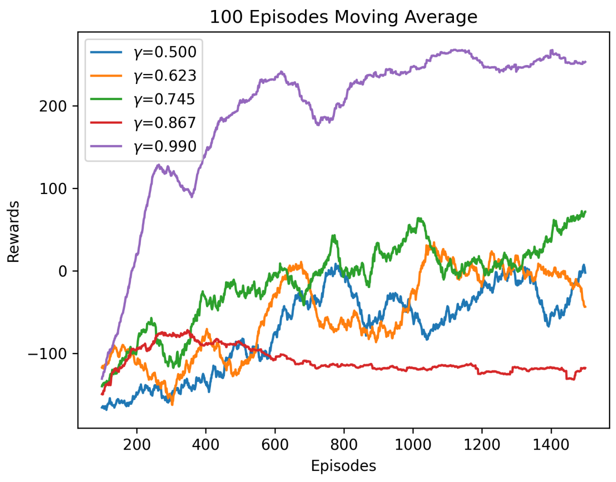

Fig. 4. The above figure shows rewards per episode over 1500 training episodes of the DQN agent for different discount rates

- **Epsilon Decay**:One way to balance between exploration and exploitation during training is by using the ϵ-greedy method. With ϵ probability a random action is chosen and with (1-ϵ) probability the "greedy" action with the highest Q-value is selected. The enhanced version of the epsilon-greedy method is called a **decayed-epsilon-greedy**. In this method, the agent has freedom to explore with a high probability during the initial phases of training, and then gradually the epsilon is decreased with a decay rate over subsequent training episodes.

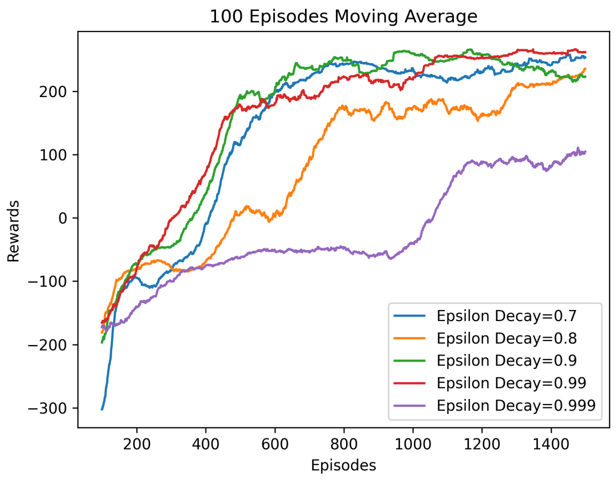

Fig. 5. The above figure shows rewards per episode over 1500 training episodes of the DQN agent for different epsilon decays

Figure 5 shows the variations in the rewards values for different values of the epsilon(ϵ) decay. Agent performance is worst for epsilon decay of 0.999 and the best performance is for epsilon decay values of 0.9 and 0.99 which is shown in green and red colors. This behavior might be because these values of epsilon decay is providing a better balance between Exploration-Exploitation. Higher epsilon(ϵ) decay means higher exploration caused due to gradual decrease in ϵ. Therefore other values like 0.7, 0.8 take more episodes to reach optimal values as the exploration is less during the initial episodes.

V. Implementation Pitfalls

- **Hyper-parameter tuning**: Tuning all the hyper-parameters to achieve optimal results in DQN is not only difficult but also time consuming. Number of episodes were reduced to e.g. 500 while training with different combinations of parameters until a satisfactory average reward per 100 episodes was achieved. After finding a good hyper-parameter combination, the agent was trained for 1500 episodes. This procedure saved a lot of time.

- **Choosing Neural Network Architecture**: Choosing number of neurons and hidden layers of the neural network was another issue that was faced during the implementation. After some trail and error, one hidden layer with 64 nodes were chosen for optimal performance.

VI. Future Work

Author would have explored the below points provided with sufficient time

- **Other Algorithms**:Implementing either SARSA with neural networks or any policy based algorithm to solve the Lunar Lander. Evaluating the results w.r.t rewards achieved by multiple algorithms would have given a better idea on what to choose in a particular environment.

- **Lunar Lander with uncertainties**: Uncertainties like Random Engine Failure, Random force acting on the lander or a noisy position information could be applied to the lander to see how the algorithm performed under such conditions. These uncertainties are more likely to occur in the real world.

VII. Conclusion

The paper has explored about model free control methods, value function approximations and in particular DQN using neural networks. Next, the paper presents the set up of Lunar Lander and DQN algorithm in detail. The experiments and results section talks about the training and evaluation model of the agent and also presents the effect of three hyper-parameters. Finally, the paper ends with pitfalls during the implementation and future works.

References

[1] Sutton, R. S., Barto, A. (2018). Reinforcement learning: An introduction. Cambridge, MA: The MIT Press

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M. et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533. doi: 10.1038/nature14236

[3] David Silver lectures on Reinforcement learning