CS7642 Project 2 Report – Lunar Lander

Minghan Xu (minghan.xu@gatech.edu)
b14160038442a9f2197bd060665c886536dc64c1 (Git hash)

***Abstract*— This project report discusses the fundamentals covering Q-learning, Value-function approximation, and the detailed implementations of Deep Q-Network (DQN) in solving the Lunar Lander problem.**

I. INTRODUCTION

In Homework 4, Q-learning is applied to perform approximation on the optimal Q-value function with a discrete deterministic MDP (taxi problem). With the ϵ-greedy algorithm of selecting the best action from the learned Q-values, the agent is able to identify the optimal policy and perform the right action accordingly. In project 2 problem of the lunar lander, the state space is no longer discrete but continuous, this poses challenges to using simple Q-learning to derive optimal action-value functions. Hence this paper discusses the implementation of the function approximation: Deep Q-Network (DQN) to estimate the Q-value in the continuous state space, perform ϵ-greedy algorithm to find the optimal policy, and successfully land the lunar spaceship .

II. TD CONTROL

In Project 1, Temporal Difference (TD) prediction methods generate the value estimates for the states. To use TD for the control problem. Sutton in the textbook brings the concepts of On-policy and Off-policy control. For On-policy methods, they attempt to evaluate or improve the policy that previously used to make decisions, while off-policy methods evaluate or improve a policy different from the one used to generate the training experiences.

A. Q-learning

Q-learning builds the foundation of the Deep Q-Network (DQN) implementation discussed in later sections of this paper. Sutton describes Q-learning as one of the early breakthroughs in reinforcement learning.

Q-learning is an off-policy control algorithm. It learns action-value function Q and directly uses it as an approximation of the optimal action-value function $q*$. This removes the dependence of policy evaluation in the learning process. The algorithm simply needs to update and remember the Q values generated by each action at each state in a Q-table. In the control part of the algorithm, it simply picks up the best action by checking the Q-table for any given state.

The update formula is as follows: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$

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. Sutton in the textbook also mentions the guaranteed convergence to $q*$ of Q-learning.

III. ON-POLICY PREDICTION WITH APPROXIMATION

At Chapter 9 of Sutton's textbook, various techniques of on-policy prediction with approximation are discussed. These techniques serve to build theoretical foundations for DQN learning and will be briefly discussed in this section. In Q-learning, the approximation of value function is represented as a Q-value table. In function approximation, it is in the form of parameterized function with the a weight vector w of d dimension: $w \in \mathcal{R}^d$. Given a policy and state, we approximate the value as a function of state s and weights w: $\hat{v}(s, w) \approx v_\pi(s)$. The rationale of applying function approximation is in its strength of generalization. With a single state being learned, all the weights of the paramertized approximation get updated as well, impacting not only that state but the value estimations of all states. Sutton suggests this property makes learning potentially more powerful but at the same time more difficult to understand and manage.

A. Update and Loss Function

Sutton describes the update on an estimated value function as shifting its value s toward update target u: $s \rightarrow u$. Updating at s induces generalization hence changes the estimation of many other states. In theory, any method of the supervised learning can be adopted for function estimation and weights update. 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 ($\overline{VE}$) on the value function which is defined as:

$$\overline{VE} = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2$$

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

B. Stochastic Gradient Descent (SGD)

Sutton states that stochastic gradient descent (SGD) methods are widely used of all function approximation methods and serve well in online reinforcement learning. The weight vector $w = (w_1, w_2, ..., w_d)^T$ has fixed number of real-value components. The parameterized and differentiable approximation function $\hat{v}(s, w)$ gets its w updated in a series

of discrete time steps: $t = 0, 1, 2, 3....$ The strategy is to minimize the $\overline{VE}$ error on all the observed experiences. SGD performs the weight updates after each experience by a very small amount (defined by learning rate α) in the direction that reduces the $\overline{VE}$ most. SGD update formula as follows:

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

where $v_\pi(S_t)$ is the true value of state S_t given policy π. However the true value of state S_t is unknown and is exactly what we are trying to approximate at the first place. Hence we have to rely on some form of approximation on $v_\pi(S_t)$ in the gradient descent. We denote such approximation as U_t, which is a possibly random and noise-corrupted version of $v_\pi(S_t)$. So the above formula can be re-written as:

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

If given U_t is an unbiased estimate ($E[U_t|S_t = s] = v_\pi(S_t)$), SGD is guaranteed to converge to at least local optima for a decreasing learning rate α.

C. Linear Method

In the textbook, linear methods are considered as a special case of function approximation, as the value function approximation and states only have linear mappings: $\hat{v}(s, w) = w^T x(s) = \sum_{i=1}^{d} w_i x_i(s)$. The SGD update for such linear case is:

$$w_{t+1} = w_t + \alpha[U_t - \hat{v}(S_t, w_t)]X(S_t)$$

where $X(S_t)$ is the feature vector representing state s at time t. The linear method is also shown to have convergence guarantees and being data & computation efficient. Nevertheless, to build linear function approximation in practice requires carefully hand-crafted features that best describe the state space and feature interactions. Additional features has to be constructed to capture possible interactions between features.

D. Nonlinear Method: Artificial Neural Network (ANN)

Artificial neural networks (ANN) has been widely adopted in the field of computer vision and natural language processing in recent years. Figure 1 illustrates the architecture of an ANN example. It has four input units and two output units. The network has two hidden layers each with a dimension of four. Non-linear activation functions and the sum of signals after the activation are applied between the neurons. Training of ANN is essentially updating the weights w between neurons using back-propagation. The weights generalize the behaviors of the system. One advantage of training ANN over linear method 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.

Putting into the context of function approximation and reinforcement learning. ANN can automatically create features in the hidden layers and generalize complex environments in the continuous state space. It is possible to map the state representations s to the input layer and action representations a to the output layer. The final values after output layers can be interpreted as the value approximation of taking action a at given state s. This is the essential formulation in using DQN to solve lunar lander later.

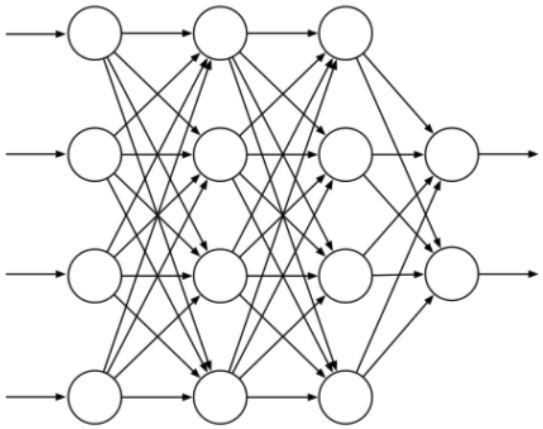

Fig. 1. A generic ANN architecture from Sutton's textbook

IV. Lunar Lander & Deep Q-Network

In this section, the problem settings and agent representation will be illustrated. Then Deep Q-Network (DQN) algorithm will be explained in details followed by actual solution design and code implementations. Constructing the DQN involves selecting multiple hyperparameters including learning rate, exploration-exploitation trade-off, and discount factors. The author will discuss parameter selection strategy and rationale to ensure the learning converge in a reasonable number of episodes. Experiments have been conducted to assess the impact to convergence and learning outcome from different sets of parameters.

A. Lunar Lander

The objective of the lunar lander problem is to successfully land the lunar space module at the target coordinates of $(0, 0)$. From environment definition, moving from the top of screen to the landing pad at zero speed gives reward 100 to 140 points. Additional -100 or +100 rewards are given if the module crashes or comes to rest. The solving criteria is to achieve average score of at least +200 over 100 consecutive trials. The state space is represented by a 8-dimensional vector: $[x, y, v_x, v_y, \theta, v_\theta, left\_leg, right\_leg]$, where x, y are 2-dimensional coordinates with v_x, v_y representing speeds on these two directions. θ and v_θ are orientation angle and angular speed of the lunar module. $left\_leg, right\_leg$ are binary indicators to show whether the leg has touched the ground. Among the eight dimensions: six of them are continuous numerical and two are discrete binary. There are four possible actions at any non-terminal state: $[no\_action, fire\_left, fire\_right, fire\_main]$

Analyzing the action-state space, it can be realized that simple Q-learning will not solve the problem as the state

space is primarily continuous. Unlike the taxi problem in Homework 4 where we can create Q-table to store every possible states and value estimates, the continuous spaces are infinitely large and requires the discretization on the state space for Q-learning implementation. Then assessing the simple linear method for function approximation, it brings the challenges of generating suitable feature representations. From the problem setup, it requires mathematical derivations to find interactions between angular velocity and speed on (x, y) coordinates. Hence non-linear approximation method like ANN seems a good fit for this problem.

B. Deep Q-Network (DQN)

Mnih et al. developed DQN algorithm which combines Q-learning with ANN. It has been shown that DQN is able to achieve satisfactory performance on the problems without having to use problem-specific feature sets. That is to say the learning algorithm is able to generate relevant features from the deep neural network in the learning process. The essential semi-gradient weights update formula is as follows:

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

where w_t is the nework's weights, A_t being the action at step t and S_t, S_{t+1} are consecutive states between t and $t+1$.

To improve the stability and prevent divergence for DQN, Minh et al. propose constructing two ANNs to enable bootstraping while still maintaining the supervised learning paradigm. For every C steps, of updates had been done to weights w of action-value network, they insert the network's current weights into another network and held the duplicates for the next C updates of w. So the update formula becomes:

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

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

Another novelty of DQN is the implementation of replay memory. Replay memory with the fixed size accumulates experiences over many past episodes and steps. For each learning step, a mini-batch sample is randomly drawn from such replay buffer. Unlike S_{t+1} becoming the new S_t for the next update in the simple Q-learning, unconnected/uncorrelated mixed samples are supplied for the weight updates. Such a technique makes more efficient use of the data and reduces the variance in the learning. As a result, experience reply eliminates one source of instability. The implemented DQN for lunar lander is illustrated in Algorithm 1.

Algorithm 1: DQN Algorithm for Lunar Lander

Input: Lunar Lander Gym Environment
Output: Keras ANN model with action-value function approximation

```
Init replay memory D
Init Keras action-value function model Q
Init Keras target action-value function model Q̂
for episode = 1 to max_episode do
    reset gym env
    for step = 1 to max_step do
        ϵ-greedy to select the best action a_t
        execute action a_t and observe reward r_t, new state s_{t+1}
        add experience (S_t, a_t, r_t, S_{t+1}) to replay memory D
        for every C steps do
            sample mini-batch from D
            use target action-value function Q̂ to generate target estimates on mini-batch samples
            target estimates update using semi-gradient method
            using the updated target estimates to train action-value model Q
        end
    end
end
```

C. Considerations in Implementations

Network size decides the general complexity of the DQN. A larger network has more weights to update and takes longer time to train. However it brings a finer granularity to differentiate the continuous state space. In contrast, a smaller network is faster to train but lacks the resolution to tell difference between similar states. From the empirical implementation and referencing to other DQN implementations, 2 hidden layers with size of (64×64) are implemented.

Replay momory with a fixed size stores past experiences and is used to draw samples for the weights update, which helps the efficient re-use of data from past trials. The size of replay buffer impacts the learning. In the early implementation of the algorithm, a smaller buffer size of 2000 was used and shown to be insufficient to store successful landing experiences. The agent tends to constantly fire engine and hover in the air (to prevent crash) rather than reducing throttle to slowly land. To address this, a much larger buffer of size 100,000 is adopted to ensure the network learn from the successful landing from the replay memory.

In the initial implementation, after drawing the mini-batch sample from the replay memory, the weights of action-value network Q got updated sequentially by each action-state transition. This caused serious performance issue of slow learning. The solution in Algorithm 2 is to update target action-value sequentially and to train the action-value model Q in a batch-processing fashion.

V. Experiments & Results

A. Trained Agent

The trained agent adopts the following parameters

- discount rate: $\gamma = 0.99$
- learning rate: $lr = 5e^{-4}$
- soft update rate: $\tau = 1e^{-4}$
- hidden layer size: (64×64)
- mini-batch size: 64
- update every $C = 4$ steps
- $epsilon\_decay = 0.995$

Algorithm 2: Learning/Weight Update in DQN

Input: Mini-batch samples from replay memory D
Output: update the action-value model Q

for *every C steps* **do**
  for *each sample* **do**
    get $state, action, reward, new\_state, done$ from the sample
    if *terminal state* **then**
      $target = reward$
    else
      $target = reward + \gamma * Q^*_{new\_state}$
    end
    accumulate $state$ and updated $target$ values
  end
  batch training on the value-action model using the accumulated states and target values$Q$
end

Figure 3 illustrates the rewards at each training episode till the success condition is met at episode 627. As we could observe the global improvement when the episode number increases which indicates convergence. However, at the local level after episode 527, we are still seeing negative rewards at specific episodes. First of all, the continuous state space is very large and the DQN model may not be able to learn every possible state. Secondly, the changing landing terrain and possibly different initialization could make the agent take the wrong action and either fly out of the frame or crash to the ground. Another explanation could be attributed to the stopping criteria. As the stopping condition is met as long as the average score of last 100 episodes exceeding +200, if we make such condition more stringent like "there is no negative reward for all past 100 episodes", the training agent might take more episodes to learn and eventually be able to prevent possible crash.

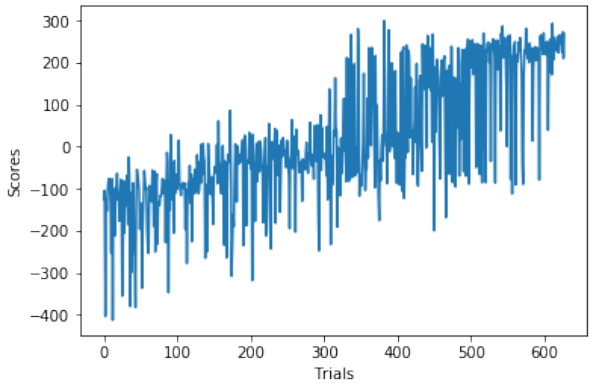

Fig. 2. Score per Training Trial

Figure 4 illustrates the reward at each testing episode using the trained agent. The mean reward of these 100 episodes is +209. Though above the success condition, it demonstrates the similar issue discussed above. Though the agent performs really well on average, in some rare occasion, it can still end with a negative reward. This might be improved through the modification of stopping criteria to eliminate negative reward.

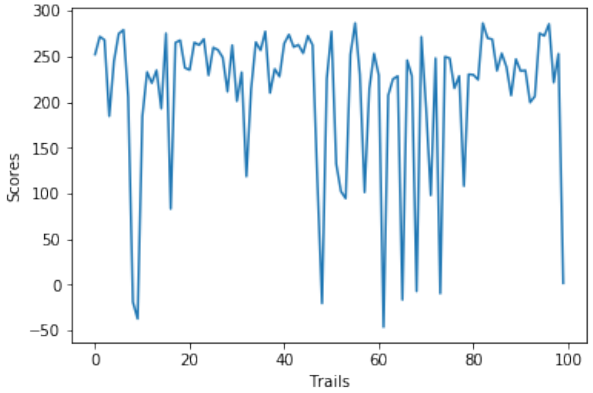

Fig. 3. Score per Test Trial with the Trained Agent

B. Hyperparameters

Hyperparameters discount rate γ, learning rate lr, and exploration-exploitation control parameter $epsilon\_decay$ have been studied to assess their influence on the convergence speed and learning outcomes. The experiments are conducted with the maximum episode of 1000 with a maximum step of each episode as 500.

Discount Factor γ in general determines how future reward impacts the decision of action at the current state. A larger discount factor treats rewards in distance future closer to the immediate reward. A smaller discount factor favors the immediate reward and tends to maximize short-term gain. Figure 5 illustrates such effects. As different γ values start the training with very similar reward values, smaller gamma values creates more divergence and are slower to achieve successful landing (large positive reward). The purple line has γ value of 0.99 and performs the best.

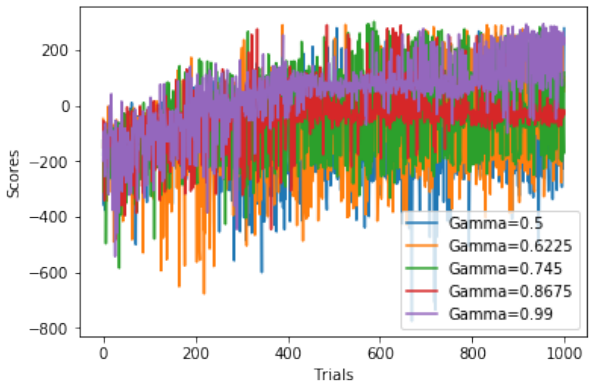

Fig. 4. Scores per Training Trial Given Different Discount Factor γ

Learning rate lr decides how much the weights in each pass get updated to the direction that minimize the current

$\overline{VE}$ the most. Figure 6 illustrate the outcome of adopting different learning rates ranging from $5e^{-5}$ to 0.5. A smaller learning rate makes the learning inefficient and difficult to converge. If it is too large, it may easily miss the local optimum in the SGD and could possibly behaves like taking random actions. Balanced cases like $5e^{-5}$ and $5e^{-4}$ achieve the most satisfactory learning outcome.

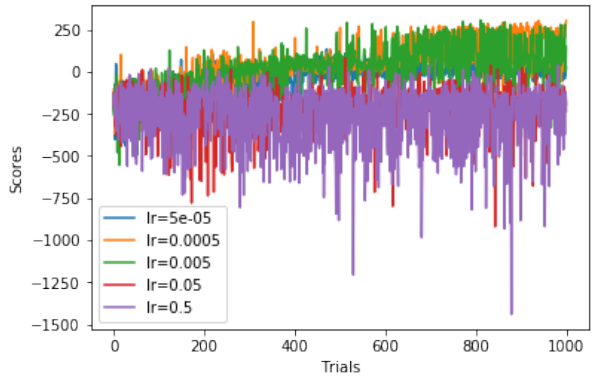

Fig. 5. Scores per Training Trial Given Different Learning Rate lr

Exploration-exploitation control parameter $epsilon\_decay$ dictates how fast the agent switches from exploration to exploitation. Figure 7 illustrates how fast the epsilon decays by the number of trials. A smaller $epsilon\_decay = 0.8$ drops $epsilon$ close to zero after very few episode. This means the agent quickly switch to exploitation with very limited knowledge about the environment. When $epsilon\_decay = 1$, it means the agent is always performing exploration and never utilize the learned experience. The result can be seen from the score outputs. A recommended $epsilon\_decay = 0.995$ (red curve) focus on exploration and learning in the early episodes and slowly switch to exploitation in the later stage, hence generates the best learning results.

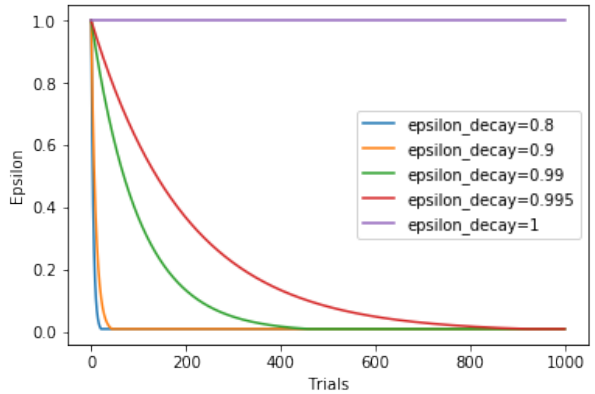

Fig. 6. Epsilon Per Training Trial Given Different Epsilon Decay

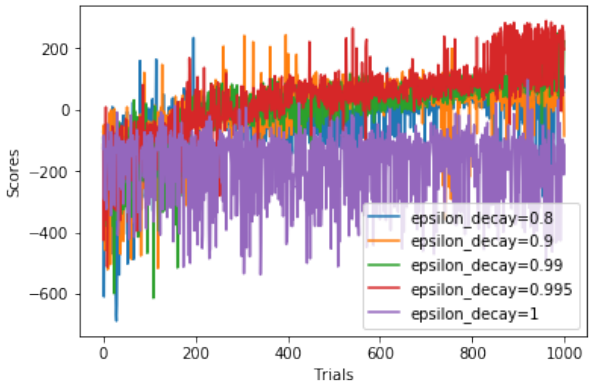

Fig. 7. Scores per Training Trial Given Different Epsilon Decay

VI. CONCLUSIONS

The author of this report has walked through the theoretical foundations including Q-learning and value-function approximation, implementations of Deep Q-Network (DQN) algorithm in the lunar lander problem. The author successfully trained the agent and met the solving condition. In this report, considerations related to network size, replay memory and batch processing have also been discussed. Last but not least, the trained agent has been evaluated along with the analysis on hyperparameter selections and their implications on learning. If given with more time and bandwidth, the author plans to explore and implement other advanced reinforcement learning algorithms like Dueling DQN and policy gradient in solving the lunar lander problem.

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] Li, Y. (2019). Deep Reinforcement Learning. Retrieved from https://arxiv.org/abs/1810.06339

[4] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2019). Deep Reinforcement Learning that Matters. Retrieved from https://arxiv.org/abs/1709.06560

[5] Fu, J., Kumar, A., Soh, M., & Levine, S. (2019). Diagnosing Bottlenecks in Deep Q-learning Algorithms. Retrieved from https://arxiv.org/abs/1902.10250

[6] udacity/deep-reinforcement-learning. (2019). Retrieved from https://github.com/udacity/deep-reinforcement-learning