Reinforcement Learning and Decision Making
Project 2: Lunar Lander

Hanxiao Wu, hwu429@gatech.edu

Last Git Hash: e90b0982e6526c27a2814dd4d693d9ec4008ec79

1 INTRODUCTION

In this paper, a RL agent is trained using Deep Q-Network (DQN) to solve the Lunar Lander problem. The paper will discuss the background knowledge from Q learning to DQN, experiment implementation details and the performance of the trained agent.

2 METHODOLOGY

2.1 From Tabular Q learning to function approximation

In reinforcement learning, the most well-known model-free method is Q learning. Tabular Q learning essentially uses a Q value lookup table for every state and every action Q(s,a). However, it has limitations when facing a problem with large state space, especially when the state space is continuous, as it could be impractical to find the large matrix iteratively. In order to solve such problems, the key is generalization: the agent needs to estimate the value of a state it hasn't experienced before, based on similar states it experienced. It essentially approximates a Q-function, which is the same idea as function approximation in supervised learning:

$$Q(S,\ a)\ =\ f(S,\ a,\ w)$$

where w stands for the function parameters, S is a set of state features that represents a state.

Finding the fitted value function is essentially minimizing the loss function:

$$L(w)\ =[Q(S,\ a)-f\ (S,a,w)]^2$$

In order to find the minimum of loss function, stochastic gradient descent is implemented:

$$\Delta w\ =\ -\alpha\ [Q(S,\ a)-f\ (S,a,w)]\ \ \nabla_w f(S,\ a,\ w)$$

2.2 Deep Q-Network

Deep Q-Network (DQN) uses a neural network to approximate Q-function. Neural network is preferred due to its differentiability and its ability to approximate nonlinear functions. In the learning process, it updates the model parameters with the goal of minimizing the loss function between actual Q-function and current prediction. In reinforcement learning, the actual Q-function is unknown, so we'll update the parameters towards a proxy target, which is the sum of the reward and the next best outcome prediction.

$$L(w)\ =\ [r+\gamma max\ Q(S',\ a';\ w^{-})\ -\ Q(s,a;w)]^2$$

One thing to note that the second part of the target is not the prediction from the estimated network itself, instead it is from a target network that uses an older set of weights of the

estimated network. This is designed for stabilizing Q-learning. The separate target network reduces the correlation between the target and current prediction and prevents divergence.

Another idea to stabilize the process is to use experience replay and learn from a sampled batch. Instead of online, it saves experience tuples to a replay buffer and randomly samples a batch to learn. There are two advantages: one is to make good use of the past experience, another benefit is that the random sampling process reduces the correlation among experience and makes data distribution more stationary. Below shows pseudo code of the algorithm:

Algorithm 1: Deep Q-Network with Experience Replay and Target Network

---

Initialize replay buffer D with size N
Initialize evaluation model with random parameter w
Initialize target model as evaluation model
For i in range(max_episodes):
    Initialize environment and state
    For step in range(max_steps):
        According to ε-greedy policy, choose the optimal action with probability 1 - ε otherwise choose a random action
        Execute the action and observe the experience tuple $(S,\ a,\ r,\ S',\ done)$
        Save experience tuple to replay buffer D
        $S \leftarrow S'$
        Every C steps:
            Randomly sample a batch of experience tuples from D
            Calculate Q-learning targets using target network: $r + \gamma max\ Q(S',\ a';\ w^{-})$
            Calculate Q-network predictions using evaluation network: $Q(S,\ a;\ w)$
            Update evaluation network parameters to minimize loss between Q-network prediction and Q-learning targets
            Soft update target network with rate τ : $w^{-} \leftarrow (1-\tau)w^{-} + \tau w$
    End for
    $\varepsilon \leftarrow \varepsilon$ * epsilon decay
End for

2.3 Lunar Lander

The Lunar Lander problem is based on the LunarLander-v2 environment in OpenAI gym. It has an 8-dimensional state space, with six continuous state variables (vertical and horizontal coordinates, vertical and horizontal speeds, angle and angle speed) and two discrete ones (legs contact). Those state features are sufficient to describe any state the lunar lander may be in. There are four discrete actions: do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine.

The goal is to move from the top of the screen to the landing pad. The base reward of doing so is 100 to 140 points. An episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 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.

3 EXPERIMENT IMPLEMENTATION

3.1 Implementation

Since the Lunar Lander environment has a large state space with continuous state variables such as speeds, function approximation method is preferred. Specifically, we construct a neural network with an input layer, an output layer and a hidden layer with size of 64x64.

3.2 Agent Performance

The Figure 1 shows the reward at each training episode in the agent training process. On average, the agent reaches higher scores as more episodes are learned. After 700 episodes, the agent achieves a score of 200 points on rolling average and the problem is considered solved. There are still rare accidents where the scores are below 200, one hypothesis is that the agent still has a low chance to take random actions that could lead to loss of points; another hypothesis is that the agent encounters situations it hasn't learned properly.

Figure 1 — Reward at each training episode.

Figure 2 — Reward per episode using the trained agent.

The Figure 2 shows the reward of testing episodes using the trained agent. In the testing environment, the agent only exploits the optimal action at each state without random exploration. It's able to achieve an average reward of 252.8. Here is a short video of the trained agent performance in the testing environment.

3.3 Hyperparameters

During the training process, there is a list of hyperparameters worth exploring. Below shows some hyperparameters and their effects on the agent training process.

- **Epsilon, Epsilon decay and Epsilon minimum:** These epsilon-related parameters control the exploration. The higher the epsilon is, the agent is more often to execute a random action to explore. In a complicated environment, the epsilon should be large at the beginning in order to explore the world, and as more knowledge it learns, the epsilon will decay over time to exploit the optimal policy it learned. Failing to do so will result in the agent exploiting the little experience it knows and stucked at a local optima.
- **Learning rate (Alpha):** The learning rate determines how much the parameters of the neural network should be updated towards the Q-learning target from the target network.

- **Discount rate (Gamma):** it determines how the agent values the reward later on. It's generally a good practice to have a higher gamma to value the long term reward.
- **Replay buffer size:** this is how many experience tuples the replay buffer maintains.
- **Batch size:** this is how many experience tuples the agent samples and learns every time. A larger batch size enables the weight updating process to learn more efficiently but it risks wose performance if the samples in the batch are highly correlated.
- **Evaluation network update interval (C)**
- **Target network soft update rate (τ)**

Below are three experiments about epsilon decay, discount rate and learning rate.

In Figure 3, the epsilon decay is studied. It's essentially an exploration-exploitation tradeoff lever. The agent with a decay rate of 0.99 performs the best. With a slow epsilon decay schedule (decay rate of 0.998), the agent still explores frequently at later stages even though a good policy is learned. The reward increases slowly and is hardly considered solved within 800 episodes. On the other hand, for the agent with a faster decay schedule (decay rate of 0.9), the reward is more fluctuated and the average score is lower compared to the reward with 0.99 decay rate: it is more often to achieve extreme low and high values. The low score happens when it encounters a situation that hasn't been explored yet, and the high score happens when it's exploiting the optimal policy.

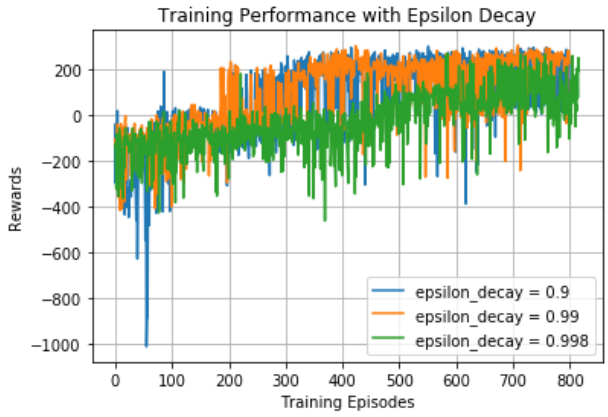

Figure 3 — Training with different episode decay parameters.

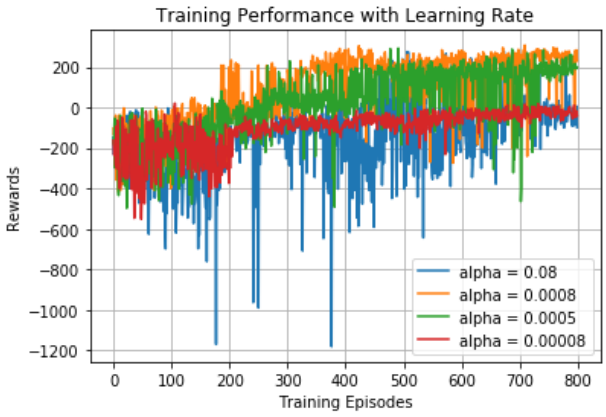

Figure 4 — Training with different learning rate parameters.

In Figure 4, the learning rate is studied.A learning rate that is too small is less efficient to update weight and take more time to converge but is more likely to converge if enough episodes are learned. A learning rate that is too large is fast to train but may risk overshooting and diverging. In the lunar lander problem, an alpha of 0.0008 performs the best.

In Figure 5, the discount rate gamma is studied. In the lunar lander problem, firing engines will receive -0.3 to -0.03 reward and landing at the right place receives +100 reward. It's important for the agent to value the 100 reward more than the small negative reward in order to solve it. For example, assuming the agent needs 100 steps to land at the right place and receive the +100 reward, with a gamma of 0.99, the 100 reward worth 36.6, still pretty large, while with a gamma of 0.8, the 100 reward only worth 2e-8 then, which is almost zero. The small gamma will lead the agent to minimize the negative rewards at the beginning instead of finding the large

positive reward at the end. As Figure 4 reveals, the agent with gamma of 0.8 converges to an average score just below zero, while the agent with 0.99 achieves a much higher average score.

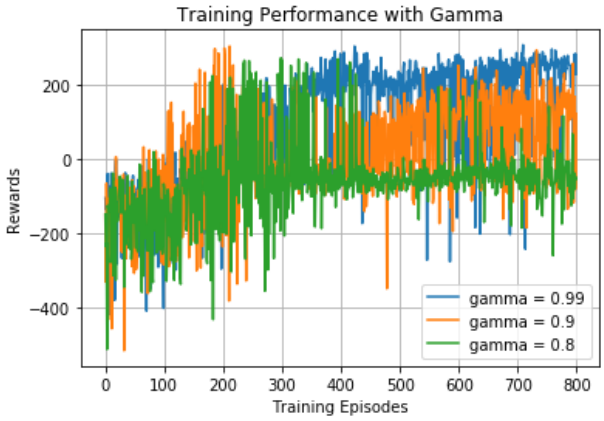

Figure 5 — Training with different gamma parameters.

After experiments, here is the list of hyperparameters used: epsilon = 1, epsilon decay = 0.99, epsilon minimum = 0.01, gamma = 0.99, learning rate = 0.0008, replay buffer size = 200000, batch size = 64, evaluation network update interval = 4, target network soft update rate = 0.001.

4 PITFALLS, LEARNING, AND FUTURE WORK

4.1 Pitfalls and learning

One pitall is that I didn't initialize the epsilon value to 1 when restarting an experiment, and the following experiments have very small epsilon to start with and get bad performance. I figured it out after printing out the parameters and performance to monitor the changes.

Another pitfall is about the weight updating. Everytime only the error of one state-action pair is calculated while the network output has 4 actions values for one state, and I struggled to understand how to feed the information to the network properly. It is fixed with a hint from TA: setting the error for the actions we aren't updating to be 0 as we don't have any information on which direction to move them.

4.2 Future Work

If more time is allowed, I would try to use huber loss as the loss function and compare its performance with L2 Loss. Another interesting thing is to see whether a sampling method that prefers high reward outcomes leads to more efficient learning compared to random sampling.

5 REFERENCES

Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2nd Ed. MIT press, 2020. url: http://incompleteideas.net/book/the-book-2nd.html.

Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.

Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.

Mnih V. DQN Lecture. https://drive.google.com/file/d/0BxXI_RttTZAhVUhpbDhiSUFFNjg/view

Silver D. Reinforcement Learning Lectures. https://www.davidsilver.uk/teaching/