CS7642 Project 2: Lunar Lander

Seungwon Kim
Incheon International Airport Corporation
Georgia Institute of Technology
skim3222@gatech.edu

Abstract

In this paper, I implemented four different algorithms - Deep Q-Networks(DQN), Double Deep Q-Networks(DDQN), and DQN with linear approximation model and shallow neural networks to solve the Lunar Lander problem [1, 2, 3, 4]. I demonstrate that DQN, DDQN, and DQN with even simple two-layer network are able to solve the Lunar Lander problem and make the agent successfully land on the moon.

1 Introduction

Lunar Lander is a video game developed by Atari.Inc and the environment of Lunar Lander is implemented in OpenAI gym library [3, 5]. The goal of this game is to land on the moon safely by deliberately firing the engine of the lander to adjust the trajectory properly. The agent gets a higher score, i.e. reward, if it spends less gas and safely lands on the landing pad between the two flags on the ground.

In the gym environment, there are four actions in the action space: do nothing, firing the left engine, fire the main engine, and fire the right engine. The state is in 8-dimensional space, which consists of 8 components $(x, y, vx, vy, \Theta, v\Theta$, left leg, right leg$)$. x and y are x and y coordinates of the position of the lander, vx and vy are the velocity towards each coordinate. Θ is the angle of the lander and $v\Theta$ is the angular velocity. Left-leg and right-leg are the binary values that indicate whether the leg of the lander touches the ground. The agent achieves from 100 to 140 score with respect to the landing placement and it receives additional 100 or -100 reward depending on whether it lands on the moon safely. Firing engine gives the agent -0.3 reward and if each leg of the lander touches the ground, each incurs 10 reward respectively. If the average rewawrd of 100 previous trials exceeds 200, the problem is considered to be solved [3].

In this paper, I implemented variants of Q Learning algorithms, including Deep Q-Networks(DQN), Double Deep Q-Networks(DDQN), and DQN with linear approximation model and shallow neural networks, to make the lander successfully land on the moon in OpenAI gym environment [1, 2, 3, 4]. Section 2 briefly recaps DQN and DDQN algorithms and section 3 shows the results of experiments and compare the performance of each algorithm with each other as well as the effect of different hyperparameters such as learning rate, ϵ, and others.

2 Background

In Lunar Lander environment, which the state is in continuous space, it is necessary to approximate value function since an infinite number of states makes it hard to estimate value function via Q-learning update. DQN adapts deep neural networks to approximate state-action value for high dimensional state space and makes use of experience replay and fixed target networks to stabilize Q-learning update [2]. At each step, the agent explores the environment with ϵ - greedy strategy. The experience from exploration, which consists of 4 components (state, action, reward, and next state), is stored in replay memory and training set with batch size is retrieved from the replay memory

to estimate the parameters of the state-action value function. The parameters can be obtained by performing gradient descent on loss function, which is expressed in equation (1). Note that there are two kinds of parameter: Θ for the online network and $\overline{\theta}$ for the fixed network.

$$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2 \tag{1}$$

On the other hand, DDQN is an advanced version of DQN and it is able to solve the upward bias of DQN, which is due to maximization step in euqation (1), resulting in more stable update and learning process [4]. Unlike equation (1), DDQN selects the greedy action a' by using the online parameter Θ to estimate target value as in equation (2).

$$(R_{t+1} + \gamma_{t+1} q_{\overline{\theta}}(S_{t+1}, \arg\max_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2 \tag{2}$$

In section 3, above two algorithms as well as DQN with two-layer network, linear approximation are used for the experiments in Lunar Lander environment.

3 Experiments

In this section, I explore Lunar Lander using DQN algorithm with two hidden layers and show the results of experiments. During the experiments, I encountered a lot of pitfalls. First of all, there is no clear guidance for selecting the value of hyperparameters including learning rate, ϵ or ϵ decay, the memory size, the length of interval for resetting target networks, and even the architecture of neural networks. I almost failed to make the agent even get a positive reward during the training process at the beginning because of the randomly chosen value of hyperparameters. I realized that performing strict greed search on all combinations of these hyperparameters with infinite range results in enormous computational costs and impossible to finish the project within a limited time. As such, I selected arbitrary value and range of hyperparameters which are similar to those used for training Atari games in DQN paper but some values such as the size of experience replay or number of units in neural networks are much smaller, since Lunar Lander seems to be more simple game than classic Atari 2600 games [2]. The range of each hyperparameter is presented in each subsection in detail.

Second, the stopping criterion of the training process seems to be ambiguous. Since the problem is considered to be solved when the average reward of the previous 100 episodes during the training process is over 200, one can set the stopping criterion as perform training until the average reward exceeds 200. However, if we set that condition as the only stopping criterion, the training process runs forever when the reward per episode doesn't converge over 200. To shorten the running time and encourage the agent to learn efficiently, I limited the number of episodes with 2000 and exponentially decreased the value of ϵ for each step instead of linearly decaying ϵ. In the implementation, while loop makes the training process terminate if one of the following three conditions is satisfied: the number of episodes reaches 2000 times, the value of ϵ is less than 0.01, and the rolling average of 100 previous episodes exceeds 200.

Third, although the rolling average exceeds 200, the final parameter after the entire training process does not always give good performance in 100 trials because the agent selects the action with ϵ - greedy in training process but always acts greedily in the evaluation process. Thus, the experiments were performed with different hyperparameters such as ϵ, ϵ decay, and even the number of layers in neural networks to collect more candidates for evaluation. The best model which performs the best in 100 trials is presented in the following section.

3.1 DQN performance

In this section, I applied DQN to train the agent on the Lunar Lander environment. Experiments were performed with almost the same DQN algorithm used for training Atari 2600 [2]. The difference that I made during implementation is that the values of hyperparameter and the architecture of the neural network. The values of hyperparameters such as memory size, target reset interval, and the number of samples required to start gradient descent became smaller and I implemented three fully connected layers with two hidden layers which have 64 hidden units respectively, instead of using both Convolution Neural Networks(CNN) and fully connected layers. CNN seemed to be inappropriate for Lunar Lander since the number of input space is only 8 and the components of

input are not the image pixel. The experiments were performed with 30 different combinations ϵ and ϵ decay: 6 number of different ϵ (0.9, 0.8, 0.7, 0.6, 0.5, 0.4) and 5 number of different ϵ decay (1e-4, 3e-5, 1e-5, 3e-6, 1e-6; Note that 1e-4 indicates 0.0001). The value of other hyperparameters was fixed. Learning rate for optimization was fixed with 1e-3, which is the default value of tensorflow's Adam optimizer. Figure 1 shows the reward per episode in the training process (Left) and evaluation of trained agent (Right) when ϵ and ϵ decay is at (0.4, 1e-5).

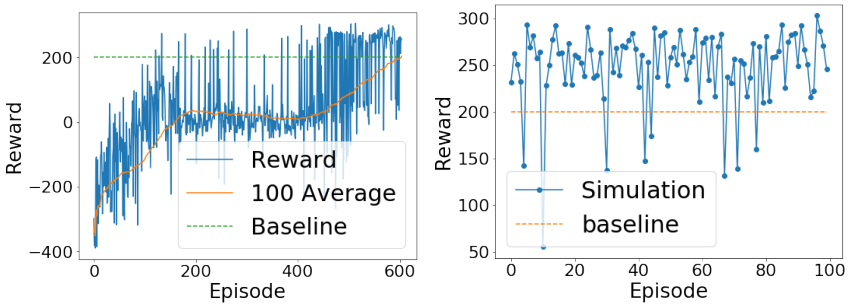

Figure 1: Best performance occurred with ϵ at 0.4 and ϵ decay rate at 1e-5. The dashed line is presented as a baseline with the value at 200. Other hyperparameters were fixed: learning rate at 1e-3, gamma at 1, target rest at 1000, memory size at 1000, mini-batch size at 32, gradient clipping at 5, number of hidden units at 64, train starts after 500 number of trainsitions are stored in the memory, Adam optimizer, Huber loss.

In figure 1 (Left), average reward oscillates at the beginning but gradually increases and reaches over 200 around 600 number of episodes. Since the starting value of ϵ is relatively small (0.4), it caused less exploration and makes 200 average reward with a small number of episodes. The agent learned a somewhat good trajectory to land on the moon within 600 episodes. The performance of the trained agent (Right) was evaluated with the weight parameter after the training process (mean: 248, std: 40). In general, the trained agent easily outperforms the baseline reward. However, it sometimes scored below the baseline about 10 times. It seems that other hyperparameters such as the interval of resetting target need to be tuned to achieve the perfect performance.

3.2 Effect of hyperparameters

Here I present the effect of three different hyperparameters: ϵ decay, ϵ, the learning rate of Adam optimizer. Since these hyperparameters directly effect on the running time of the training process, finding good ϵ or learning rate can reduce computational costs caused by deep neural networks. Sufficiently small ϵ and large learning rate are expected to lead to fast convergence. The average reward of 100 previous episodes is presented as a standard to compare the performance of each model. First, figure 2 shows the rolling average reward during training process with respect to different ϵ decay (Left) and ϵ (Right).

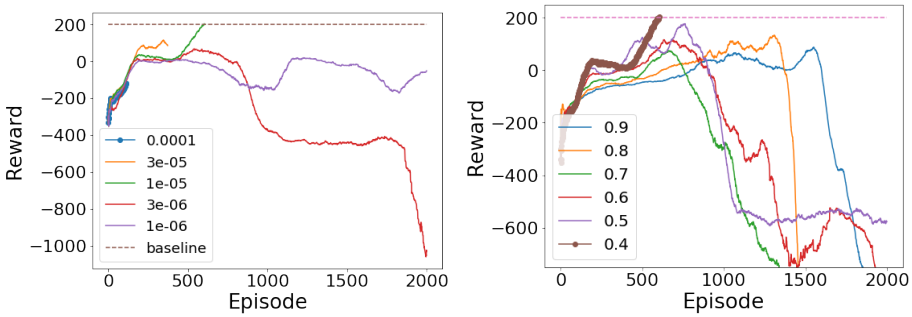

Figure 2: Rolling average reward of previous 100 episodes during training the agent. The dashed line represents a baseline with a reward at 200. For ϵ deacy (Left), ϵ is fixed at 0.4. For ϵ (Right), ϵ deacy is fixed at 1e-5. All other hyperparameters are fixed with the same value provided in figure 1. Bold lines are presenteed to make some hidden lines legible.

In figure 2 (Left), the problem is solved, i.e. the average reward exceeds 200, only when ϵ decay is intermediate value, which is 1e-05 (green). Otherwise, the average reward is far less than those of the well-trained agent. This is because too large ϵ decay (0.0001, 3e-5) makes the agent not be able to explore enough times and terminates the training process too early. On the other hand, too small ϵ decay (3e-6, 1e-6) results in being stuck in local optimum or continuously decreases the average reward by executing random behavior.

Figure 2 (Right) also shows the rolling average reward during the training process with respect to different the ϵ. The average reward exceeds 200 only when the ϵ is at 0.4. It takes around 600 number of episodes to solve the problem, whereas, with other ϵ, the agent failed to solve the problem even if it experiences the 2000 number of episodes. This is because too large ϵ can cause random behavior too many times, resulting in inefficient learning. The results show that a small value of ϵ makes the agent learn efficiently and faster than other cases.

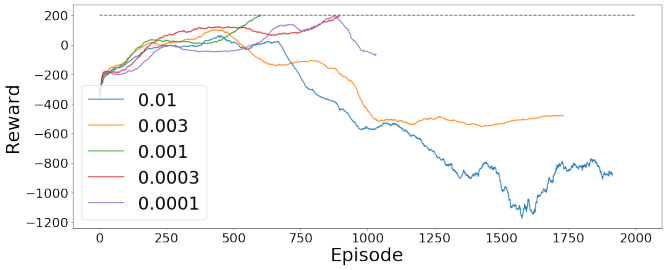

Figure 3: For this experiment, ϵ is fixed at 0.4 and ϵ decay is fixed at 1e-6. Other hyperparameters are same as those in section 3.1. The dashed line represents 200 reward.

Lastly, figure 3 shows the average reward with respect to the different learning rate which is used for optimization. The agent with learning rate at 1e-3 or 3e-4 is able to obtain more than average 200 reward and with 1e-4, it almost reaches 200 reward but fails to outperform the baseline. For learning rate 1e-2 and 3e-3, the average reward is far less than other cases. The results show that the ideal learning rate is between 1e-3 and 1e-4. If the learning rate is bigger than 1e-3, stochastic gradient descent tends to oscillate around optimum and can't make the weight parameters converge. If the value is smaller than 1e-4, it makes the training process slower and causes the weight parameters stuck in the local optimum. Figure 2 and 3 ensure that the performance of DQN is sensitive with the value of ϵ, ϵ decay, and the learning rate.

3.3 Algorithm comparison

In this section, I compare the performance of four different models: DQN with linear approximation model that has linear combination of features, two-layer neural network which has one hidden layer, DQN that is same as the previous section, and DDQN. All hyperparameters are the same except for the model architecture. Hidden units are fixed at 64 and for DQN and DDQN, only the update rule of state-action value is different. As shown in equation (1) and (2), DDQN uses the online network to select the action for estimating target value. The average rewards of previous 100 episodes during the training process are presented in figure 4.

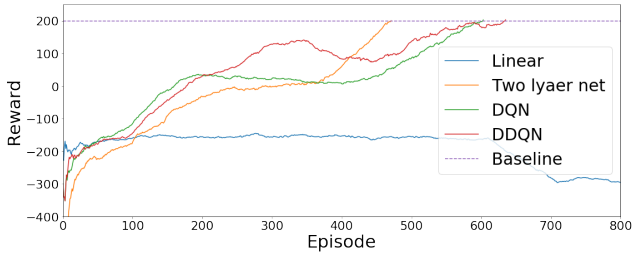

Figure 4: ϵ is fixed at 0.4 and ϵ decay is fixed at 1e-5 for all models. Other hyperparameters are the same as those in section 3.1. The dashed line is provided as a baseline with 200 reward. Note that x-axis is cropped with 800 since there is no big difference in the performance of linear approximation.

Except for the linear approximation model, DDQN, DQN (with the three-layer net), and even DQN with the simple two-layer net are able to reach over 200 average reward around 500 number of episodes. Linear approximation model performs worst with far less average reward under -200. The result shows that simple non-linear approximation with one hidden layer is enough to solve the Lunar Lander game with appropriate value of hyperparameters. This is because the state is in 8-dimensional space, which is far less than input space in Atari 2600 [2]. If the environments have high-dimensional state space or the value of other hyperparameters such as memory size or target reset interval is different, the performance of each algorithm would be different.

3.4 Further discussion

From section 3.1 to 3.3, I have explored four different versions of DQN algorithm with different hyperparameters. However, I didn't investigate the effect of other hyperparameters such as the interval of resetting target or the size of replay memory. Also, I have not considered the standard deviation of the performance of the trained agent. Since the goal of the problem is to make the average reward over 200 in the previous 100 episodes, I immediately stopped training the agent if the criterion is satisfied. This might effect on the deviation of the trained agent's performance as in figure 1 (Right). If I had more time, I would like to change the stopping criterion and run the agent over 2000 episodes to make him perfect.

Second, I would like to apply feature engineering to the input state of DQN because the training process can be viewed as supervised learning, which maps the input of state and action to the output target. Feature reconstruction algorithms such as principal component analysis, independent component analysis, or feature engineering using decision tree might boost the performance of the agent.

Lastly, there are more advanced versions of DQN such as Rainbow DQN [1]. These are expected to bring more stable learning and fast convergence. After finishing the course, I would like to implement all variants of DQN and compare the performance with each other. It would be interesting to know the minimum number of episodes to solve the Lunar Lander problem.

4 Conclusion

In this paper, I have explored four different DQN algorithms with different hyperparameters. DDQN, DQN (with three-layer net), and DQN with the shallow two-layer network are able to solve the Lunar Lander game with over 200 average reward of previous 100 episodes given the same value of hyperparameters.

References

[1] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[3] OpenAI. Lunarlander-v2, (n.d.). Retrived from `https://gym.openai.com/envs/LunarLander-v2/`.

[4] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[5] Wikipedia contributors. Lunar lander (1979 video game) — Wikipedia, the free encyclopedia, 2019. Retrived from `https://en.wikipedia.org/w/index.php?title=Lunar_Lander_(1979_video_game)&oldid=891567216`.