Project 2 Report

Yihuan Su

Git hash: 36c56b40385364dd78159098f6e2f05a15401e93

I Introduction

The goal of this project is to solve the Lunar Lander problem(https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py). The environment has a 8-dimensional state space, of six are continuous state variables, and 4 discreet actions. Among all the methods can be used to solve the problem, I picked Deep Q-Network (Minh et al 2015). In this report I will first describe the environment we are dealing with. Then I will present a introduction of Deep Q-Network. Results of the experiments will be discussed afterwards.

II Lunar Lander

The Lunar Lander environment has a 8-dimensional state space, of six are continuous state variables and two descrete state variables. The six continous states are: horizonal position, vertical position, horizonal verlocity, vertical verlocity, angle, and angular speed of the lander. The two discreet binary states are indicators for if the left leg or right leg of lander is touch the ground. There is a landing pad which is always positioned at coordinates (0,0). Figure 1 is a rendered image of the environment.

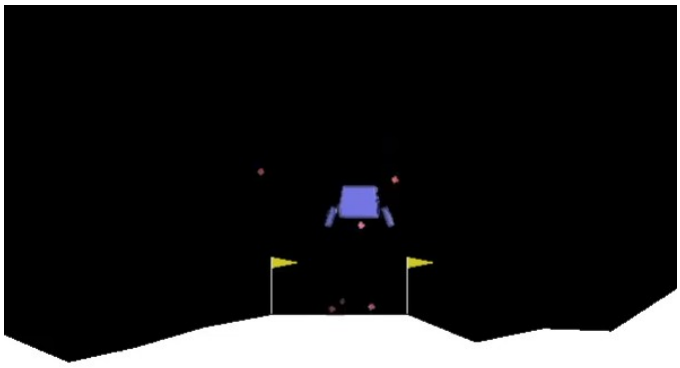

Figure 1

An episode is completed if the lander lands or crashes, which result in +100 or -100 reward respectively. The lander can land outside the landing pad. If the lander moves closer to the landing pad, it will be rewarded. If it moves away, it will be penalized. If the velocity increasess, it will penalized. If the lander's angle increases, it will also be penalized. If There is also a +10 points reward for each leg ground contact. There four available actions: do nothing, fire the left orientation engine, fire the main engine, and fire the right orientation engine. Firing the main engine results in a -0.3 point penalty and firing the orientation engines cause a -0.03 point penalty. There is not limit to the fuel. For this project, the problem is considered solved if the average score over 100 consecutive runs is at least 200.

III Solution

III.A Q-learning

According to Sutton & Barto (2018), Q-learning is an off-policy TD control algorithm. It is defined by

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

where S_t is the state at time step t, A_t is the selected action at time step t, R_{t+1} is the reward received at time step t+1, α is the learning rate, and γ is the discount rate. The learned value fuction Q approximates the optimal value function. Unlike on-policy algorithm like Sarsa, the policy being followed doesn't affect the update of Q function. The policy only determines which state-action pairs are explored and updated.

Q-learning is an etremely simple yet very effective alorithm. Sutton & Barto (2018) mentioned Q funciton is guaranteed to converge under the assumption that all state-action paris are updated continuously. However, Q-learning is ill-suited for the problem at hand because Q-learning is a model free method and it learns by updating the tabular Q table. Therefore, it requires states and actions to be discreet, whereas continous state variables exist in the Lunar Lander environment. It is still possible to solve the problem using Q-learning by discreetizing continuous state variables into discreet states. But given the dimension of the state space and the complexity of the problem, it can easily results in a very large number of unique states and making the computation very expensive.

III.B Deep Q-Network

Function approximation appears to be the more suitable approach to solve problems with continuous state values. Instead of learning through updating a Q table, the agent learns by approximating the action-value funciton $\hat{q}(s, a, w) \approx q_*(s, a)\ where\ w \in \mathbb{R}^d$ is a weight vector (Sutton & Barto 2018). There are different function approximation methods that can be use to solve the Lunar Lander problem, I selected the Deep Q-Network (DQN) which is a novel method proposed by

Minh et al (2015). DQN is based on Q-learning method. It relies on a artificial nueral networks to approximate the action value function. Minh et al (2015) used a deep convolutional neural network. The agent they developed was able to perform at a level similar to professional human players across 49 Atari 2600 games. In comparison to the Atari games, Lunar Lander is a much less complex problem, therefore a feedworward artificial neural network should be sufficient for the task.

Artificial neural networks (ANNs) are widely adopted for nonlinear function approximation. Even though it have a very long history, not until recently it became extremely popular for computer vision and natural language processing thanks to development in GPU computational capability. Figure 2 shows the structure of a generic feedforward ANN. It's a network formed by interconnected units, which are often referred as neurons. The ANN shown here has four layers, the first column of four units is the input layer. The two columns in the middle are known as hidden layers. The two units at the end are the output units and they form output layer. Each unit are often a semi-linear unit which means it computes a sum of the inputs with respect to a set of weights. A nonlinear function often referred as activation function is applied afterward to produce the unit's output. Sigmoid function $f(x) = 1/(1 + e^{-x})$ and rectified linear unit funciton $f(x) = max(0, x)$ are among the most commonly used activation functions. Multi-layer feedforward neural network can approximate very complex nonlinear functions. The activation function is an essential part of it. Without nonlinear activation function, the network can be represented by one or more linear funcitons depending out the number of output units (Sutton & Barto 2018).

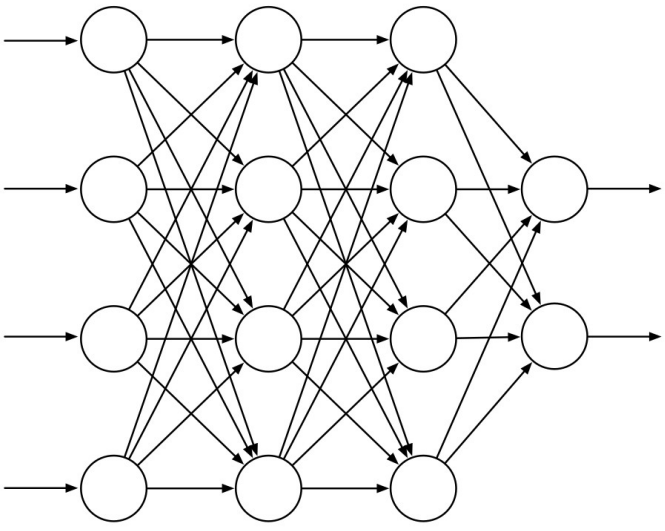

Figure 2

Training a ANN is usually accomplished through backpropagation algorithm. Without going into too much detail, the backpropagation algorithm is consisted of a forward pass and a backward pass through the network. A forward pass computes the activation of the each unit given across the network with given inputs respects to the weights and evluates the error. It's followed by a backward pass which computes a partial derivative for each weight. Then each weight is updated according to a learning rate with the goal to minimize the error (Sutton & Barto 2018).

Standard gradient descent has been one of the most popular optimization algorithm. Given the complexity of neural networks, variants of gradient descent algorithms have emerged in recent years witht the goal to make training neural networks more efficient and more effective. Each algorithm have it's own strength. It's out of the scope of this report to delve in the details of various optimization algorithms. Just to name a few popular algorithms, there are RMSprop, Adadelta, Adaptive Moment Estimation (Adam), Stochastic Gradient Descent (SGD) (Ruder, 2017).

One key feature of the DQN is that it utilizes what's referred as experience replay. The experience of the agent at each time step is saved into a replay memory over many episodes. During learning, a minibatch which is a random sample of the experience stored is used at each Q-learning update. Because of the strong correlations between the samples, learning from sequential samples directly can be inefficient. These correlations are broken by randomizing the samples and consequently decrease the variances of the updates (Minh et al 2013).

Based on algorithm developed by Minh et al (2013), I impletemented my own solution of Lunar Lander. During training, the action is selected based on a ε-greedy policy. Similar to what Minh et al (2013) did, the ε value gradually decreses according to a given decay rate. Below is the pseudo code of my impletementation.

<u>Pseudo Code – DQN for Lunar Lander</u>
Input: n_episodes, α, γ, ε, epsilon_decay
Return: action-value function Q (ANN)
Init action-value function Q (ANN) with random weights
Init memory
For each episode
    Init state S
    For each step
        Select a random action A with probability ε
        otherwise select action A according to
        action-value function Q
        Take action A, oberve R, S'
        Append (S, A, R, S') to memory
        Take a minibatch of observations (S_i, A_i, R_i, S'_i)
        from memory
        Set target $Y_i = R_i$ if S'_i is terminal otherwise
        $Y_i=R_i+\gamma*max_aQ(S'_i,a)$
        Perform a update of Q using the minibatch
        S=S'
    ε=ε*epsilon_decay

IV Training and Results

IV.A Solving Lunar Lander

Using the algorithm impletemented based on DQN, I trained an agent to solve the Lunar Lander problem. Due to limited time and computational resources. It's not feesible to perform an exhaustive hyper-parameter selection. Instead, values of hyper-parameters are selected based on intuition and some level of experimentation. The agent are trained with hyper-parameters shown below:

discount rate γ=0.99
learning rate α=0.0005
hidden layer size (32, 64, 16)
optimizer='Adam'
minibatch size=64
ε=1.0
epsilon decay=0.995
number of episodes=500

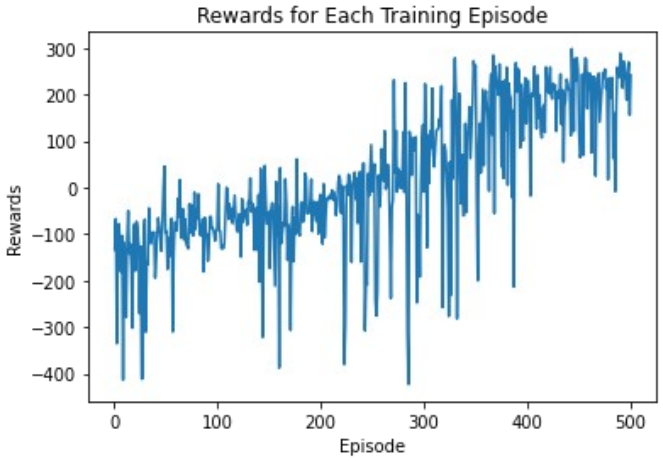

Figure 3

Figure 3 shows the rewards for each training episode. As the number of episode increases, the agent learns from the past experience and gets better at maxmizing rewards, the total rewards increses.. It's also because of the ε value decays over training episodes, so the agent is less like to to take random action as the value of ε decreases.

Figure 4 shows the performance of the trained agent over 100 consecutive episodes. It achieved an average reward of 242.68. Due to the randomness of the environment, occasionally the total reward can be as low as close to 0 for some episodes, but most of the time the time the reward is above 200, and we've yet to see any negative reward. It's still possible to further improve the agent by training with more episodes.

IV.BExploring Hyper-parameters

To understand how different values for hyper-parameters can affect the performance of the agent, I select different values for ε, learning rate α and γ to experiment. Training an agent with 500 episode can take up to 6 hours with my impletemented algorithm. Because of time limitation, for each one of the three hyper-parameters, I only experimented 3 different values.

Figure 4

For ε, I experimented ε=0.2, 0.6, and 1.0. The rest of the hyper-parameters are kept the same for all three senarios, and they are:

discount rate γ=0.99
learning rate α=0.0005
hidden layer size (32, 64, 16)
optimizer='Adam'
minibatch size=64
ε=0.2, 0.6, 1.0
epsilon decay=0.995
number of episodes=500

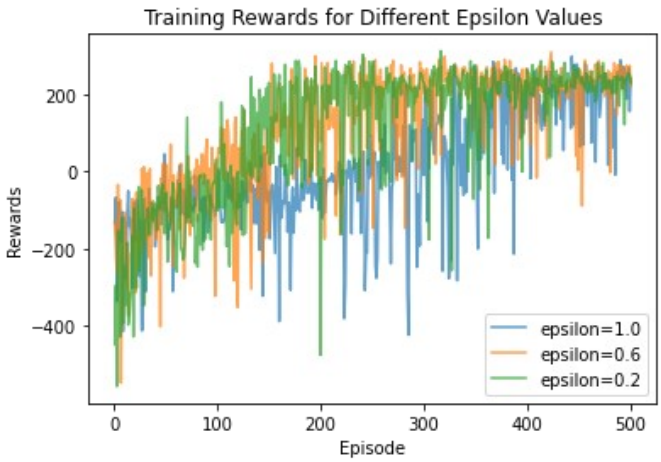

Figure 5

Figure 5 illustrates the rewards for each training episode for agents trained with different values for ε. It shows that when ε is 0.2 and 0.6, the rewards increase much faster than when ε is 1.0. In both scenarios the agents are able to achieve the same level of rewards as agent trained with ε=1.0. This differs from what I was expecting. My expectation is that when ε value is small, given the same rate of epsilon decay, the agent won't be able to do much exploration which would results in poor performance. Figure 6 shows the rewards of the trained

agents over 100 consectuve episodes. This again demonstrates that when ε is 0.2, the agent performs really well. I think the explaination is that: 1. The environment itself isn't that complex. There are only 8 state variables and 4 possible actions to take. Meanwhile, I'm using a fairly deep ANN to approximate the action-value function. So the ANN was able to learn well even with limited exploration. 2. The experience replay mechanism allows the agent to utilize the experience from exploration more efficiently. As to why the training rewards converge much faster with small ε values, the reason is that with small ε values and same rate of epsilon decay, the espilon becames very small much more quickly, as illustrated in Figure 7. With small ε, the agents follow the learned policy much more closely during training.

Figure 6

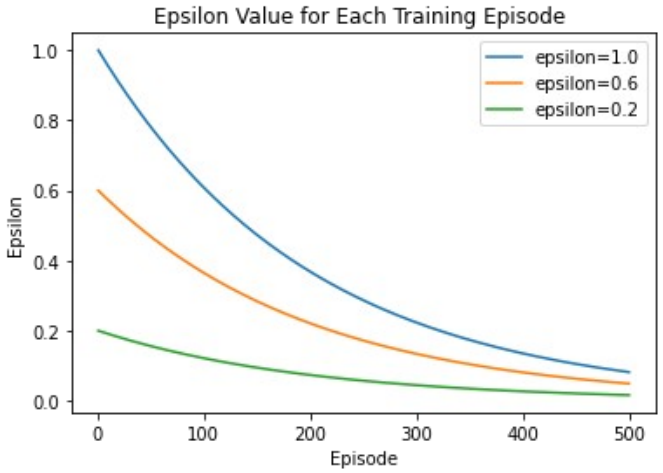

Figure 7

For discount rate γ, I experimented γ=0.2, 0.6, and 0.99. The rest of the hyper-parameters are kept the same for all three senarios, and they are:

discount rate γ=0.2, 0.6, 0.99
learning rate α=0.0005
hidden layer size (32, 64, 16)
optimizer='Adam'
minibatch size=64
ε=1.0
epsilon decay=0.995
number of episodes=500

When γ is small, the action-value function gives less weight to future reward, and the weight decrease exponentially over time steps into the future. As Figure 8 shows, when γ is 0.6 and 0.2 the agents doesn't learn well at all, and agent trained using γ=0.2 performs slight worse than agent with γ=0.6. Because the value function is focus on current and more recent rewards and greatly discounting future rewards. In other words, at a earlier time step, whether the lander crashes or lands in the future can be way less important than the verlocity or the angle of the lander.

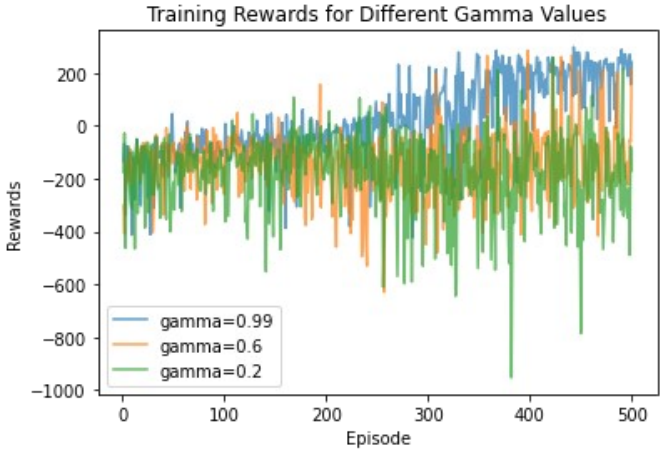

Figure 8

For learning rate α, I experimented α=0.00001, 0.0005, and 0.01. The rest of the hyper-parameters are kept the same for all three senarios, and they are:

discount rate γ=0.99
learning rate α=0.00001, 0.0005, 0.01
hidden layer size (32, 64, 16)
optimizer='Adam'
minibatch size=64
ε=1.0
epsilon decay=0.995
number of episodes=500

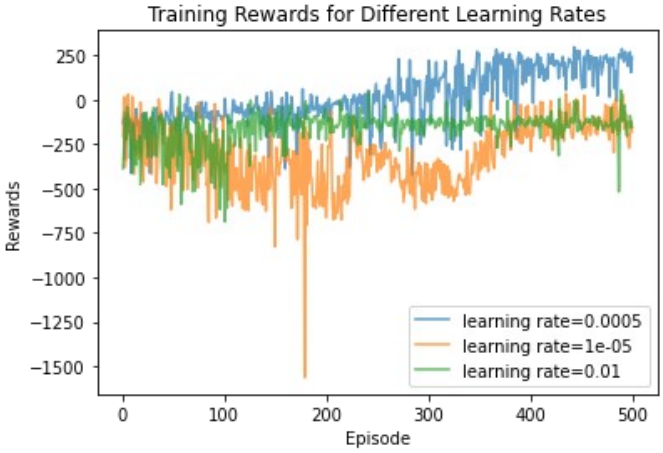

Figure 9

One mistake I made while working on the project was to use a learning rate that was too large. My algorithm didn't seem to work at all at the beginning, I thought there was bug in my code. After spending hours trying to debug the code, I couldn't find anything. I realized in the end that the problem was with the learning rate. As shown in Figure 9, when learning rate is too small or too large, the agents perform very poorly. When learning rate is too large, the ANN fails to converge to the optimum. When learning rate is too small, the ANN learns too slowly. As ε decreases to a very small value and the agent start to take less random actions, it still hasn't yet been able to learn the action-value function.

V Conclusion

The goal of this project is to solve the Lunar Lander problem. In the report, I first discussed the Lunar Lander environment. One way to solve the problem is to use Deep Q-Network (Minh et al 2015). I briefly introduced the key concepts of DQN, and explained my impletmentation of DQN, which is used to solve the Lunar Lander problem. After that, I demonstrated that an agent trained using the algorithm I impletemented based on DQN was able to solve Lunar Lander problem and achieved an average reward of 242.68 over 100 consecutive episode. I also explored the effect of different hyper-parameters. I found that different values of all the hyper-parameters I experimented, ε, γ, and learning rate, have significant impact on the performance of the algorithm.

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. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529--533.*

3 Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning.* arXiv:1312.5602 [cs.LG].

4 Ruder, S. (2017). *An overview of gradient descent optimization algorithms*. arXiv:1609.04747 [cs.LG]