Lunar Lander

---- Deep Reinforcement Learning

Manqing Mao
{GTID: mmao33}

1. Introduction

Q-Learning algorithm, a reinforcement learning technique, works by learning an action-value function which utilizes immediate reward value and future expected rewards for possible states and corresponding actions.

In this project, I first introduce the Q learning, deep Q network (DQN) and OpenAI environment named *Luna Lander* in Section 2. Next, Section 3 shows different algorithms, my DQN implementation details and difficulties I have encountered in the project. In Section 4, I simulated DQN agent in Luna Lander environment for training and get 201 rewards for sequential 100 trials. I also show how hyper-parameters like α (learning rate), ε and γ have effect on learning performance (rewards). Different curves for different hyperparameters are also provided.

2. Background

2.1 Q Learning

In Q-learning, we define a function *Q(s, a)* representing the maximum discounted future reward when we perform action *a* in state *s*, and continue optimally from that point on. Let's focus on just one transition <*s, a, r, s'*>. We can express the *Q*-value of state *s* and action *a* in terms of the *Q*-value of the next state *s'*.

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

The $max_{a'} Q(s',a')$ that we use to update $Q(s,a)$ is only an approximation and in early stages of learning and hence it may be completely wrong, however, the approximation gets more and more accurate with every iteration.

2.2 Deep Q Network (DQN)

The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the presence or absence of each individual brick. If we apply the same preprocessing to game screens as in the DeepMind paper – take the four last screen images, resize them to 84×84 and convert to grayscale with 256 gray levels – we would have $256^{84x84x4} \approx 1067970$ possible game states.

This is the point where deep learning steps in. We could represent our *Q*-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding *Q*-value. Alternatively, we could take only game screens as input and output the *Q*-value for each possible action. This approach has the advantage, that if we want to perform a *Q*-value update or pick the action with the highest *Q*-value, we only have to do one forward pass through the network and have all *Q*-values for all actions available immediately.

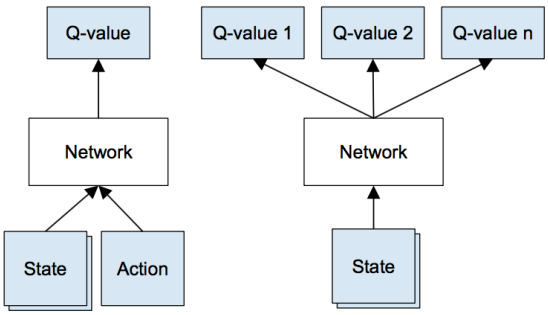


Fig. 1. (a) Naïve formulation of deep Q-network; (b) More optimized architecture of deep Q-network, used in DeepMind paper [1].

2.3 Luna Lander Environment

Lunar Lander is a simplistic game where the goal is to try to land a spaceship in a given goal area. The vector S consists of the x and y coordinates, the x and y velocities, angle, angular velocity, and ground contact information of the lander. Six of the eight variables are continuous, with only the last two being discrete (0 or 1). What's important for our agent though is only the size of the vector S meaning we can generally ignore the contents and what each variable represents. There are four actions available to control the lander at each state: do nothing, left thruster, main thruster, right thruster. The landing pad is always located in the same location (0, 0).

Rewards are given for landing within the marked area and having zero speed. Crashing results in a large negative reward (-100) and using the main thruster results in -0.3 reward at each state it's used. Finally, leg ground contact results in a +10 reward. Game is solved if an average of 200 points are collected over 100 sequential trials.

3. Algorithm Choices and Difficulties

3.1 Algorithm Choices

The first difficulty I have encountered is to choose an algorithm (agent) that works well. I have investigated several algorithms before I finally chose DQN.

Q Learning: The simplest and most straightforward idea was to discretize the continuous state space using bins of various sizes. However, this algorithm is very impractical for this project. Both too small bin size and too large bin size causes poor learning performance.

Evaluation Strategies (ES): In ES, the policies are usually stochastic, in that they only compute probabilities of taking any action. As expected, some of those actions will cause good performance, while some of them will not work out. During training, the agent may find itself in a specific state many times, and that's obviously a waste of time. Compared to DQN, ES is more straightforward to implement but it takes longer time to converge.

Deep Q Network (DQN): Google's DeepMind published its famous paper Playing Atari with Deep Reinforcement Learning, in which they introduced a new algorithm, named DQN. During the training, the agent is to build the target function as a deep neural network where the input nodes are possible states and output nodes are Q-values for all these actions. It takes a decision for next state relying on maximum output value for each action.

3.2 DQN Implementation Details

A. Neuron Network Implementation

In this project, I implementation DQN used ideas from [1], [2]. I use fully connected layer to implement DQN agent: I have one input layer that receives 8 information, which implies 8 states. There are 2 hidden layers with 40 nodes per each layer. The activation function named 'RELU' is used for each hidden layer. Then, 4 nodes in the output layer since there are 4 possible action. Graph is fully connected and it outputs Q-values which are generated using mean squared error function. The summary of the model is shown as below.

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 8)	0
dense_1 (Dense)	(None, 40)	360
activation_1 (Activation)	(None, 40)	0
dense_2 (Dense)	(None, 40)	1640
activation_2 (Activation)	(None, 40)	0
dense_3 (Dense)	(None, 4)	164
activation_3 (Activation)	(None, 4)	0

Total params: 2,164
Trainable params: 2,164
Non-trainable params: 0

B. ε Greedy Policy

In ε greedy policy, we are allowed to choose random action with probability of ε and choose optimal action based on Q values with probability of 1 - ε. ε greedy policy enables us to discover other solution that might cause better performance and avoid getting into a local optimum.

C. Experience Replay from Memory

One of the challenges for DQN is that neural network used in the algorithm tends to forget the previous experiences as it overwrites them with new experiences. Thus, we need a list of previous experiences and observations to re-train the model with the previous experiences. A method that trains the neural net with experiences in the memory is called replay().

3.3 Difficulties & Pitfalls

Keras [4] helps to simplify the code so that we can build up a neural network, predict Q values and train on mini batch very easily. However, there are 3 main challenges to implement DQN, and some of them would affect final learning performance. **(1) Extra implementation required:** remember memory, replay memory, and target function; **(2) Hyper-parameter Tuning:** there are 2 more hyper-parameters in DQN due to extra implementation of replay function, such as replay memory size and batch size. Larger replay memory size results in better performance but at price of much higher time consumption. I fix them at 10000 and 32, respectively. **(3) Large Computation but Limited Time Budget:** the training procedure takes more than 8 hours so it is not easy to do fine tuning before the deadline.

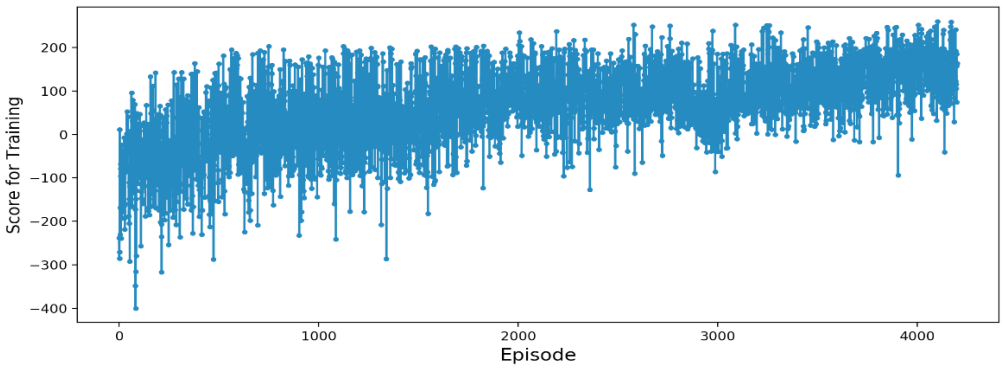
4. Experimental Results and Analysis

4.1 Results and Analysis

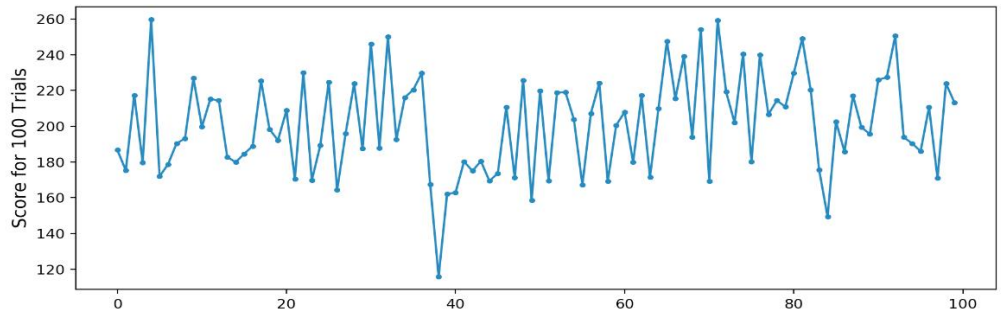
Figure 2 (a) shows the 4200 training runs of DQN. As mentioned before, I chose batch size of 32, replay memory size of 10000. I set γ to 0.99, α to 0.001. ε is started out with 1.0 with decay factor of 0.99 and decay minimum of 0.1. From the figure, we can see that the agent could

see the successful landing (reward > 200) at early episode, while the reward at the beginning diverges significantly. Increasing memory size improves stability of the learning but at price of higher training time consumption. With limited time budget, we fix replay memory size at 10000.

Figure 2 (b) shows the 100 sequential trials. The average score of 100 trials (=201.15) is over 200, treated as "solving". This happens after 4200 training episodes. From the figure, we can see the reward is almost stable during these 100 trials.



(a)



(b)

Figure 2. Rewards for each episode during (a) training and (b) 100 trials.

4.2 Hyper-parameters Investigations

In this subsection, I am going to discuss how three hyper-parameters, namely, α learning rate, ε, and γ discount factor, have effect on rewards during 1000 episodes.

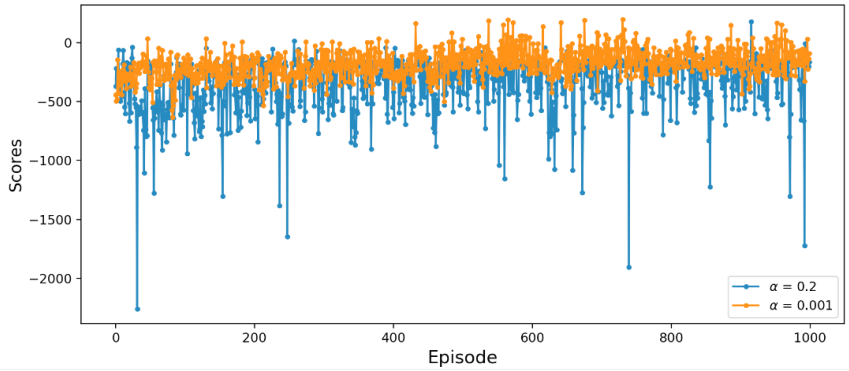


Figure 3. Comparison of rewards for each episode between agents with α of 0.2 and 0.001.

A. Very high α v.s. Reasonable α

As I learned so far, learning rate α determines how much neural net learns in each iteration. Neither too large α nor too small α results in good learning performance. The reasonable learning

rate will fall into the range of [10^{-5}, 10^{-3}], which is discussed in many papers. Compared to the α of 10^{-3}, very high results in huge loss and very low rewards, as shown in Fig 3.

B. Fixed ε v.s. ε Decays

For the agent with ε decay, ε decreases with increasing episodes. This is because we want to decrease the number of explorations (in terms of picking random action) as the agent gets better and better at playing games. Besides, we want the agent to explore at least ε_{min}, which is set to 0.1. For the agent with fixed ε, we need to allow some randomness for the undertaken action, which can be handled by setting a pre-defined probability ε. I set pre-defined ε to 0.1. Fig. 4 shows the comparison between two agents. From the figure, we can see that the agent with ε decay policy outperforms the agent with fixed ε with increasing time, as expected. Fig. 4 also shows that too small ε results in larger variances of rewards.

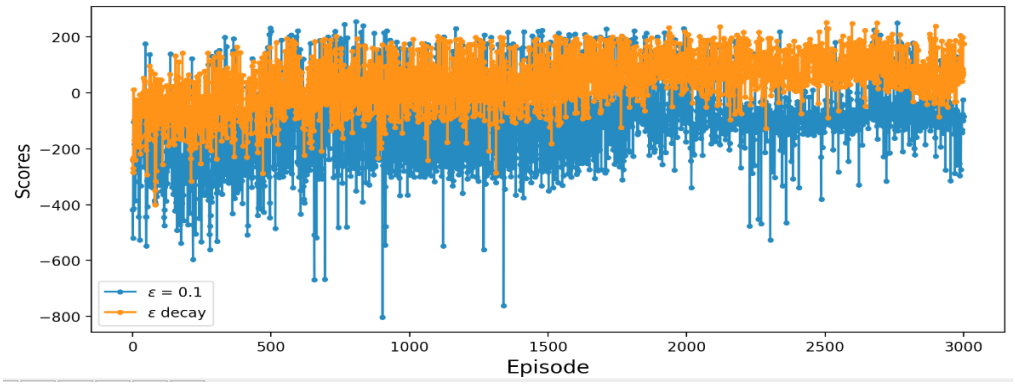


Figure 4. Comparison of rewards between agents with fixed ε of 0.1 and ε decay.

C. Small γ v.s. Large γ

γ determines how much we trust our future reward estimation. In this project, I set pre-defined γ to 0.99. Fig. 5 shows the comparison between two agents. From the figure, we can see that the agent with γ of 0.99 outperforms the agent with fixed γ of 0.5, as expected. Similarly, too small γ results in larger variances of rewards.

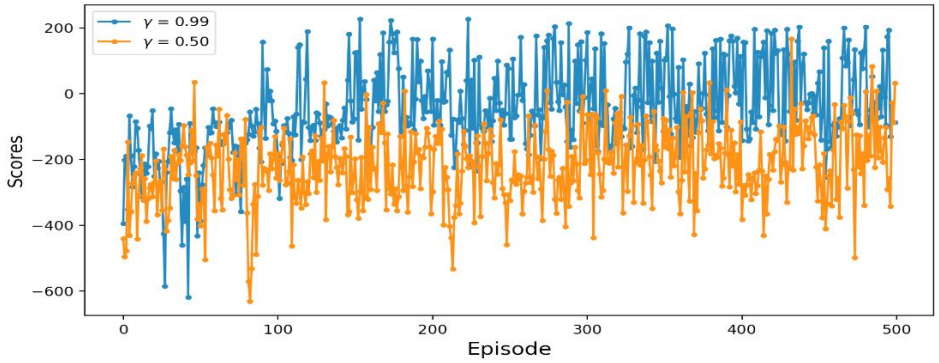


Figure 5. Comparison of rewards for each episode between agents with γ of 0.99 and γ of 0.5.

Reference:

[1] V. Mnih, et. al., "Playing Atari with deep reinforcement learning", 2013.

[2] https://www.intelnervana.com/demystifying-deep-reinforcement-learning

[3] https://blog.openai.com/evolution-strategies/

[4] https://keras.io/