Project 2: Lunar Lander Experiment Report
Gatech ID: xyu340

1. Deep Q Network Introduction
In this project, I chose DQN algorithm to train the agent. First I will explain this algorithm step by step. DQN is based on Q-learning Algorithm and a neural network. In the Q-learning Algorithm, our target is to train a policy to minimize the loss function between the target Q values and predict values.

1.1 The Basic Q-learning processes are as follows:
Initialize all parameters (reward, actions, epsilons)
For episode in episodes:
    Initialize state
    Repeat in each episode (while not done):
        Selected one action base on the policy and the states
        Update a new reward and state

1.2 The method of choosing the policy used in 1.1 are:
One is the random choose, it is the "exploration". It can explore the "new world" quickly, and update Q value efficiently. So, we prefer to use it in the beginning of the training processes.

The other is calculate the optimal action according to the current Q value. We call it greedy policy. It is the "exploitation", it can help us test whether the algorithm is useful or not. So we would like applying it in later training process.

The two policies are controlled by the epilson:
If random value < epsilon:
    Apply Exploration policy
If random value > epsilon:
    Apply Exploitation policy, namely greedy policy.

1.3 Q Value function approximation.
DQN uses Supervised Learning method to train the predicted Q value into the target value:
Create a loss function $loss = (r + \gamma \, max \, \acute{Q}(s, a') - Q(s, a))^2$
r *is the reward,* γ *is the decay rate,* s *is the state,* a *is the action.*
Using Stochastic gradient descent (SGD) to update the weights in predict function to minimize loss function.

1.4 Apply a neural network
In the normal Q-learning algorithm, we update the state and action after one episode, so it takes a long time training and low efficient. While in the Deep Q-learning, we use a neural network to get the reward based on each state. This piece of training process is within each episode.
We chose three layer, 128, 128, 4 notes respectively for each. Batch size is 64, randomly chose 64 data as SGD training sets.

Remember previous state and action by storing previous state and actions into a limited-length memory bank.
Re-train the model by using previous experience in memory bank. Update the bank automatically at the same time.

When add the 1.1 - 1.4 together, we can get the Deep Q-learning structure and use *Keras* to make it work.

2 Experiment results:
2.1 Grades change

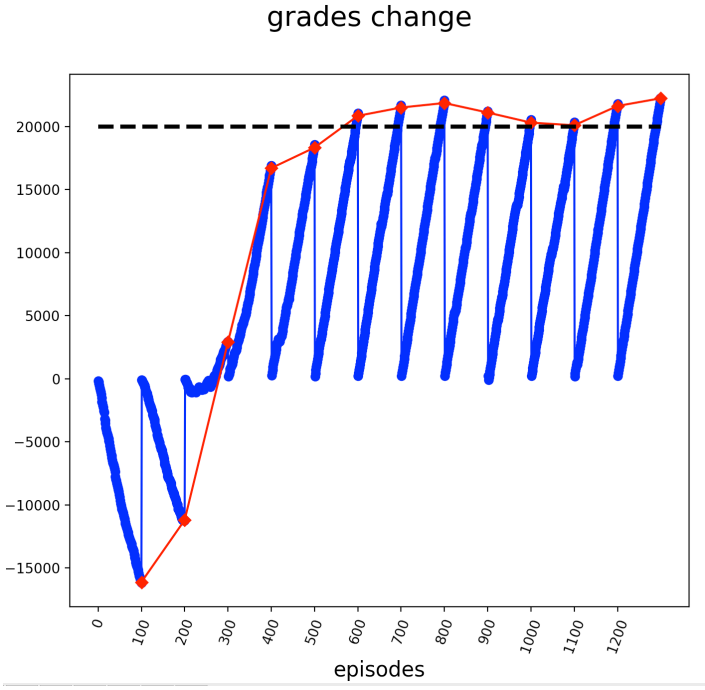

Figure 1. The blue points are accumulated rewards for all training episodes while training the agent. The plot red dots are the accumulated rewards after every 100 episodes. So we have 13 red dots and 1300 blue points for total 1300 episodes.

In the model, we reset the state after every 100 episode. So we can see within the 100 episodes, the accumulated rewards change constantly. After 100 episodes, it changes suddenly. Especially from 200 to 400 episode, the rewards change significantly from decrease trend to increase trend.

The grades keep increasing until the 20 episodes. After the grades started be greater than 200, they keep relatives stable.

2.2 Grades change for 100-trials experiment
After trained the model well, we use it to play games! See how its work done?

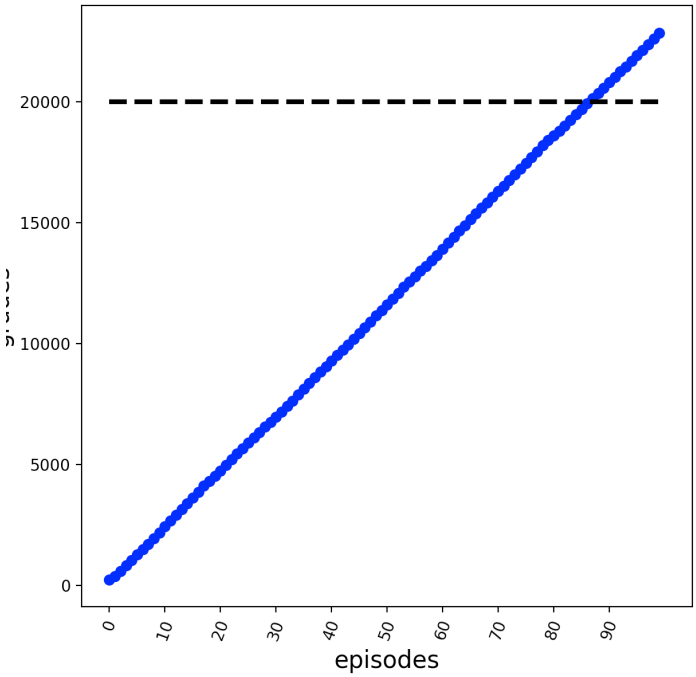

Figure 2. For a test experiments. We set 100 trials.

The reward increase very smoothly into 200 scores! Good job.

2.3 The effects of λ , γ and epsilons

Hyper-parameters perform very important roles in controlling the training processes.
λ and γ are relatively easy choose:
λ is the learning rate in the SGD process. It means the gradient step length, we set it as 0.00025. The longer it is, the less accurate the learning is. While the shorter it is, the longer time it costs.
γ is the decay rate, which means how much the current state's Q value is depend on the next state Q value. Since we use the end to end training, we would set it as 0.99.

However, choosing epsilon is very tricky. As we mentioned in 1.2 before, we will use both "Exploration" and "Exploitation" in the training process. The former helps us update Q value efficiently, so it should be used in the beginning. The latter helps us test whether the algorithm is useful or not, so it should be used later.
Two policies are controlled by the epilson, big epilson means using Exploration policy, while the small epsilon using exploitation. Thus, we need a automatically decreasing epilson values.

The methods for generating epsilon:
1). separated linear decreasing.
2). exponential decreasing. *epsilon =0.995**episode*

Two experiments which different epsilon generation processes were conducted.

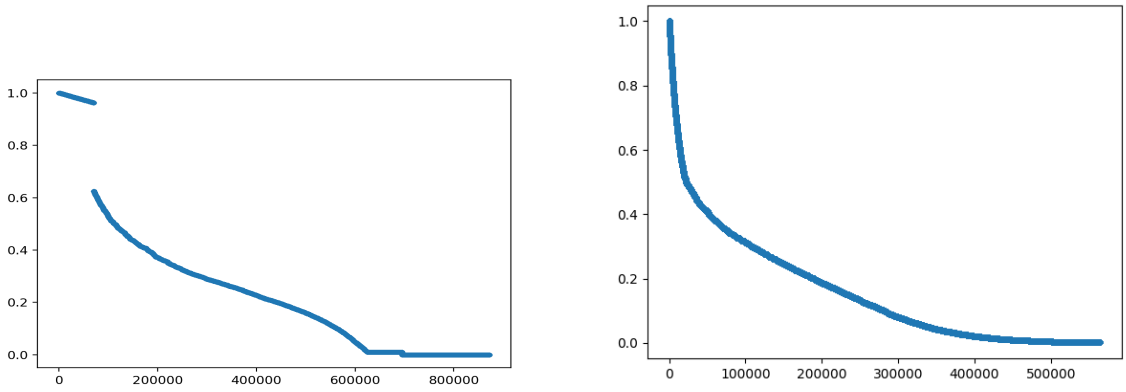

Figure 3. The epsilons updating process while training the agent in two experiments. The left is the three-step linear decreasing, the right is the exponential decreasing.

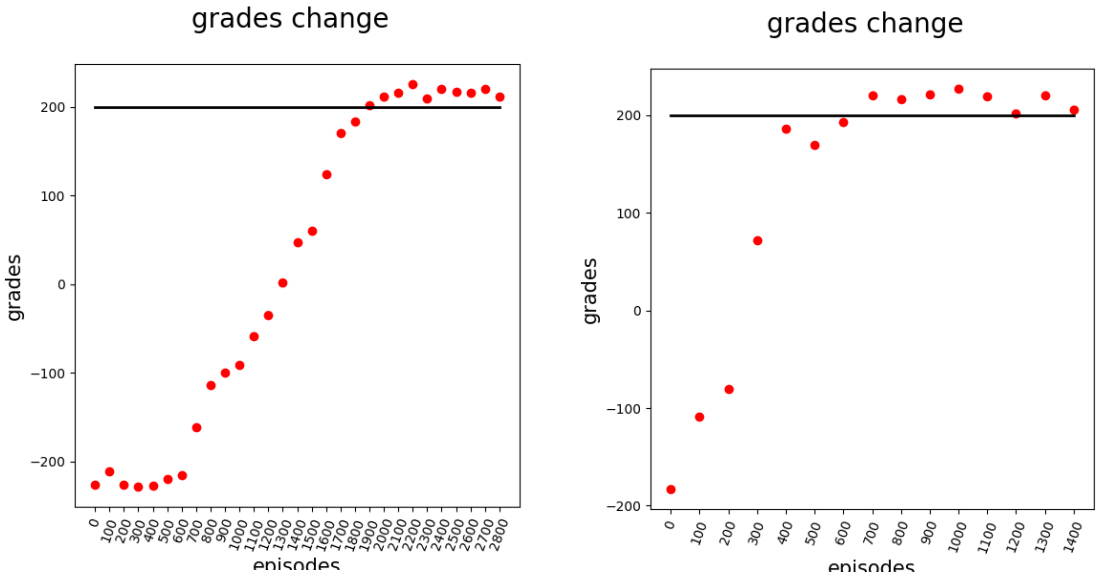

Figure 4. The grades in two experiment.

The exponential methods converge faster, it achieved 200 rewards in the 700th episode. The linearly method is much slower, it achieved 200 rewards after 1700th episode. We can also tell the faster the epsilon decreases, the faster it converges.

3. Other interesting question: reward is efficient in monitoring the training process?
No! It has some problem sometimes. I did two experiments to explain this.

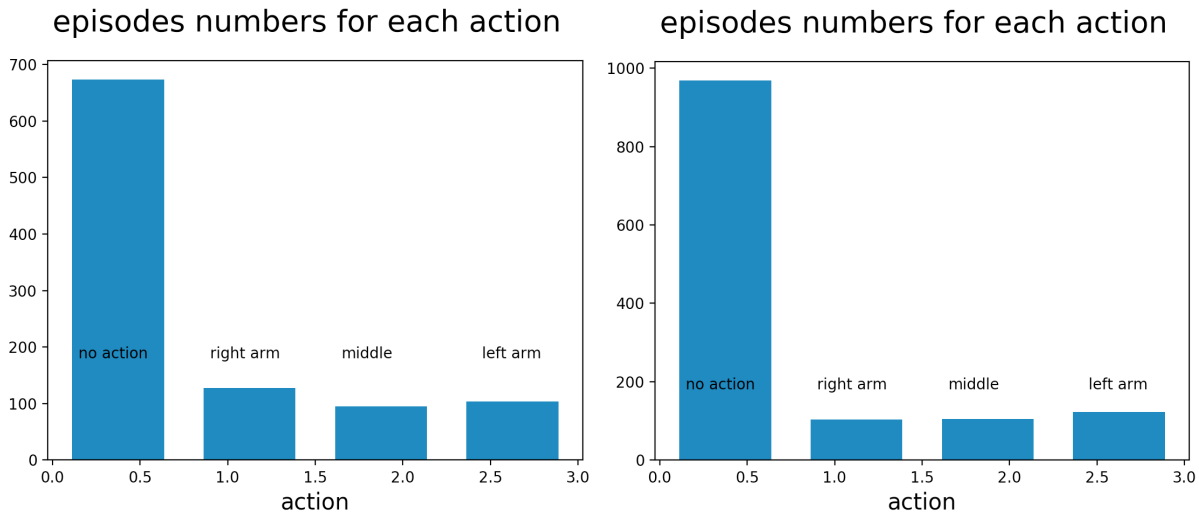

Figure 5. The action distribution for all episodes in two experiments. The left is trained 1000 episodes. The right is trained 1300 episodes. Other setting are the same.

Sometimes, it's hard to tell whether the model is well trained by just checking the rewards. In both two experiments, the rewards are stably above 200 after the 700th episode. However, in the 1000 episodes training experiment, the fire right arm action has more distribution than fire left arm. That means, the model is likely firing the right arm than vice versa. One anomaly trial experiment result from the left trained model is Shown in this video.
https://youtu.be/iqPWTSYHrQw The plane is trying so hard…

Also, we can see the plane easily moves left because the right arm fires more frequently than left arm.

4. The problems I encountered
The running time!!! My first experiment cost 5 hours!

I found some key points to cut your running time.
"Epsilon value" has a great impact on the convergence speed as explained before. Other parameters as well.
Except this, I found the vectorization also helps a lot.

Then, I tried using GPU to fasten the execution process. However, for my code, it didn't help. For the other online reference code, which is more skillful, it really did work well. The way you coding maters a lot.