**Lunar Lander Deep Q-Learning building**

**Ruonan Ding**

https://github.gatech.edu/rding31/RL

This project was designed to a learning agent for LunarLander-v2 by OpenAI gym. In this game, the agent (Lunar Lander) navigates the landing trajectory by aiming to landing pad. There are four action items with different rewards level to construct this learning. The goal is to maximize total rewards by minimizing loss to have the agent learn to land properly.

Methods:

In this problem, the method chosen is Q-Learning with a discrete action and state space. The project used a version of Q-learning called DQN( Deep Q network) learning which uses neural network to approximate the state-value function of Q(s, a). Previously, we used state action table to store the Q-value of each state-action pair. However, given that the table is super sparse and the computational limitation, we could use Neural Network to set it up. 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. The neural network is training by using observation of <state, action, reward, next-state>.

The network is trained by using <s, a, r, s’> to estimate the Bellman equation of:

Q\*(s, a) = r + *γmax a’Q(s’, a’)*

1. It will perform feedforward pass of neural net compute the Q values for state s for all actions.
2. Compute Q values by using feedforward pass for the next state s’ and calculated maximum overall network outputs *max a’Q(s’, a’).*
3. Set Q-value target for action to *r + γmax a’Q(s’, a’)* (The is the target in the script.)
4. Update the weights using backpropagation using Adam optimizer.

Experience Replay:

The agents store the experience tuples in a memory buffer first. When the model is being trained, 32 tuples were randomly selected from this memory buffer. This allows the agent to break the sequential influence of the real experience. Therefore, every state and action is more independent.

Eps-Greedy Policy:

To consider the exploration and exploitation dilemma, the agent follows a eps-greedy method in which the agent makes random action with a probability of eps and use the Q network’s output to decide the best action with a probability of 1-eps. Eps was set at 1 to fully explore the state first and slowly decay it by a factor of 0.995.

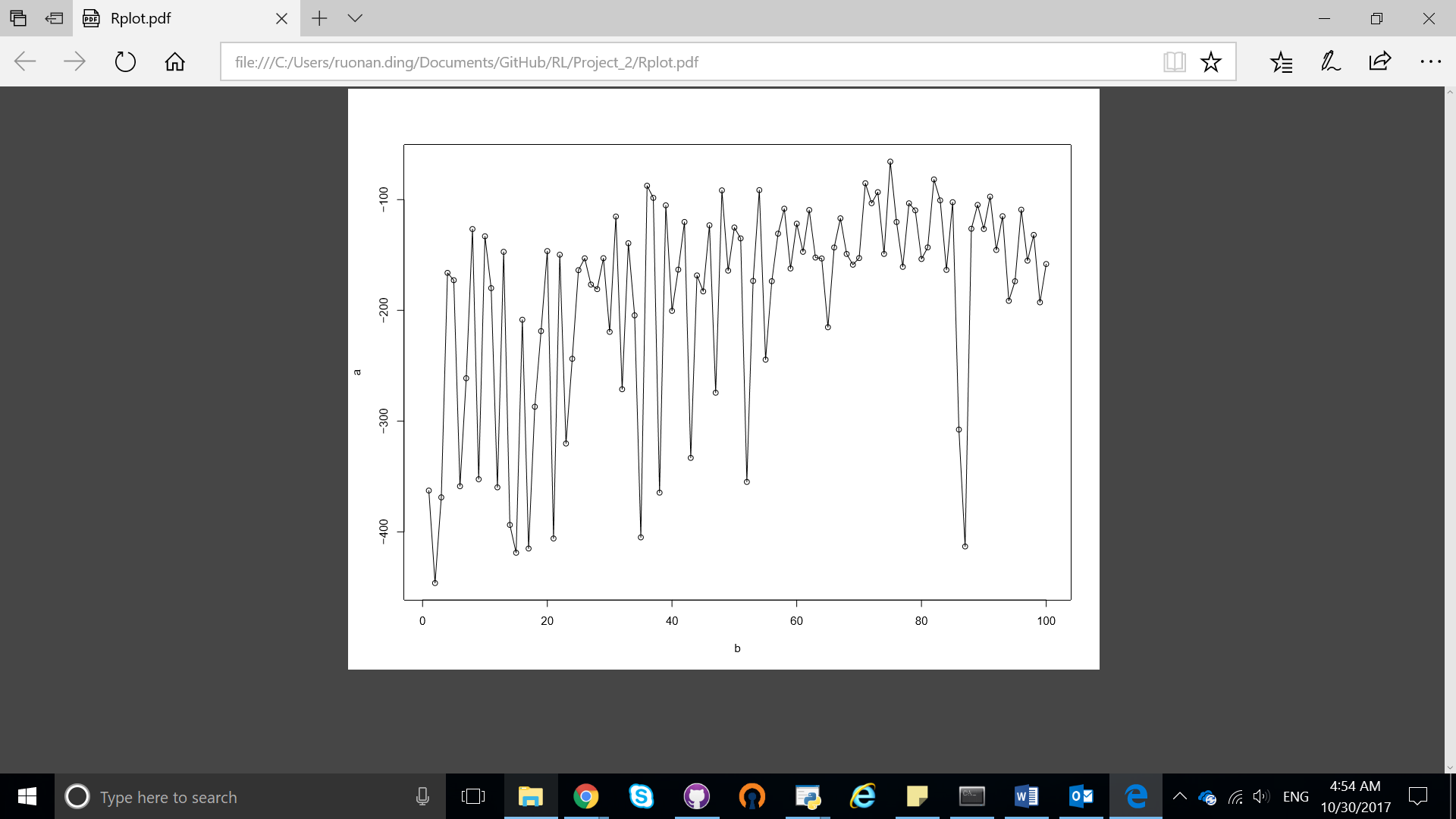
Key Assumptions of parameters:

1. 3 hidden layers with 40, 32, 16 with ReLu activation functions.
2. Target network to copy the network result
3. Loss function is MSE
4. Learning Rate 0.001
5. Number of sample observation from the buffer is 32
6. Discount factor 0.99
7. Eps decay factor is 0.996

Result:

As the script being ran, the episode output will be printed on the on the screen. There are two metrics being captured: the average loss and the total reward of an episode. In the default script, the episode is set to 500. After 500 learning periods, the average loss in the training has decreased from an average of 20+ to around 1. The total results increased from -500 to some positive numbers. The overall trend was clearly improving even though there is significant variance in the result.

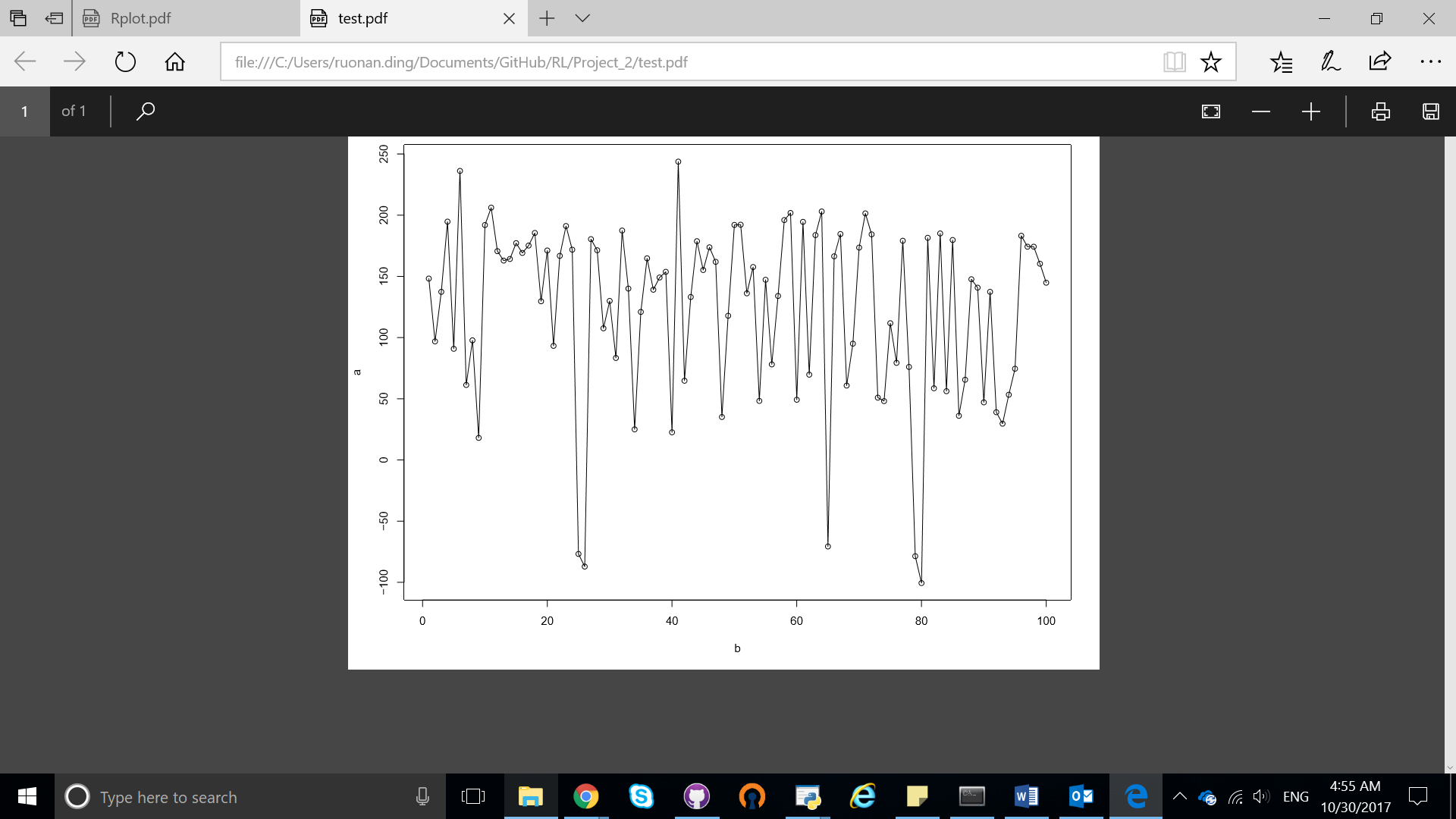
*Figure 1: Training Rewards of first 100 Episodes:*



Total Reward

The graph shows a steady improvement as the episode increases. But the total reward does fluctuate in between episode.

*Figure 2: Test Rewards by 100 Episodes:*



Total Reward

Total reward of each 500 testing episode:

Challenges:

The training was slow. The waiting time to train the 500 episodes was significant on a local computer. The other possible cause of the variance we see in the total rewards performance might be the loss function of mean squared error. It can alter the network weights substantially. Other types of loss function can be considered.

Reference:

DeepMind DQN Learning Lecture

<http://keras-rl/readthedocs.io/> Matthiasplapper’s github