ZHENG FU / zfu66@gatech.edu

Project 2: Lunar Lander V2

I. Introduction

Lunar Lander V2 is an openAI gym environment with 8-dimensional continuous state space and 4-dimensional discrete action space. The state space could be represented by the following 8-tuple:

(X, Y, Vx, Vy, Theta, V(theta), Left-leg, Right-leg)

Where X and Y are lunar lander's position coordinates, Vx is the lunar lander's velocity on x-axis and Vy is its velocity on y-axis, respectively. Theta is the lunar lander's angle towards the ground and V(theta) is its angular velocity. Left-leg and Right-leg are binary variables to indicate if the left leg and/or right leg of the lunar lander is touching the ground.

The goal of the lunar lander is to land at the landing pad that is always at coordinates (0,0). Lander moving from the top screen to landing pad with 0 velocity will receive reward between 100 and 140 points. Lander moving away from the landing pad was allowed yet it will lose reward back. Launching main engine will loss 0.3 point. Each leg touches ground will receive 10 points. The episode will be recognized as done if the lunar lander crashes or comes to rest, and it will receive -100 points and 100k points, respectively. There are four available actions for lunar lander: do nothing, fire the left orientation engine, fire the right orientation engine and fire the main orientation engine.

II. Agent

The goal of this project is to build an agent for solving the lunar lander problem, and the criteria is after a few episodes of training, the agent could receive the total reward no less than 200 points on average over 100 consecutive trails. One way to figure out this issue is to use Q-learning, in which we define a Q function to estimate the maximum future reward based on the current state and action:

Q(s_current, a_current) = alpha * [r + gamma * max[Q(s_next, a_next)]]

Where alpha is the learning rate, s_current is the current state, a_current is the current action, r is the current reward, s_next is the next state and a_next is the next action. This function is call Bellman equation. Once the Bellman equation was defined, we could choose the action with the highest Q-value.

$$\pi(s) = \text{argmax}_a\, Q(s,a)$$

$\pi(s)$ is the rule about how we should pick an action in each state.

There are many ways to solve the Bellman equation. If the state space only includes discrete variables, we could set up a Q-table and run the agent for many episodes until the Q-values in Q-table are converged. Yet in the lunar lander environment, the state space includes both continuous variables and discrete variables. Here I designed a neural network to estimate the Q-values. It has one input layer with 8 nodes corresponding to the 8-tuple of state space, 2 dense layers with 40 nodes, and one output layer with 4 nodes corresponding to the 4 possible actions of the agent. The activation function of dense layer is RELU function f(x) = max(0, x) and the activation function of output layer is a simple linear function.

At learning stage, 32 mini batches will be randomly selected from the agent's memory. For each mini batch, the neural network will take 8-tuple state vector as input and predict the reward for each possible action.

Then the Q-value will be updated by Bellman equation. Once the all 32 mini batches were processed, the neural network will fit the predictive model using 32 states and their corresponding updated Q-values. The loss function for fitting and evaluating the predictive model was a mean-squared-error function:

Loss = 0.5 * [r +   max[Q(s_next, a_next)] - Q(s_current, a_current)] ^{2}

Here I used two neural networks, one for predicting the Q-value of the current state (predict_model), and the other for predicting the Q-value of the next state(target_model). The reason for doing this is because the (r + max[Q(s_next, a_next)]) was calculated using predictive model for each training batch. Since the predictive model is changed in each training batch, this value is changed accordingly, which will cause divergence of the neural network estimation over time. Thus I used the target_model to compute (r +   max[Q(s_next, a_next)]) for each training batch, and updated it to the predict_model for every 500 steps.

III. Experiment

The experiment is composed of two stages: training stage and testing stage. In training stage, the agent will run 1000 episodes. In each episode, the agent will take an action by epsilon-greedy policy. The environment will receive the action and return the reward and next state. A 5-tuple(current state, action, reward, next state, done) will be saved in the agent's memory. Then the agent will start learning and update the state. An episode will be recognized as "finished" if any of the following three conditions were fulfilled:

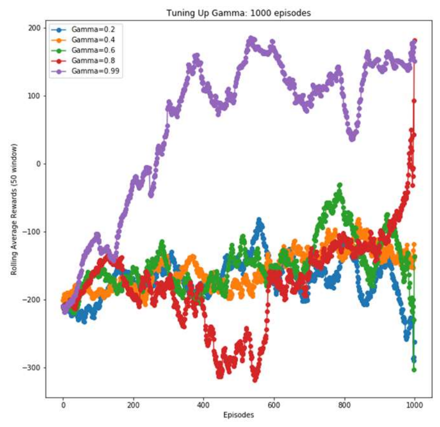
(1) reward = 100 (2) reward = -100 (3) steps within episodes more than 5000, since the environment will automatically setup done = TRUE if steps > 1000.

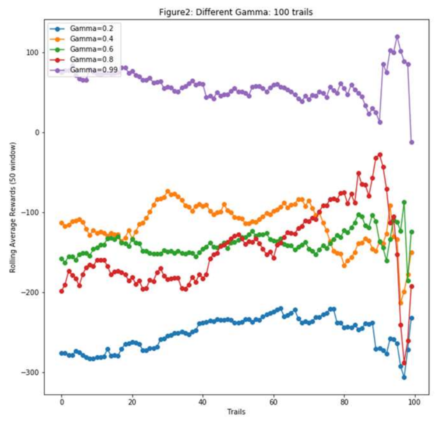
Once an episode was done, the epsilon will be decayed by a factor = 0.995. The testing stage is similar with training stage, yet it directly uses predicted Q-value to take action instead of employing epsilon-greedy policy. And it did not have memory saving steps and learning steps.

IV. Discussions

1. Discount factor:

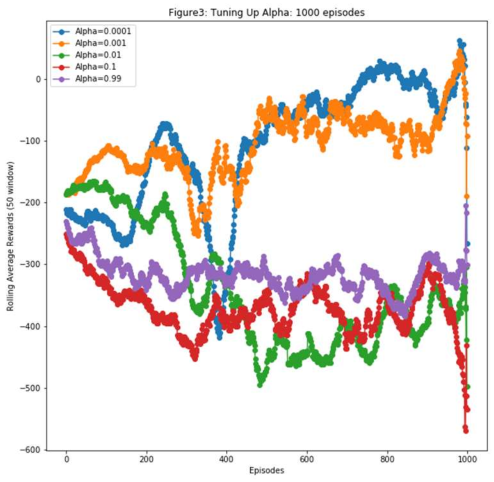
In order to investigate the effect of discount factor on the learning process, I fixed learning rate = 0.001, epsilon = 1.00 with decaying factor 0.995, used MSE as loss function and tuned discount factor from (0.2, 0.4, 0.6, 0.7, 0.99). From Figure 1 and Figure 2 we could higher discount factor will have better performance.

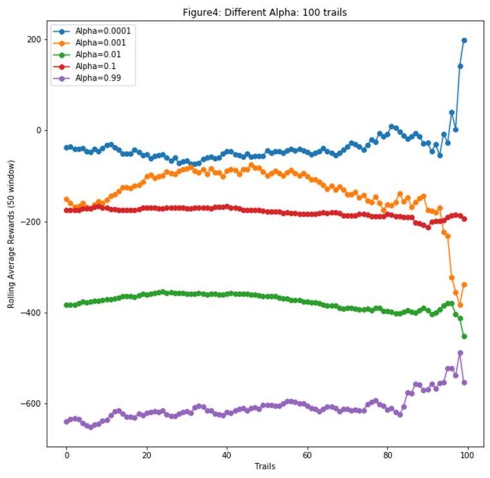




2. Learning Rate:

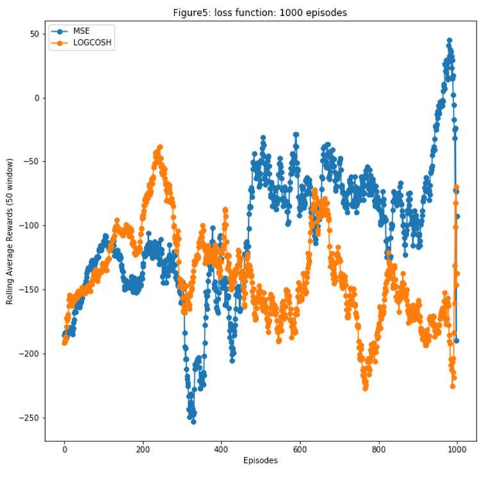
Learning rate is a key hyper parameter for Q-learning. Here I fixed gamma = 0.90, epsilon = 1.00 with decaying factor 0.995, used MSE as loss function and tuned up learning rate from (0.0001, 0.001, 0.01, 0.1, 0.99). From Figure 3 and Figure 4 we could see lower learning rate will have better performance.

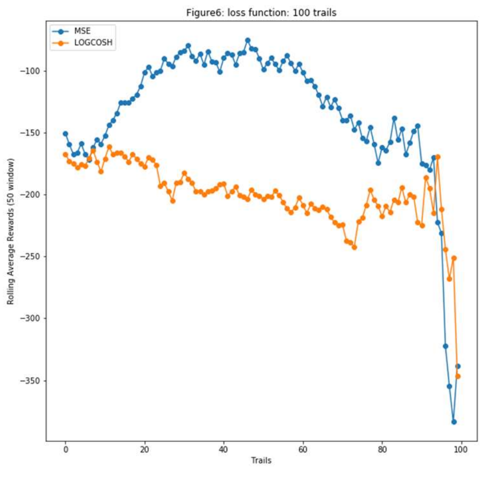




3. Loss function:

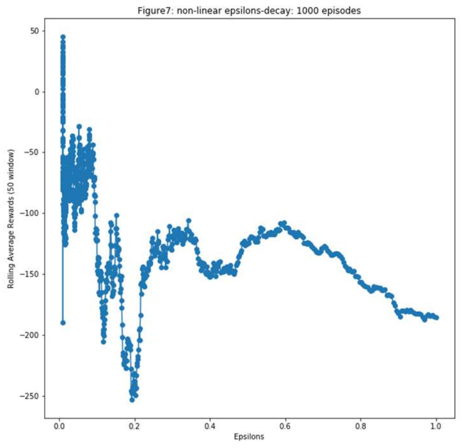
Loss function is another key factor in neural network training. Logcosh(x) is closely equal to (x ** 2) / 2 for small x and to abs(x) - log(2) for large x. It is similar with MSE yet if a wildly incorrect prediction occurred, it will not be affected dramatically. From Figure 5 and 6 we could see the overall performance of MSE are better than Logcosh in both training and testing stages (learning rate = 0.001, discount = 0.90, epsilon = 1.00 with decaying factor 0.995).

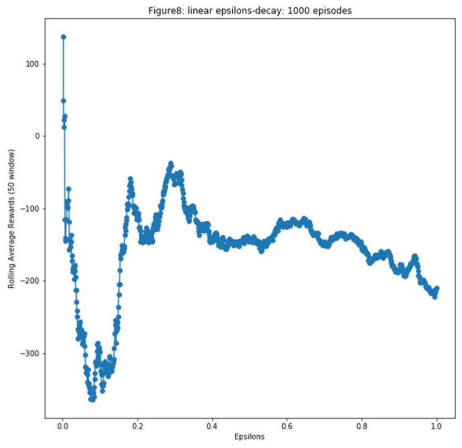




4. Epsilon-decaying strategy:

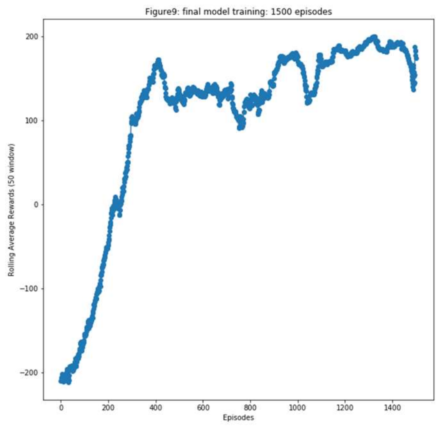
In the learning steps, the actions of agent are highly depended on the epsilon value. Here I tried two epsilon decaying strategies: one using decay factor = 0.995 and the other using linear decay. From Figure 7 and 8 we could see there is a huge reward point valley between 0.0 and 0.2 if we use linear epsilon decay. Thus I chose the strategy with decaying factor 0.995 (learning rate = 0.001, discount = 0.90, MSE as loss function).

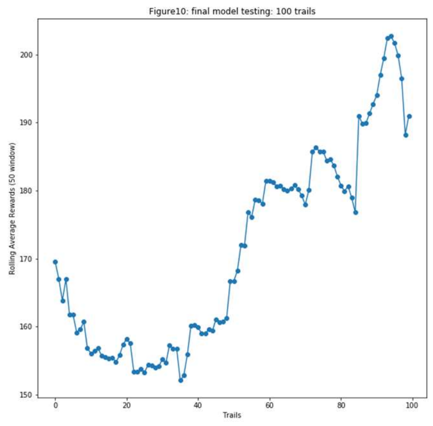




5. Final model:

From the above discussions, we could see lower learning rate, higher discount fact, along with MSE loss function and epsilon decay strategy with decaying factor 0.995 might produce better agent performance. Thus for the final model I used learning rate = 0.0001, gamma = 0.99, MSE as loss function, epsilon decay strategy with decaying factor 0.995 and 1500 episodes in training. Also I expended the width of the neural network to 80 nodes in the two dense layers. From Figure 9 and 10 we could see that at the end of training stage, the rolling average rewards are about 200, yet in the 100 trails the overall rolling average rewards are lower than 200 points. One possible reason is I set up the episode to be done when steps > 5000. 9 trails are terminated on 5001 steps with low total rewards. If that restriction is removed, I might have better results.





V. Reference:

Human-level control through deep reinforcement learning. Nature. 2015 Feb 26;518(7540):529-33.