Solve Lunar Lander with Deep Q Learning

Tiantian Guo
tguo60@gatech.edu

***Abstract*—In this project, an agent will be implemented and trained to successfully land the ”Lunar Lander” in OpenAI gym. Deep reinforcement learning [1] is adopted.**

***Index Terms*—Lunar Lander, Deep Q Learning**

I. Introduction

”Lunar Lander” can be simulated by OpenAI Gym [2]. The lunar lander is trying to land onto the lunar ground without crashing, see Fig. 1. After rounds of tries and learning, the lander will gradually learn how to land. Deep reinforcement learning [1] is a suitable solution to this landing problem.

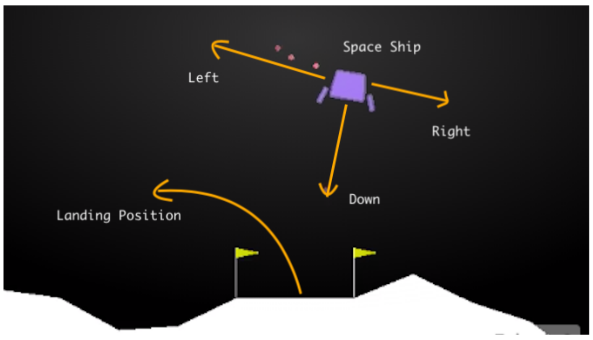

Fig. 1. Lunar Lander in Opengym.

II. Problem Statement

The ”Lunar Lander” problem consists of an 8-dimensional continuous state space s and a discrete action space a, $(x, y, vx, vy, \theta, v\theta, leg_L, leg_R)$. x and y are the x and y-coordinates of the lunar lander's position. vx and vy are the lunar lander's velocity components on the x and y axes. θ is the angle of the lunar lander. θ is the angular velocity of the lander. leg_L and leg_R are binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground. So the states are six dimensional continuous state space with the addition of two more discrete variables. The four discrete actions available are: do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine.

The landing pad is always at coordinates (0,0). Coordinates consist of the first two numbers in the state vector. The total reward for moving from the top of the screen to the landing pad ranges from 100 - 140 points varying on the lander placement on the pad. If the lander moves away from the landing pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 points respectively. Each leg ground contact is worth +10 points. Firing the main engine incurs a -0.3 point penalty for each occurrence. Landing outside of the landing pad is possible. Fuel is infinite,

Problem in this project is how to let an agent learn to land the lunar ground on its first attempt.

III. Solution

The problem is considered solved when the reward in one run achieving a score of 200 points or higher on average over 100 consecutive runs.

The agent is in a state s and has to choose one action a, upon which it receives a reward r and come to a new state s', see equation (1). The way the agent choosing actions is called $policy$. If the agent knows the expected reward of each action at every step, it will know exactly which action to perform at each step and finally learn the $policy$ that can achieve target.

All states that come in between an initial-state and a terminal-state is called one $episode$. The agent's goal it to maximize the total reward it receives during an episode.

$$s \xrightarrow{\text{a}} r, s' \tag{1}$$

Deep Q learning is the solution for this project. Deep Q learning uses neural network based on Q learning. In this report, Q learning (Section III-A) will be introduced first to facilitate the understanding of deep Q learning in (Section III-B).

A. Q learning

In Q learning, the agent will perform the sequence of actions that will eventually generate the maximum total reward.

$Q(s, a)$ yielded from being at state s and performing action a, is the immediate reward r(s,a) plus the highest Q-value possible from the next state s', see in equation (3). γ is a discount factor and $\gamma < 1$, it makes sure that the sum in the formula is finite. The lower the discount factor is, the less important future rewards are, and the agent will tend to focus on actions on immediate rewards only. $Q(s', a)$ recursively depends on $Q(s'', a)$, see equation (4).

$$Q(s, a) = r(s, a) + \gamma \text{max}_a Q(s', a) \tag{3}$$

$$Q(s, a) \to \gamma Q(s', a) + \gamma^2 Q(s'', a)......+\gamma^n Q(s^{''...n}, a) \tag{4}$$

Algorithm 1: Deep Q-learning

```
Initialize replay memory $D$ to a FIFO queue of capacity $N$;
Initialize neural network training batch size $BS$;
Initialize action-value function $Q$ with random weights;
Construct neural network mapping from states to rewards;
for episode = 1, $M$ do
    Reset game environment;
    Reward = 0 ;
    for step = 1, STEP do
        Select an action $a$;
            Select a random action with probability $\epsilon$;
            Otherwise select $a$ = argmax$'_a$Q(s,$a'$);
        Store $< s, a, r, s' >$ in replay memory $D$ after apply action $a$;
        Reward = Reward + r;
        s = $s'$;
        if replay memory buffer size <= N and replay memory buffer size >= BS and in every 3
         episode then
            Collect train batch randomly from replay memory;
            Train neural network using $(r - Q(s,a))^2$ as loss function and ;
        end
        if Done then
            Break
        end
    end
    $\epsilon$ = $\epsilon$ * decay;
    if Last 200 episodes rewards average > 200 then
        Break
    end
end
```

$$\nabla_{\theta_i} L_i(\theta_i) = \mathrm{E}_{s,a\sim\rho(.);s'\sim\epsilon}\left[(r + \gamma \max_{a'} \mathrm{Q}(s', a'; \theta_{i-1}) - (Q)(s, a; \theta_i))\ \nabla_{\theta_i}\ \mathrm{Q}(s, a; \theta_i)\right] \tag{2}$$

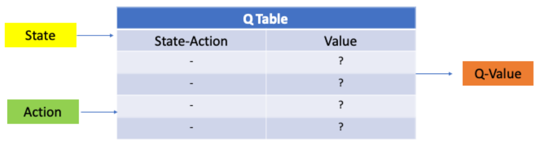

Fig. 2. Q Learning.

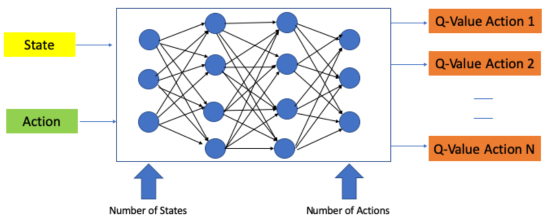

Fig. 3. Deep Q Learning.

The agent constantly performs the action that is believed to achieve the highest expected reward. The agent needs to try many different actions in many different states in order to try and learn all available possibilities and find the *policy* which will maximize its overall reward, this is known as exploration, as the agent explores the environment. However, the agent should also use the informationit learned. The agent need to exploit and explore its knowledge to maximize the rewards it receives. Greedy policy, ϵ-Greedy Policy allows agent to randomly explore. A number ϵ in the range of [0,1] is selected, if that number is larger than ϵ, the greedy action is selected while a random action is selected if the number is lower. If ϵ = 0, the policy becomes the greedy policy, and if ϵ = 1, always explore. Such value iteration algorithm converge to the optimal action value function, $Q_i \rightarrow Q*$ as $i \rightarrow \infty$. Finally, a Q table will be built for states to choose expected best action from, see Fig.2.

B. Deep Q Learning

The continues states and descrete actions in 'Lunar Lander' OpenAI Gym scenario, require deep reinforcement learning [1] to avoid the complexity of store Q table. Instead, the Q values are calculated by neural network by states and corresponding action rewards, see in Fig.3.

The algorithm can be described in algorithm 1. As reinforcement learning tasks have no pre-generated training sets which they can learn from, the agent must keep records of batches of the state-transitions it encountered in the replay memory buffer so it can learn from them later.

A function approximator is used to estimate the action-value function in deep Q learning, $Q(s, a; \theta) \approx Q^*(s, a)$. In the continuous agent state and discrete actions environment, a neural network function approximator with weights θ constructs a Q network. A Q network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = E_{s,a\sim\rho(.)}[(y_i - Q(s, a; \theta_i))^2] \tag{5}$$

where $y_i = E_{s'\sim\epsilon}[r + \gamma \text{max}_{a'} Q(s', a', \theta_{i-1})|s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and action a which is called behaviour distribution. Differentiating the loss function with respect to the weights.

Equation (5) is computationally expedient to optimise the loss function by stochastic gradient descent. The learning agent is following the Q learning algorithms [3] with the neural network's assistance.

The deep Q learning is described in Algorithm I (1). Each step of batch experience is potentially used in many weight updates. Considering strong correlations between samples, randomizing the samples can break correlations and reduces the variance of the updates. When learning on-policy the current parameters determine the next data sample that the parameters are trained on. The timing for update of weights is also considered. Many experiments have been done to test the optimal number of episodes for updating the weights. The setting of batch size and the number of every other episodes to update weights are crucial parameters for rewards to converge. After several rounds of experiments, if weights update every N episode, N ¡ 5, the network might not be stable and will not achieve rewards 200 or even drop below 0. And if the batch size is set too large or too small, the rewards also cannot converge.

Based on the theory of deep Q learning, the training result of 'Lunar Lander' is shown in Section IV.

IV. Result

Neural network in the experiments is designed to be three layers, input layer, hidden layer and output layer. Batch size of neural network training data is set to be 64, the number of episodes that update neural network weights is set to be 5.

In the training, neural network is built with Keras [4]. First, rewards at each training episode are demostrated in section IV-A. Then, hyperparameters differences' impact on the training is discussed in section IV-B. Results will be also simulated in section IV-C while neural network adopts various structures.

A. Rewards

In this subsection, the neural network has input layer with the number of neurons equaling to the number of states and relu activation function, first hidden layer with 512 neurons and relu activation function, second hidden layer with 256 neurons and relu activation, output layer with the number of naurons equaling to the number of actions and linear activation function.

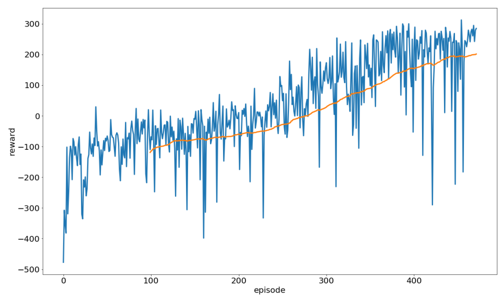

Fig. 4. Reward per episode for 8-512-256-4 linear activation layers

The parameters for this experiment is set as: learning rate = 0.001, $\epsilon = 1.0$, decay of $\epsilon = 0.995$, γ = 0.99, the maximum number of training episodes is set to be 1000.

As seen in Fig 4, the rewards are gradually increasing from about -477 to above 200. And when the number of training episode achieves 470, the average reward of recent 100 episodes converges to above 200, which achieve the success landing definition in the 'Lunar Landing' project. Then the agent based on deep Q learning is considered to be trained successfully.

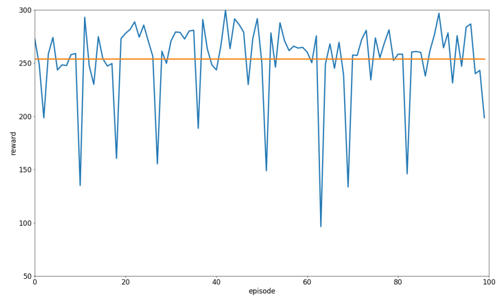

Fig. 5. Reward per episode for 100 consecutive episodes using trained agent.

Next the trained agent is tested in complete different 100 consecutive randomly generalized scenarios of 'Lunar Landing', the rewards versus episode is plotted in Fig. 5. The mean of 100 episode test rewards is 253.7. Among 100 episode, 90 episode rewards are larger than 200. The minimum episode reward is 96.37.

B. Hyperparameters

Hyperparameters like neural network learning rate, future Q value discount factor γ and ϵ decay value will be discussed in this part.

1) Neural network learning rate: Neural network learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each round the model weights θ are updated. Too small learning rate may result in a long trainign process. Too large learning rate may result in a sub-optimal set of weights or an unstable training process.

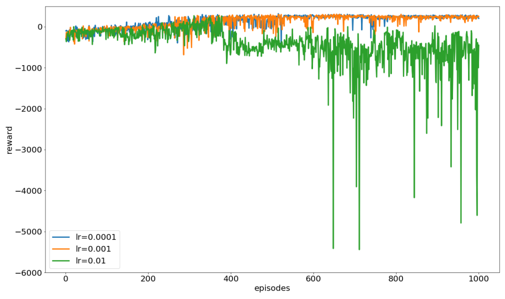

Fig. 6. Reward per episode for different learning rates.

The parameters for this experiment is set as: learning rate ranges in [0.0001, 0.001, 0.01], $\epsilon = 1.0$, decay of $\epsilon = 0.995$, γ = 0.99, the maximum number of training episodes is set to be 1000.

The results can be seen in Fig 6. When learning rate is 0.0001, the average rewards of 1000 episode is 129.78, 53.4% of episode rewards are larger than 200. The training process begins to converge at 479th episode. When learning rate is 0.001, the average rewards of 1000 episode is 122.68, 51.5% of episode rewards are larger than 200. The training process begins to converge at 510th episode. When learning rate is 0.01, the average rewards of 1000 episode is -422.0, 0.9% of episode rewards are larger than 200. The training process doesn't converge.

2) Discount factor γ: Discount factor γ quantifies how much importance future rewards weighing. It is also handy to approximate the noise in future rewards.

In this experiment, parameters are set as: γ ranges in [0.999, 0.99, 0.9], learning rate = 0.001, ϵ = 1.0, ϵ decay = 0.995, training episodes = 1000.

It can be seen in Fig 7. When γ is 0.999, the average rewards of 1000 episode is -844.30, none episode rewards are larger than 200. When γ is 0.99, the average rewards of 1000 episode is 146.70, 58.8% of episode rewards are larger than 200. The training

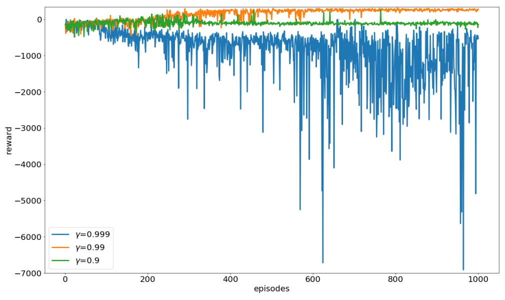

Fig. 7. Reward per episode for different γ values.

process begins to converge at 425th episode. When γ is 0.9, the average rewards of 1000 episode is -101.84, 0.3% of episode rewards are larger than 200. The training process doesn't converge.

3) ϵ decay: ϵ is associated with how random the agent take an action. As the learning goes on, ϵ should be decayed to stabilize and exploit the learned policy which converges to an optimal one, so that higher Q-values can be found in higher probability.

In this experiment, parameters are set as: ϵ decay ranges in [0.999, 0.995, 0.990], neural network learning rate = 0.001, ϵ = 1.0, γ = 0.99, training episodes = 1000.

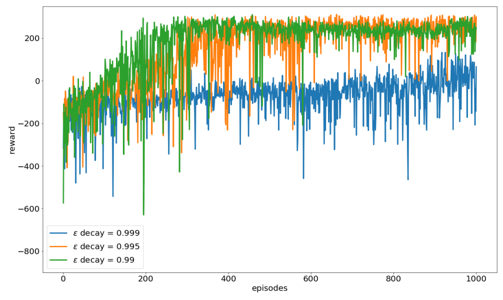

Fig. 8. Reward per episode for different decay values.

The results can be seen in Fig 8. When decay is 0.999, the average rewards of 1000 episode is -74.14, no episode rewards are larger than 200. When decay is 0.995, the average rewards of 1000 episode is 143.31, 55% of episode rewards are larger than 200. The training process begins to converge at 378th episode. When decay is 0.99, the average rewards of 1000 episode is 170.69, 68.7% of episode rewards are larger than 200. The training process rewards converges to 200 at episode 243.

C. Neural network structure

Extra experiments have been done to explore the performance of different neural network architectures.

1) Neural network setting 1: Simpler Setting: In this subsection, the neural network has a simpler setting: input layer with the number of neurons equaling to the number of states and relu activation function, first hidden layer with 128 neurons and relu activation function, second hidden layer with 64 neurons and relu activation, output layer with the number of neurons equaling to the number of actions and linear activation function.

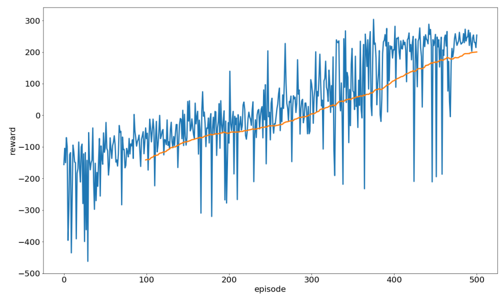

Fig. 9. Reward per episode for 8-128-64-4 linear activation layers.

The training results are shown in Fig 9, the rewards can converge to 200 in episode 500.

2) Neural network setting 2: Dropout Effect: In this subsection, an experiment has been done to observe the dropout layer effect on the neural network. The other network layer settings are the same as subsection IV, besides an additional dropout layers with dropout rate of 0.2 is added before the output layer. Dropout is remarkably effective regularization method to reduce overfitting and improve generalization error in deep neural networks.

As can be seen in Fig 10, the rewards do not converge in 1000 episodes. That means in the network contructed in subsection IV, there is no need to do dropout regularization.

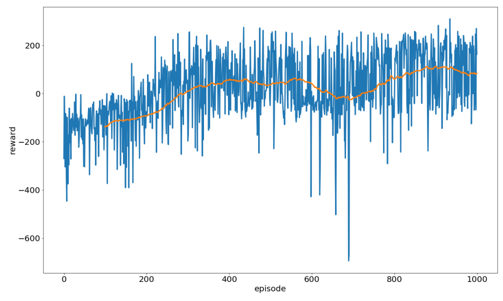

Fig. 10. Reward per episode for 8-512-256-4 linear activation layers with dropout 0.2.

3) Neural network setting 3: Activation Setting: In this subsection, the neural network adopts softmax activation instead of linear in the output layer.

The neural network configurations are input layer with the number of neurons equaling to the number of states and relu activation function, hidden layers with 512, 256 neurons, output layer with the number of neurons equaling to the number of actions and softmax activation function.

It can be seen in Fig 11, the rewards line cannot converge to 200 and even worse till -1067.73 in 500 episodes. That means the agent does not learn anything from the past experiences and environment using softmax activation.

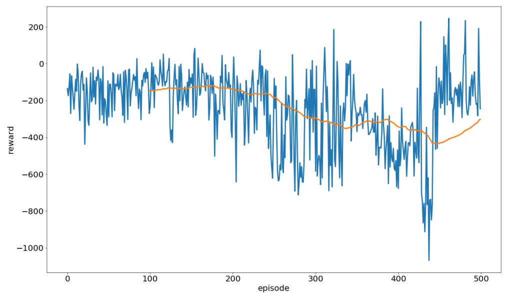

Fig. 11. Reward per episode for 8-512-256-4 softmax activation layers.

V. Related work

Currently, there are several works, that have noticed the dropback of the neural network training batch evenly randomly selection from replay memory buffer. There are kinds of training data selection algorithms, which consist of keeping training over its new behavior [5] or conditioning each agent's value function on a fingerprint that disambiguates the age of the data sampled from the replay memory [6]. Advanced designs of deep RL techniques lead to even lesser number of episodes for agent to learn the environment.

VI. Conclusion

In this project, 'Lunar Lander' problem in OpenAI Gym is solved by deep reinforcement learning. Multiple experiments have been done to explore the effect to the learning process caused by different hyberparameters and different neural network desgins.

References

[1] Volodymyr Mnih and Koray Kavukcuoglu and David Silver and Alex Graves and Ioannis Antonoglou and Daan Wierstra and Martin A. Riedmiller, *Playing Atari with Deep Reinforcement Learning*,CoRR abs/1312.5602, 322(10):891–921.

[2] Gym, https://gym.openai.com/

[3] Christopher JCH Watkins and Peter Dayan., Q-learning, *Machine Learning*, 8(3-4):279-292, 1992

[4] Keras, https://keras.io/

[5] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, Prioritized Experience Replay, *arXiv:1511.05952v4*, 2016

[6] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. Torr, Pushmeet Kohli, Shimon Whiteson, Stabilising experience replay for deep multi-agent reinforcement learning, *ICML'17: Proceedings of the 34th International Conference on Machine Learning*, vol 70, Pages 1146–1155