Lunar Lander

Lunar Lander Problem

The aim of the problem is to land the spaceship or lunar lander on its landing pad by controlling it with different actions through the space environment.
There are four discretion actions: do nothing, fire the left orientation engine, fire the right main engine, fire the right orientation engine.
At each time step, the state is provided to the agent as a 1*8 vector:

(x, y, vx, vy, θ, vθ, left-leg, right-leg)

x and y are the x and y-coordinates of spaceship. vx and vy are the lunar lander's velocity components on the x and y axes. Θ is the angle of the spaceship, v Θ is the angular velocity of the spaceship. Left-leg and right-leg are binary values to indicate whether the left leg and right leg of the lunar lander is touching the ground.
The landing pad is at coordinates (0,0). Total reward for moving from the top of screen to landing pad ranges from 100 -140 points varying on lander position on the pad. If ladder moves away from pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode finished if the lander crashed or comes to rest, receiving additional -100 or +100 respectively. Each leg ground contact is worth +10 points. Firing main engine incurs a -0.3 points penalty for each occurrence. Fuel is infinite. The problem is considered solved when achieving a score of 200 points or higher on average over 100 consecutive runs.

DDQN

Q Learning

Q learning is a model-free value-based reinforcement learning algorithm. The goal is to learn a policy such that for a given state, an agent can choose the best action and transfer to another state. A Q table is built to help agent find the best action for each state. The main algorithm is a value iteration update derived from the Bell-Equation.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Where α is the learning rate (0 <= α < =1), and ϒ is the discounted factor.

Epsilon-Greedy Algorithm
Any online learning and decision making process includes two parts: exploitation and exploration. Exploitation refers to make the best decision given current information, exploration refers to gather more information. A good algorithm should balance them, focus on exploration at the beginning of learning process and gradually focus more on exploitation.

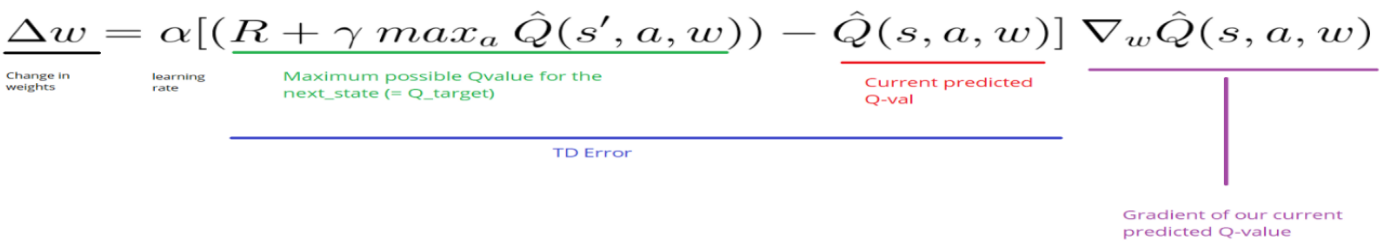
A common approach is epsilon-greedy algorithm controlled by parameter ϵ-decay value and minimal and maximal ϵ value.
When agent chooses an action each time, with probability of current epsilon, a random action is chosen among all possible actions. The epsilon is decreased by ϵ-decay value every episode. The initial epsilon is close to 1 and after large enough episodes, its value becomes 0.

Deep Q Network

Q learning is a good algorithm to handle simple problem. However, when the state space becomes larger, it is not scalable anymore. The lunar lander problem state space is continuous and infinite. So, a more powerful algorithm called deep Q network is needed.

Deep Q network algorithm uses a neural network to approximate, given a state, the different Q values for each action. For the lunar lander problem, the network has 2 fully connected hidden layers, which has 128 nodes per layer with ReLU activation functions. The input layer has 8 nodes representing the state vector 8 dimensions. The output layer has 4 nodes representing the 4 possible actions.

Instead of update Q function value during each episode, for DQN, the neural net weights are updated to reduce the error. A formula from [1] is below:



Mean square error loss function is used here to train the network.

Experience Replay

Experience reply is adopted to make more efficient use of observed experience. The experience here is a tutple <s, r, a, s_prime>. Instead of using the tuple to update Q function directly, they are stored in a fixed size replay memory. The current Q value and predicted Q value is calculated by randomly sampling a batch size of tuples from the replay memory at each episode. The advantage of experience replay is that algorithm can learn not only from recent transition but also previous old transition, and it can only reduce the correlation between transitions.

Double Deep Q Network

Double DQN is used to solve the problem of Q-values overestimations in DQN. In Q learning, we choose the best action for the next state which has the highest Q value. At the beginning of training, there is not enough information in Q functions. So, taking the best action based on the non-accurate Q value may lead to worse result. In order to solve this overestimation issue, action selection and target Q value generation are decoupled. We use the online network to select the action greedily, and the target network to update the Q value. A formula from [1] is shown below:

Model Parameters

This section list all parameters used to get the best performance.

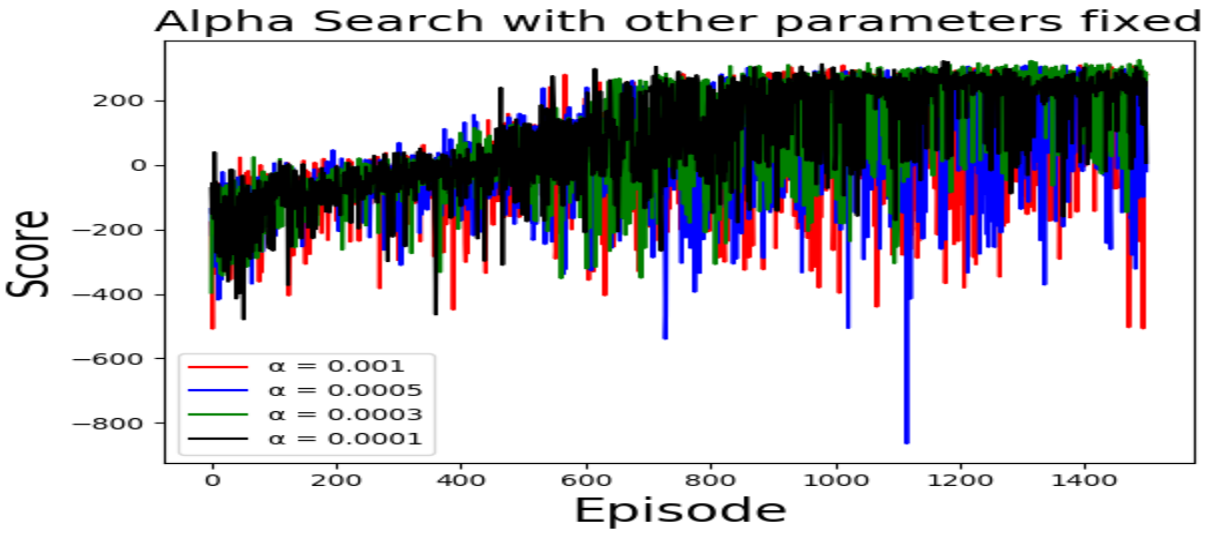
Algorithm	Double Deep Q Network

Neural Network Structure	8 nodes input layer, two 128 nodes hidden layer, 4 nodes output layer
Activation Function	ReLU
Loss Function	MSE
Batch Size	32
Learning Rate α	0.0001
Discounted Factor ϒ	0.99
Min Epsilon	0
Max Epsilon	1
Epsilon Decay	0.998
Episode Number	2500

Due to the limited time and resources, only learning rate α and discounted factor ϒ are tuned and analyzed further in later sections. All other parameters are fixed.

Learning rate α Effect

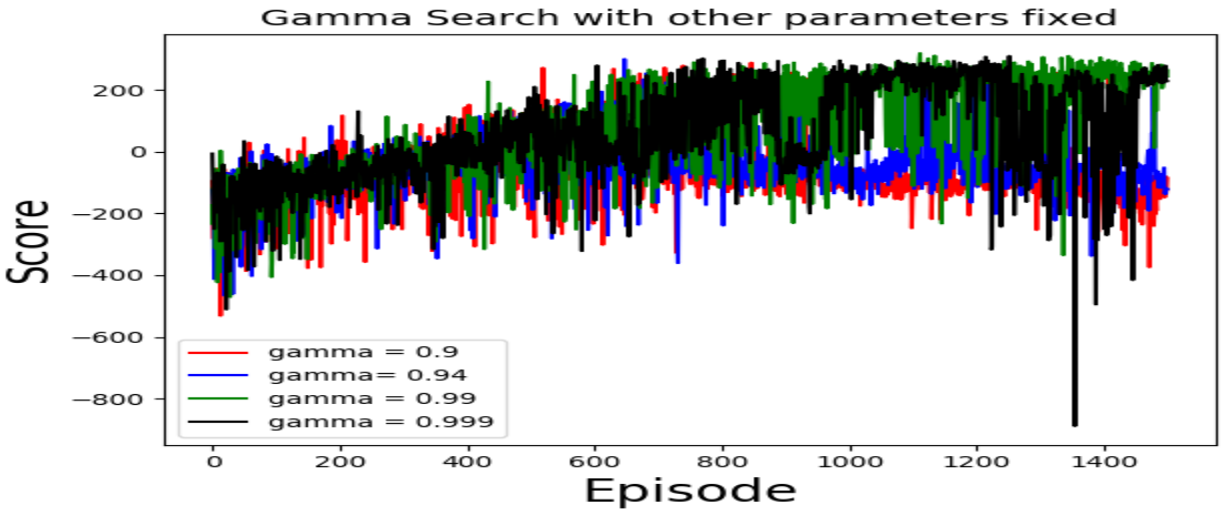
The figure shows the score for four learning rate α (0.001, 0.0005, 0.0003, 0.0001) as the episode increases.



Learning rate plays a key role in updating weights every step. As the learning rate decreases, the curve has better performance and becomes more fatten. Learning rate α = 0.0001 has the best performance because the score curve has the highest score at the end, and it is also the smoothest one with least fluctuation. Smaller learning rate can help update the network weights more accurately such that every step moves the weight to right target steady. Larger learning rate may update the weight too much every step such that it moves away from the right target.

Discounted factor λ Effect

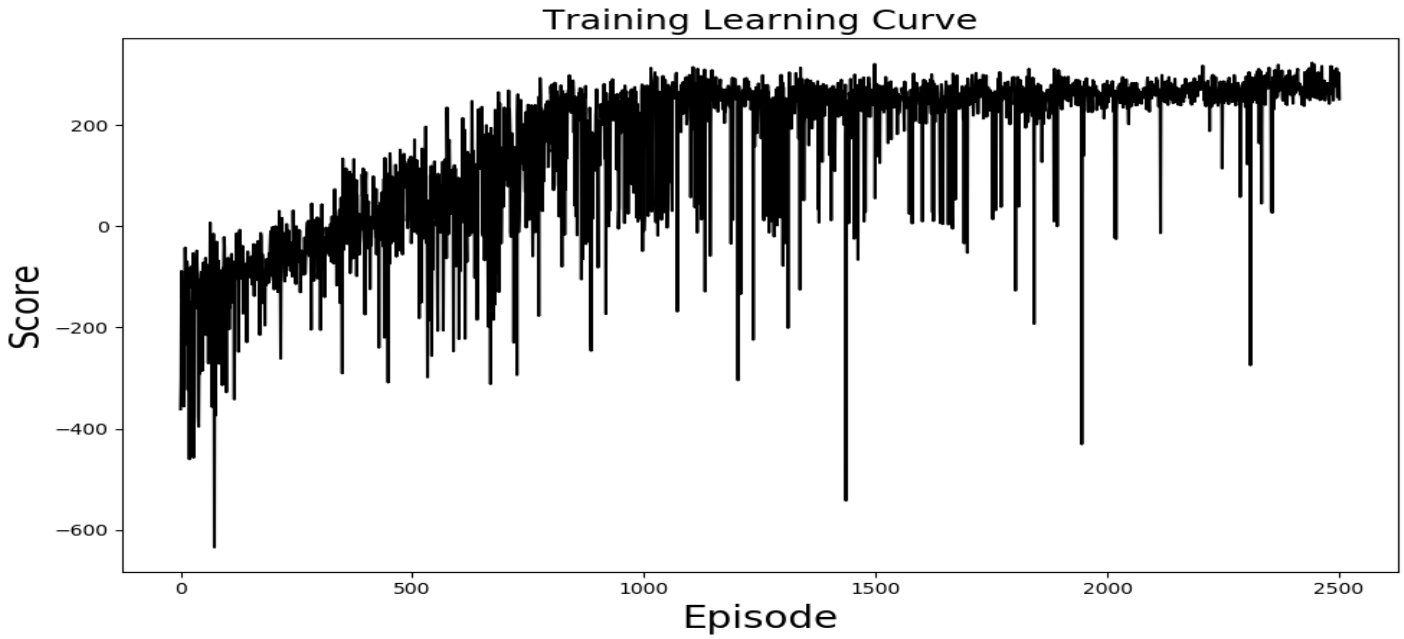
The figure shows the score for four discounted rate ϒ (0.9, 0.94, 0.99, 0.999) as the episode increases.



Discounted rate plays a key role in balancing the current reward and future reward. As the discounted rate increases, future reward is considered more. Smaller discounted rate will make the agent hovering since future rewards with correct actions are not favored enough. That's why the figure above shows gamma = 0.99 has better performance than gamma = 0.9 and 0.94. However, a larger gamma = 0.999 does not always have better performance since it become worse and unsteady later as the episode increases.

Episode training reward

The training process has 2500 episodes with learning rate $\alpha = 0.0001$ and discounted rate $\Upsilon = 0.99$.



The score starts from around -200, gradually increases to 0 at 500 episodes, then reaches 200 after episodes 1000. Then it stays at around 270 from 1000 to 2500 episodes. Although some lanner crashes happen as the near zero or very negative scores shows, the score becomes more stable as episodes increases. At the beginning of training period, the huge state-action space is not explored enough such that a lot of random actions are taken. The performance is very bad. As the episode increases, more state-action space is explored and agent begins to build the network model and acts on the predicted action from the model such that performance increases. At the end, no random actions are taken and all actions are from the more accurate network model. So, the performance is much better and reaches a very steady 270 score.

100 trails learning reward

At the end of the training process, the final model's weights are saved in a separate file (lunar-lander_solved.h5). Now we want to use the trained agent to test the performance. The model weights are loaded into DDQN directly, the agent model learning and experience replay process are also ignored. The epsilon is set to 0 such that agent only use the trained model to predict next action. The final result is shown below:

As the figure shows, the average score over 100 consecutive trials is around 277. So, lunar lander problem is solved since it achieves a score of 200 or higher on average 100 consecutive runs. Some trials have less than 200 points score, where 6 trials has around 175 – 180 points. Those trails may indicate lunar lander comes to rest with two legs on ground but not in the landing pad area. Overall, no lander crashes happen. DDQN does a good job.

Discussion

This project trains the lunar lander agent to land on the landing pad successfully by using DDQN with experience replay and epsilon-greedy technique. However, there are still many places to be improved if more time allowed since the training process take more than 1500episodes now. First, a grid search process can be done to tune the DDQN network such that a better hidden layer structure, loss function and activation can be found. Second, replay batch size and epsilon decay rate can be tuned further. Third, learning rate, discounted factor, batch size, epsilon decay can be combined and considered in the grid search instead of considering them separately. Moreover, a prioritized experience replay can be implemented such that more important experience will be sampled with higher probability.

Reference

[1] Deep Reinforcement Learning Course (https://simoninithomas.github.io/Deep_reinforcement_learning_Course/)

[2] Implementing and training Google's Double DQN AI (https://davidsanwald.github.io/2016/12/11/Double-DQN-interfacing-OpenAi-Gym.html)

[3] Deep Reinforcement Learning with Double Q-Learning, Hado van Hasselt , Arthur Guez, and David Silver, AAAI 2016