AI Agent for Lunar Lander v2

Prabhjot Singh Rai (prabhjot), Amey Naik (ameynaik), and Abhishek Bharani (abharani)

1 Introduction to choice of model and algorithm

Since no data is provided as in supervised or unsupervised learning, there is only an environment which the agent can investigate to collect data. At any time frame, the lunar lander can be imagined at a state defined by eight features with four discrete actions available. Hence, due to above reasons and uncertainty after taking a discrete action, this problem can be modelled as a Markov Decision Process(MDP) with unknown rewards and transition probabilities, which can be targeted using Reinforcement Learning algorithms.

Since this problem involves high dimension and continuous state space, standard Q-learning cannot solve this problem unless some amount of discretisation is done. Due to above mentioned difficulties, Deep Q-Network (DQN) was the choice and we wanted to explore different variants of DQN (particularly DQN with prioritize replay, Double DQN, and Dueling network architecture).

2 Environment and States Description

2.1 Terrain

The terrain is a combination of 10 points, and the helipad(landing zone) is fixed between 5th and 6th points towards the center. The values of the height of the landing zone(5th and 6th points on the terrain) are viewport height divided by 4, and the rest of the points are randomly sampled between 0 to H/2 using numpy random and smoothened (averaging 3 continuous points).

2.2 Initial State

The state is 8 dimensional values of different parameters of the lunar lander at any given time. The starting state is randomly initialised (the lunar lander takes a step in the world through the "idle" action) with certain bounds based on the environment.

Inspecting code of open_ai lunar lander, we see that the initial states are defined by simulating the environment in one frame (calling box world simulation using time step as $1/FPS$). Using this simulation, elaborating the initial state as follows:

1. Position X (Initial Position X: final x which changed from half of viewport width to a value after taking "idle" action before the simulation)
2. Position Y (Initial Position Y: final y which changed from from the viewport height to a value after taking "idle" action before simulation)
3. Velocity X (Initial Velocity X: final velocity x changed from 0 to a value after taking "idle" action before simulation)
4. Velocity Y (Initial Velocity Y: final velocity y which changed from 0 to a value after taking "idle" action before simulation)
5. Current lander angle (Initial lander angle: final lander angle after simulation on "idle" action from 0 degrees)

6. Angular velocity (Initial angular velocity: final angular velocity after simulation on "idle" action from 0 angular velocity)
7. Left leg contacted the surface (Initial value: False, since there's is no probability that the lunar lander's leg will touch the moon surface when at the top)
8. Right leg contacted the surface (Initial value: False)

2.3 End State

The episode ends in the following scenarios:

1. When the lunar lander goes outside of the viewport bounds, the game is over with -100 is negative reward.
2. When the lunar lander touches the ground with a high velocity
3. When the lunar lander touches the ground with body part except the legs
4. When the lunar lander stabilises on the moon's surface (change in shape of lunar lander is constantly 0 for a number of frames)

2.4 Rewards

Before defining the rewards, let's define the shape of the lunar lander which decides the rewards. The shape of the lunar lander is a function of position coordinates (x, y), linear velocities (v_x, v_y), lander angle θ and contact of both the lander legs. We are interested in finding the change of shape at every step for the lunar lander to calculate the rewards for each given action. Shape change is given by subtraction of previous shape and current shape. Formally:

$$\begin{aligned}\text{shaping} = &-100 * (x^2 + y^2) \\ &-100 * (v_x^2 + v_y^2) \\ &-100 * abs(\theta) + 10 * (\text{Left leg contacted}) + 10 * (\text{Right leg contacted})\end{aligned}$$

The rewards are defined as follows:

1. If the lunar lander crashes, or goes out of the bounds: -100
2. If the lunar lander is not awake anymore (stabilises at 0 shape change): $+100$
3. Firing the engine: shape change

3 Goal

Obtain a policy that once followed by agent made it capable of landing a space vehicle into landing pad region with speed close to 0(soft landing). Rewards is a combination of how much is the speed of lander (close to 0) , how close is the landing pad, every time a we fire engine there is negative reward of 0.3 per frame. The state space is a 8 dimensional and number of actions we can take are 4 [do noting, fire left, fire right, fire main engine].

We are proposing to use Q-learning to solve this descrete state space problem. We will be implementing different variants of Deep Q Network (DQN) to predict the actions given current state.

4 Model

4.1 Q-Learning

Q-learning learns action-reward function Q(s,a): determines how good to take an action in a particular state. In Q-learning we build memory table Q[s,a] to store Q-values for all possible combinations of s and a. We sample an action from the current state to find out reward and new state. From the memory table, we determine the next action to take which has maximum Q(s,a).

Algorithm:

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, ... till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha) Q_k(s, a) + \alpha \, [\text{target}]$$

$$s \leftarrow s'$$

Fig. 1. Q-Learning Algorithm

4.2 DQN

The number of actions we can take from current state is large and we need to observe each action space to solve this problem. We will be using Deep Q Network (DQN) to find Q(s,a). However while exploring each state space the Q value(label) will be changing each time and we will be updating model parameters to update based on new Q value each time. The newly Q value will be higher at the same time the target Q value will be move higher making it difficult for algorithm to optimize. To solve this challenges we can slow down the changing Q value using Experience replay and Target network.

We will train the neural network on subset of transitions into a buffer. From this buffer we will sample mini batch which will be stable for training. We will be buiding two neural network one to retrieve Q values while second one is to update the Q value. After a fixed intervals we will synchronize the parameters.

4.3 Loss function

DQN Uses huber loss where loss is quadratic for small values of a and linear for larger values.

4.4 Architecture

Input to the DQN network is compressed video frames of 84 x 84 pixels followed by fully connected layers to compute Q value for each action.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

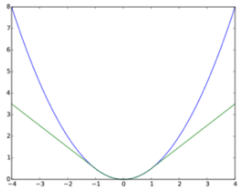
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Fig. 2. DQN with Experience Replay



Green is the Huber loss and blue is the quadratic loss (Wikipedia)

Fig. 3. DQN loss function

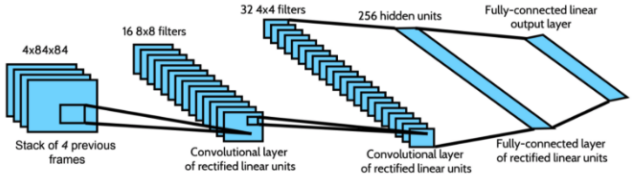


Fig. 4. DQN Architecture

Fig. 5. Initial Experimental Results

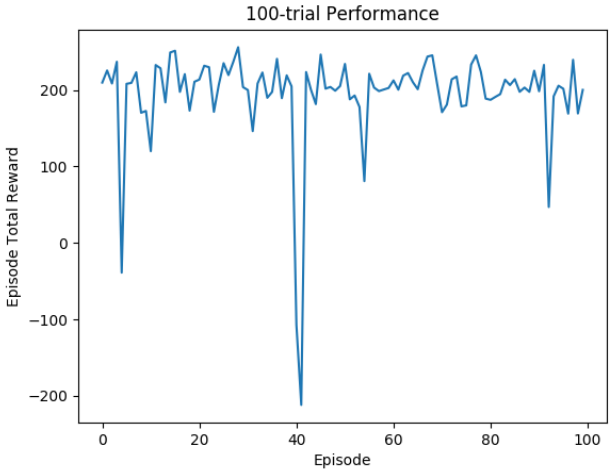


Fig. 6. Rewards per Episode

4.5 Experiments and Evaluation

5 Future Work

For future work, we are planning on applying different Improvements to DQN networks.

6 Contributions

All of us contributed in the discussions about what problem to target, and what techniques to apply. All of us helped with writing and reviewing the report.