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

A reinforcement learning environment for lunar-lander game is provided by open AI gym. This environment provides an interface to test and probe. At any time frame, the lunar lander can be imagined at a state defined by eight features with four discrete actions available. Due to uncertainty after taking a discrete action, this problem is modeled as a Markov Decision Process (MDP) where in, moving from one state to another gives us a reward. We want to choose an optimal policy at each state with maximizes the cumulative, long-term reward.

Since this problem involves high dimension and continuous state space, standard Q-learning cannot solve this problem unless some amount of discretization is done. Due to above mentioned difficulties, Deep Q-Network (DQN) was the choice. We plan 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 initialized (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 and Transitions

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, shaping at time frame t:

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

$$\text{shape change} = \text{shaping}_t - \text{shaping}_{t-1}$$

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. Doing nothing: shape change
4. Firing the engine: shape change - 0.3
5. Rotating: shape change - 0.03

The total reward will automatically be a sum of all the rewards at each time frame, and if the lander touches the ground with it's legs, will add those rewards to the total rewards earned during an episode. Transition probabilities are unknown, we get next states by simulating the lunar lander in the box environment given the current state and action taken.

Note: The transition probabilities and rewards are unknown to our agent, which it will try to figure out through exploration and incorporate in learning.

3 Model Description

Our model is Deep Reinforcement Learning, and the algorithms are variants of Deep Q-Network (DQN). DQN is a model-free learning agent.

3.1 Framework

Basic framework is defined in Fig 1. The agent performs action(a) on the environment. Performing the action on the environment returns reward(r) and new state s'. Our agent then incorporates this information.

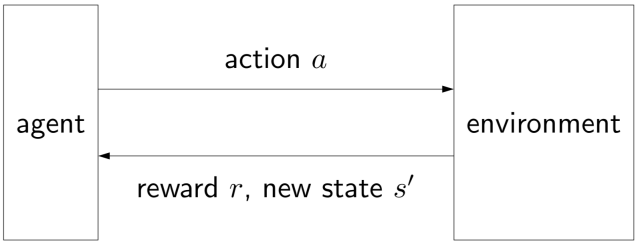

Fig. 1. Reinforcement Learning framework

In such a model, for our problem, here is quick description of notations:

1. S is defined as all possible states, s is one particular 8 dimensional state
2. A is defined as all possible actions, a is one of the actions out of [idle, fire engine, rotate left, rotate right]
3. R is the reward distribution given (s, a)
4. P is the set of possible transitions and their probabilities given (s, a)
5. γ: the discount factor. How much we want our agent to discount future reward. It is a hyperparameter that we define ourselves.

3.2 Goal

Our model builds off of Q-learning algorithms by using a Deep Neural Network (DNN) for approximating the state-action Q-value, Q(s, a).

Given the state s, our goal is to identify a policy π_{opt} that maps the states to actions in order to maximize the total reward we get.

$$\text{S} \rightarrow \text{A based on } \pi_{opt}$$

As we know, Q value is defined as the expected reward that we get following action a in a given state s and then following the policy π, our objective is to define a $Q_{opt}(s, a)$, which can be maximised over all possible actions at a state s, in order to find the π_{opt}.

4 Algorithm Description

4.1 Q-Learning

Q-learning learns action-reward function Q(s,a): determines best action to take in a particular state. In Q-learning we build memory table Q[s,a] to store Q-values for all combinations a, as and when we encounter state s. 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. 2. 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). We will be updating model parameters for a mini-batch. To solve this challenge 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.

4.3 Architecture

Input to the neural network used in DQN is a 8 dimensional vector followed by two dense layers and then by fully connected layers to compute Q value for each of the 4 actions.

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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

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

  End For

End For

Fig. 3. DQN with Experience Replay

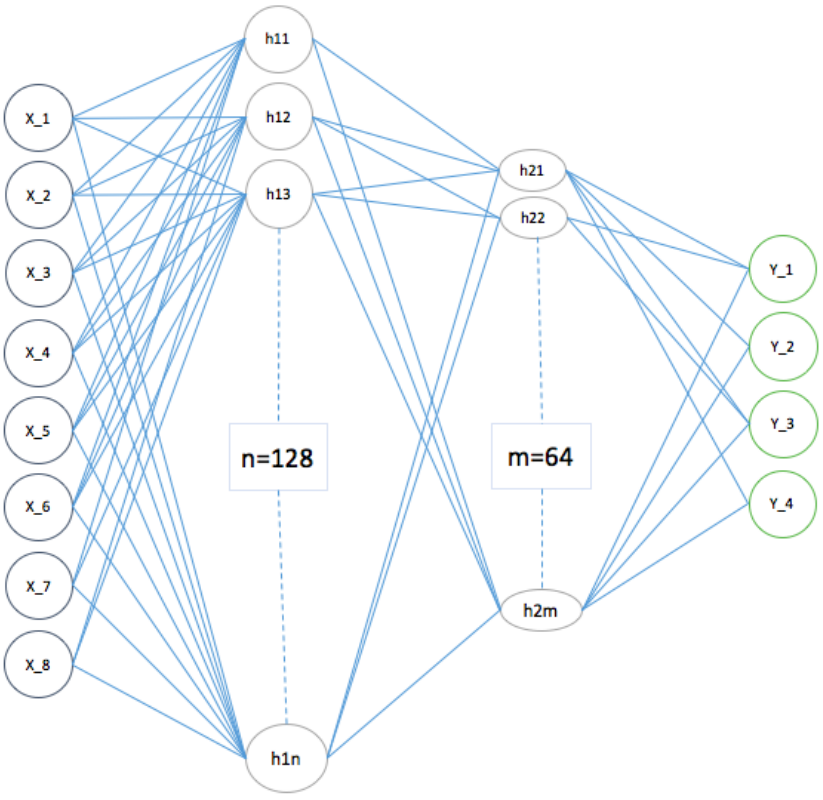

Fig. 4. Neural Network The Deep Q neural network topology, consisting of 8 input nodes, 128-node hidden layer, a 64-node hidden layer, and 4 output nodes.

4.4 Experiments and Evaluation

Based on our initial experiments, we got an average reward of close to 200 consistently after about 2000 episodes. In future, we will try to do more feature engineering and try out variants of DQN network to reduce the learning time.

Fig. 5. Initial Experimental Results

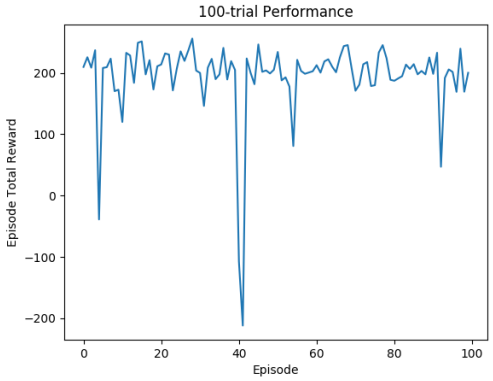

Fig. 6. Rewards per Episode