AI Agent for Lunar Lander v2

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

1 Introduction

The purpose of this project is to build an AI agent to play Lunar Lander game to safely land on a landing pad. We will utilize different variants of DQN such as Double DQNDoubleQ-learning, DQN with prioritize replayPrioritizedReplay , Dueling network architecture Dueling, continuos action space Continuos to solve this problem.Our goal is to obtain a policy that once followed by the agent, makes it capable of obtaining scores comparable to human playing the game (Oracle). We used reinforcement learning techniques to train an AI agent to play Lunar Lander v2. Current research in the field includes using the images for states and deep reinforcement learning to train the AI agent. This approach will generally have the pixels of each image and the game score as inputs.

2 Problem Statement

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

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.

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. 2. DQN with Experience Replay

4.3 Loss function

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

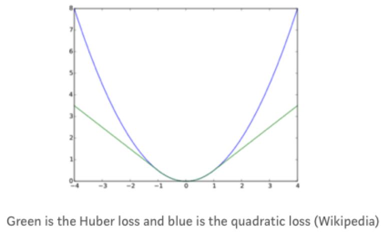

Fig. 3. DQN loss function

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.

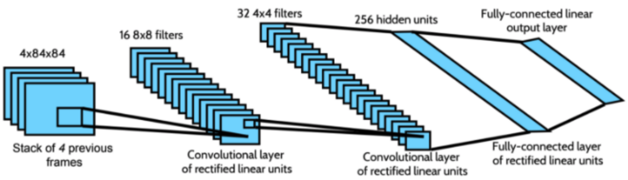

Fig. 4. DQN Architecture

4.5 Experiments and Evaluation

Fig. 5. Initial Experimental Results

5 Future Work

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

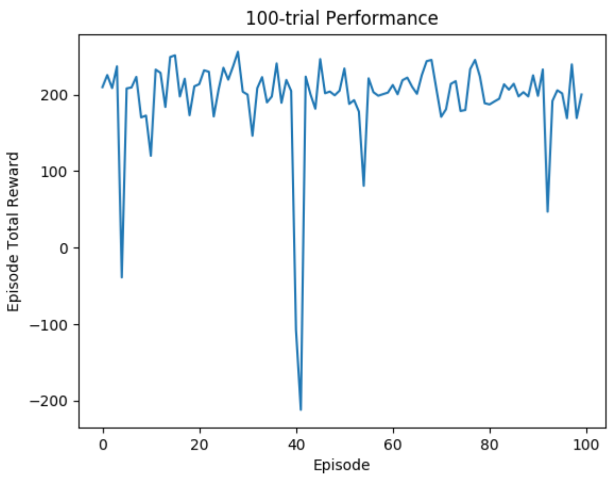

Fig. 6. Rewards per Episode

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.