AI Agent for Lunar Lander v2

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

1 Introduction

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.

1.1 Goal

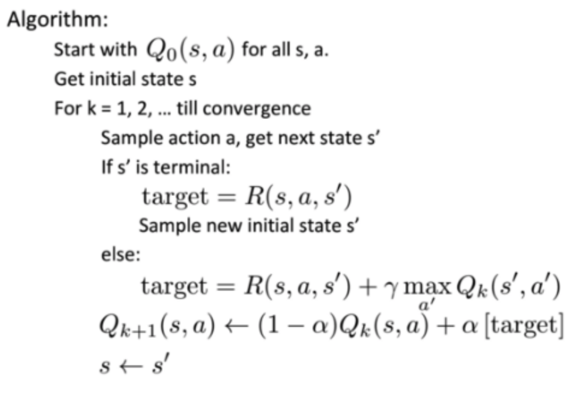
Obtain a policy that once followed by agent made it capable of landing a space vehicle into landing pad region with speed close to 0(soft landing). Rewards is a combination of how much is the speed of lander (close to 0) , how close is the landing pad, every time a we fire engine there is negative reward 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.

2 Model

2.1 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).



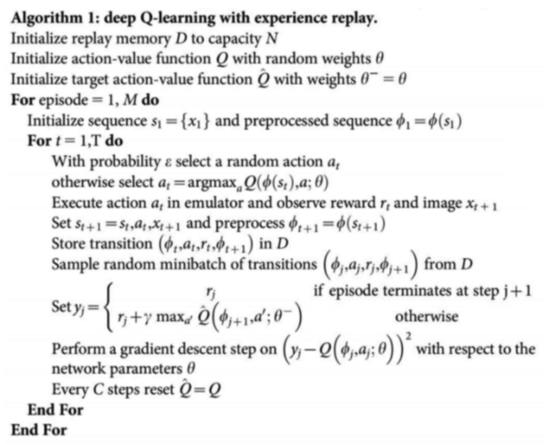
```
Algorithm:
    Start with $Q_0(s,a)$ for all s, a.
    Get initial state s
    For k = 1, 2, … till convergence
        Sample action a, get next state s'
        If s' is terminal:
            $\text{target} = R(s,a,s')$
            Sample new initial state s'
        else:
            $\text{target} = R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$
        $Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\,[\text{target}]$
        $s \leftarrow s'$
```

Fig. 1. Q-Learning Algorithm

2.2 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.

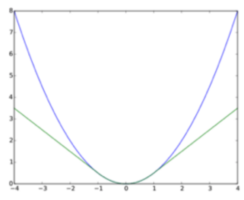


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
     With probability $\varepsilon$ 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 $\theta$
     Every C steps reset $\hat{Q} = Q$
  End For
End For
```

Fig. 2. DQN with Experience Replay

2.3 Loss function

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



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

Fig. 3. DQN loss function

2.4 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.

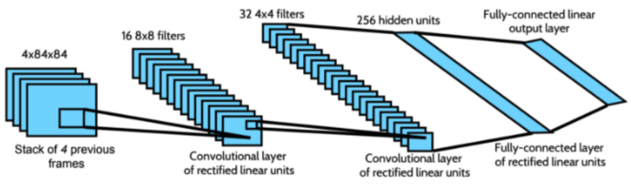

4x84x84
16 8x8 filters
32 4x4 filters
256 hidden units
Fully-connected linear output layer
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Fig. 4. DQN Architecture

2.5 Experiments and Evaluation

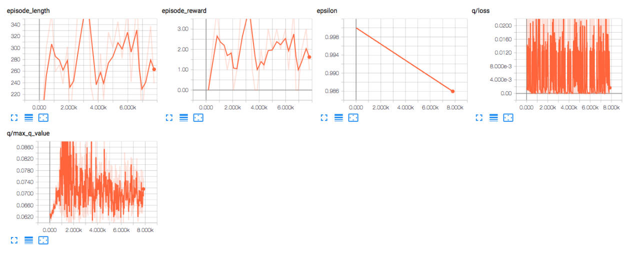


Fig. 5. Initial Experimental Results

3 Future Work

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

4 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.