CSE7642 Project: Lunar lander Using Deep Q-Networks

Shaikh Shamid

Department of Computer Science, Georgia Institute of Technology

March 19, 2018

Abstract

This report presents the pitfalls of programing a reinforcement learning agent to land the Lunar Lander that is implemented in OpenAI gym. We used a non-linear function approximation solutions where the policy is parameterized by a weight vector, instead of tabular solutions to solve the environment.

1 Lunar Lander Environment

The environment consists of a 8-dimensional continuous state space and a discrete action space. There are four discrete actions available: do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine. The landing pad is always at coordinates (0,0). Coordinates consist of the first two numbers in the state vector. The total reward for moving from the top of the screen to landing pad ranges from 100 - 140 points varying on lander placement on the pad. If lander moves away from landing pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points respectively. Each leg ground contact is worth +10 points. Firing main engine incurs a -0.3 point penalty for each occurrence. Landing outside of the landing pad is possible. Fuel is infinite, so, an agent could learn to fly and then land on its first attempt. The problem is considered solved when achieving a score of 200 points or higher.

At each time step, the state is provided to the agent as a 8-tuple: (x, y, v_x, v_y,θ, v_θ, left-leg, right-leg) where x and y are the x and y-coordinates of the lunar lander's position. v_x and v_y are the lunar lander's velocity components on the x and y axes, θ is the angle of the lunar lander, v_θ is the angular velocity of the lander. Finally, left-leg and right-leg are binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground. So, there are six dimensional continuous state space with the addition of two more discrete variables.

2 Theory of Deep Q-Networks

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. When the state and action spaces are small enough for the approximate value functions to be represented as arrays, or tables, as in tabular Q-learning

1 Start with $Q_0(s, a)$ for all s,a.
2 Get initial state s
3 **for** $i \leftarrow 1, 2, ...$ *till convergence* **do**
4 sample action a
5 get next state $s^{'}$
6 **if** $s^{'}$ *is terminal* **then**
7 target=$R(s, a, s^{'})$
8 **end**
9 **else**
10 target=$R(s, a, s^{'}) + \gamma\ maxQ_i(s^{'}, a^{'})$
11 **end**
12 $Q_{i+1}(s, a) = (1 - \alpha)Q_i(s, a) + \alpha$ target
13 $s = s^{'}$
14 **end**

Algorithm 1: Tabular Q-learning

In this case, the methods can often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy. This contrasts with the approximate methods, which only find approximate solutions, but which in return can be applied to arbitrarily large state spaces.The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation (line 12 in algorithm 1) as an iterative update. Such value iteration algorithms converge to the optimal action-value function. In a sense, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function, In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. The idea is to make Q-learning look like supervised learning. Mnih et. al [**?**] introduce two main ideas for stabilizing Q-learning. First, applying q-updates on batches of past experience instead of online called experience replay which in turn makes the data distribution more stationary. Second, using older set of weights to compute the target network.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for *episode* $\leftarrow 1$ **to** M **do**
    Initialize sequence $s1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t \leftarrow 1$ **to** T **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = max_a Q(\phi(s_t), a; \theta)$
        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 pre-process $\phi_{t+1} = \phi(s_1)$
        Store transition $\phi_t, a_t, r_t, \phi_{t+1}$ in D
        Sample random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        set $y_j = r_j$ if terminal ϕ_{j+1} else $r_j + \gamma max_{a'} Q(\phi_{j+1}, a^{'}; \theta)$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end
end

Algorithm 2: Deep Q-network with experience replay

3 Implementation

3.1 Implementing Neural Network using Keras

In this section, we describe how the neural network is used in the DQN algorithm. Keras makes it really simple to implement a basic neural network model. First we tried a network that receives input dimension equal to the lunar lander state size (8) and 2 hidden layers of 24 and 8 neurons with relu activation function. For the output layer we use number of neurons equal to the lunar lander action space size (4) with linear activation functions. Then we tried a bigger network of dimension 8-24-24-4 for better performance. We use mean square error as loss function and Adam optimizer for compiling the neural network model.

3.2 Implementing DQN with experience replay

Using algorithm 2 we first initialize and populate the replay memory using 100k state-action transitions, then run 1000 episodes each with 1000 time steps. At each time step, if we find the memory is full we remove an example to be able to store a new state-action-reward experience. We update the NN weights using batch samples pulled from the replay memory. Because of the efficient implementation of this experience replay the program takes about an hour to finish.

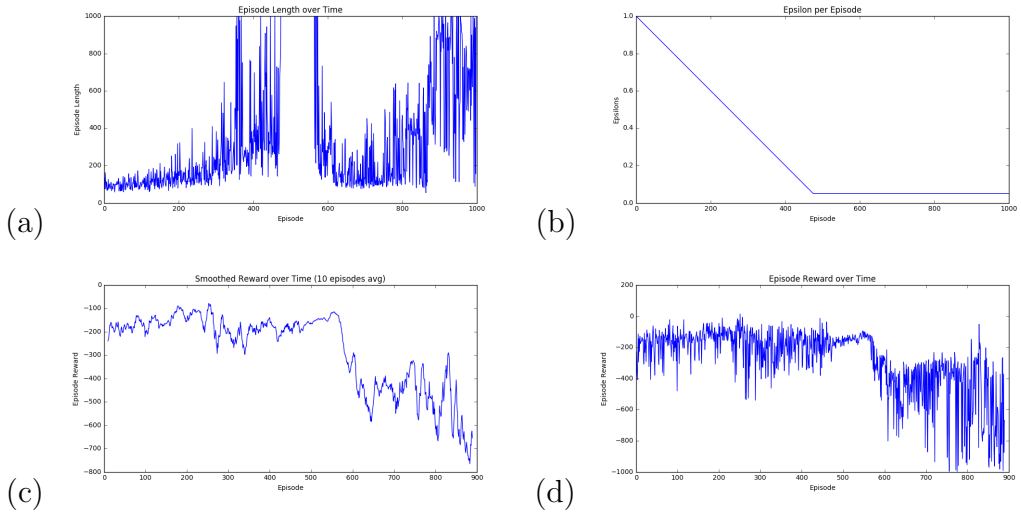

Figure 1: Plot for 100k state-action transitions, batch size 256 discount factor γ equal to 0.999, 8-24-8-4 Network with learning rate 0.001 (a) Time step taken for each episode (b) Linear epsilon decay over episode (c) Smoothed reward (average over 10 episode) for each episode (d) Reward for each episode

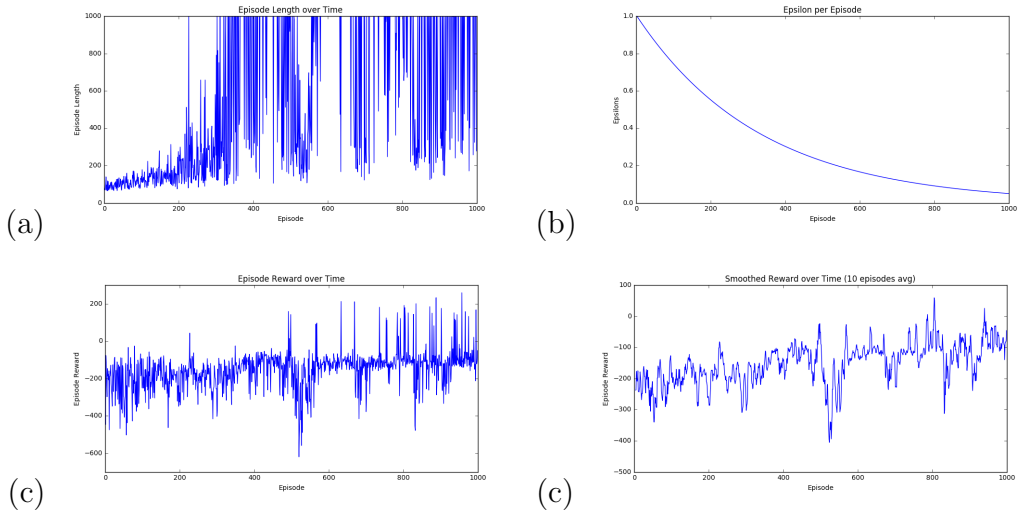

Figure 2: Plot for 100k state-action transitions, batch size 512 discount factor γ equal to 0.95, 8-24-24-4 Network with learning rate 0.00025. (a) Time step taken for each episode(b) Exponential epsilon decay over episode (c) Smoothed reward (average over 10 episode) for each episode (d) Reward for each episode

Figure 3: Plot for the reward per trial for 100 trials using the trained agent

4 Discussion

We believe that we successfully program a reinforcement learning agent to land the Lunar Lander that is implemented in OpenAI gym using a non-linear function approximation solutions. But it was important to find the right hyper-parameter set for the deep Q-network to be able to solve the lunar lander environment. First, we observe that the agent is using up all the 1000 time steps as shown in fig. 1 (a), but not landing/crashing into the ground. We then implement exponential decay of exploration rate (ϵ) and/or increase NN training batch size from 256 to 512, but could not solve this issue, and obviously the reward plots in 1 (a) and (b) show that the algorithm does not converge in case of linear exploration rate decay. But it looks to me the algorithm is going to converge in case of exponential decay with appropriate learning rate and sufficient number of episodes. Ideally we want small learning rate values to avoid overshooting, but it is tricky to choose one such that it does not get stuck in local minima or take too long to descend. For figure 1 we use learning rate equal to 0.001, and we observe that the algorithm get stuck in several different local mimina. For figure 2 we use lower learning rate of 0.00025, and see that the reward is improving slowly but it does not get stuck. Although, the agent is using up all the 1000 time steps. Probably, learning rate decay/annealing over the course of training would be better because early on there is a clear learning signal so aggressive updates encourage exploration while later on the smaller learning rates allow for more delicate exploitation of local error surface. Using this training agent we calculate the reward and monitor the environment, as shown in fig 3 sometime it crosses 200 rewards just like while training. We also confirm this watching those videos that the agent is successfully landing on the landing pad for those high reward trials.

In future, we will run the program for 2000 episodes to continue see the trend of increasing reward, and eventually converse the algorithm.