HARIKRISHNAN KC
harikrishnan3@gatech.edu

Project 2

Lunar Lander

Overview

The project involves building a learning agent to solve the LunarLander-v2 environment provided by OpenAI [1]. The agent must navigate a LunarLander to land on a landing pad without crashing. Our interaction with the environment is through an 8-dimensional continuous state space and a set of discrete action space. The Lunar Lander model has an 8-dimensional state space which is provided as a tuple represented below:

(x, y, vx, vy, θ, vθ, left-leg, right-leg)

Where x and y are the x and y-coordinates of the Lunar Lander's position, vx and vy are the lunar lander's velocity components on the x and y axes, θ is the angle of the lunar lander and vθ is the angular velocity of the lander. The 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.

At each timestep, we select one of the four actions to executed on the model. The coordinate of the landing pad is always (0,0). The total reward ranges from 100 to 140, for moving from top of the screen to landing pad. A penalty of -100 points is given if the lander crashes and the episode finish. The other scenario where the episode finish is if the lander comes to rest. Touching any of the legs of the lander on the ground gets a reward of +10 points. Whenever main engine is fired a penalty of -0.3 points is incurred. The Lunar Lander can land outside of the landing pad. Accumulating a score of 200 points or more solves the problem.

To solve a sequential decision problem, we can use the Q learning algorithm. Under a given policy π a true value of an action *a* in state *s* is

$$Q_\pi(s,a) \equiv \mathbb{E}\left[R_1 + \gamma R_2 + \ldots \mid S_0 = s, A_0 = a, \pi\right]$$

The optimal value is then given by $Q_*(s,a) = \max_\pi Q_\pi(s,a)$

If the state space is very huge it is not easy to completely learn all the Q value pairs. In such scenarios, we can use a parametrized function optimizer to generate the Q values. Such a optimizer can be represented as below $Q(s, a; \theta_t)$. For our experiment we have used a deep neural network-based function optimizer (Deep Q network). A deep Q network (DQN) is a multi-layered neural network that for a given state *s* outputs a vector of action values $Q(s, a\cdot; \theta)$, where θ are the parameters of the network. [2] proposed two important ingredients of DQN algorithm namely using a target network and experience replay. An experienced replay is a memory buffer, which stores the state, action, reward, next state tuples. This is useful to solve the problem of correlation between consecutive states. In DQN we basically use two neural networks, one called an action network the other called a target network. Our basic objective is to learn the parameters theta or weights of the DQN networks such that it can provide the optimal policy. During learning we

randomly sample mini batches from the experience replay buffer and apply this to our action network. The Q-learning update at iteration *i* uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i) \right)^2 \right]$$

Here, gamma is the discount factor, θ_i are the parameters of the action and θ_i^- the parameters of the target network. The target network parameters are updated to action network parameters every C steps and are held fixed between individual updates.

Experimental Setup

For performing the experiment, I created a Python class which mimics the algorithm given in [3] under Algorithm 1 section. I tried both DQN and Double DQN(DDQN) [4] to solve the problem. In DQN we use the max Q value from the target network during the update stage. In DDQN we use the action which corresponds to the max. Q value from the action network to select the Q value for the target network during update stage. I decided to finally stick with DDQN based approach since it gave better results.

For both action and target network I created a deep neural network with 2 hidden layers as shown in the code below. The network input was the 8-input tuple from the LunarLander environment and the outputs are set to the four action values of the LunarLander environment. The hidden layers use rectified linear activation functions while the final layer uses linear activation function. I used an Adam optimizer with a learning rate of 0.00025

```
def create_nn_model(self):
    model = Sequential()
    model.add(Flatten(input_shape=(1,8)))
    model.add(Dense(64))
    model.add(Activation('relu'))
    model.add(Dense(64))
    model.add(Activation('relu'))
    model.add(Dense(self.output_size))
    model.add(Activation('linear'))
    m_opt = Adam(lr=0.00025)
    model.compile(optimizer=m_opt,
        loss='mean_squared_error')
    model.summary()
    return model
```

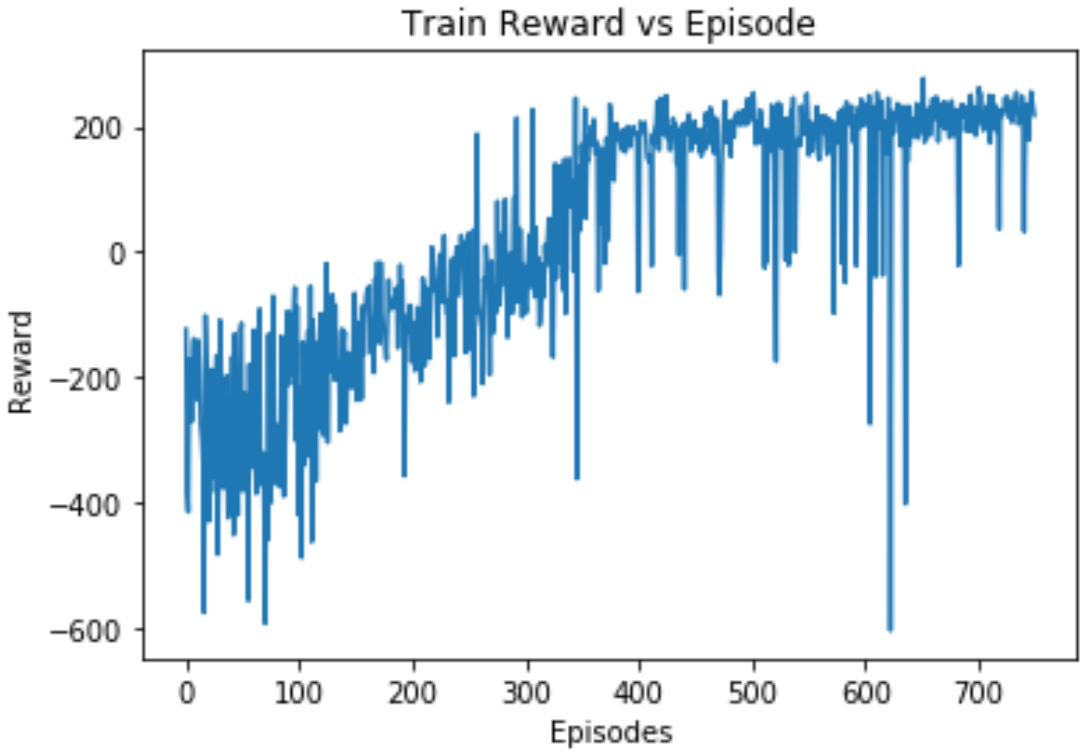
I followed [3] to select most of the hyper parameters for this problem. For experience replay buffer I used a buffer size of 50,000. I used the same mini batch size as 32 from [3]. The weights of the target network were updated to action network weights every 500 steps. Discount factor was set at 0.99. The learning rate of the neural network was kept at 0.00025. The Neural Network starts learning once the Replay buffer had at least 10,000 samples.

HARIKRISHNAN KC
harikrishnan3@gatech.edu

Experiment 1

Reward for each training episode while training your agent

The agent was run for a total of 750 episodes. By this time, the agent was able to consistently converge to values greater than 200. The plot below shows the relationship between episode and rewards observed during each episode. During the initial episodes, the agent is doing more random actions. This results in mostly negative rewards. As the agent begins more greedy approach, the rewards begin to increase and finally reaches above 200 points.



Experiment 2

Reward per trial for 100 trials using your trained agent

The plot below shows the reward obtained over 100 trials using the learned agent. This was run using the agent trained from the previous experiment. I also plotted the moving average of the reward for all the 100 episodes. As observed from the plot below, the agent was able to consistely obtain rewards greater than 200 in most episodes. The moving average of the rewards stayed near 220. This concludes that the DDQN agent was able to successfully learn the weights, to predict be best action to take to maximise the rewards.

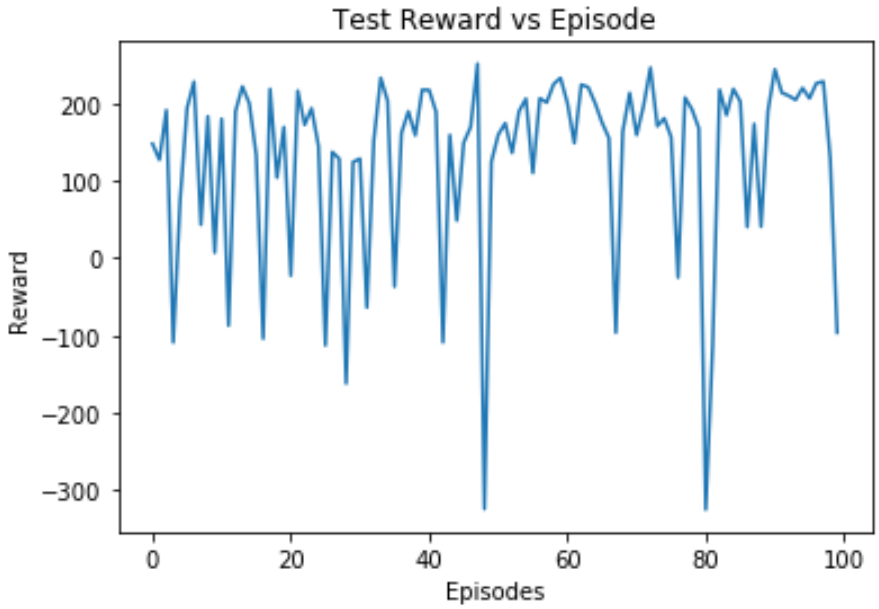
HARIKRISHNAN KC
harikrishnan3@gatech.edu

Experiment 3

The effect of hyper-parameters (alpha, lambda, epsilon) on your agent

1. Learning rate alpha

I decided to use the same learning rate (0.00025) mentioned in [3] for running the experiments. Learning rate is the rate at which the weights are updated. I tried experimenting with a larger learning rate (0.001). The plot generated by this learning rate is shown below. Increasing the learning rate seems to have had an adverse effect. This may be because the NN model weight updates are causing the model to diverge.



2. Discount factor lambda

I decided to use the same discount factor (0.99) from [3] for running the experiments. Discount factor is the rate which future rewards need to be discounted. The basic motivation is to discount future rewards such that the agent tries to solve the problem faster. I also tried running an experiment with a smaller discount factor (0.90). What I observed was that it took the agent much larger time to run each episode. It was trying to run more steps during each episode (most episodes ran till 1000 steps). This is understandable since, the Q value updates are now getting propagated more slowly. The agent was unable to converge to values greater than 200. This suggests that it is important to select proper values for discount factor. The value of 0.99 seems like a very good choice.

3. Epsilon

Epsilon decides the way in which we decide to choose an action to act on the environment. For my experiment I decided to start with an epsilon value of 1.0. After this, the epsilon value was decayed every episode by a value of 0.0995 to ensure that greedy policy gets more weightage as the episode progresses. The agent initially needs to have a larger epsilon value, so that it can explore the environment more. As the episodes pass, we need to shift more towards a greedy policy.

Conclusion

I was able to implement the algorithm mentioned in papers [2], [3] and [4]. The learning agent was able to consistently reach a reward value above 200. I was able to learn how function optimizers can be used to predict Q values, and how to train it to explore the state space. I was also able to learn about DQN, DDQN and Experience replay techniques. This project opened me to the possibly of combing Deep learning and Reinforcement learning to effectively solve sequential problems.

References

[1] https://gym.openai.com/envs/LunarLander-v2/
[2] Playing Atari with Deep Reinforcement Learning (http://arxiv.org/abs/1312.5602)
[3] Theoretically-grounded policy advice from multiple teachers in reinforcement learning settings with applications to negative transfer (http://arxiv.org/abs/1509.06461)
[4] Deep Reinforcement Learning with Double Q-learning (http://arxiv.org/abs/1509.06461)
[5] Prioritized Experience Replay (http://arxiv.org/abs/1511.05952)