Report for Udacity Reinforcement Learning Nanodegree - P1 Navigation

Project Setup and Learning Algorithm

This project involves training a RL agent to navigate and collect yellow bananas in a Unity Envtt. The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

- 0 - move forward.
- 1 - move backward.
- 2 - turn left.
- 3 - turn right.

The goal of the agent is to successfully navigate the environment and learn to pick yellow bananas while avoiding blue bananas. The agent gets a reward of +1 for picking a yellow banana and -1 for picking a blue banana. The environment is considered solved when an agent is able to get an average score of +13 over 100 consecutive episodes.

The Deep Q Network(DQN) has been used as the Reinforcement Learning Algorithm. DQN was the first major breakthrough in the use of neural networks as a Q-learning function that was able to play a range of Atari games.
My implementation of this project uses the Agent and the Replay Buffer class provided in the project Deep Q Network. I have modified the neural network used and the functions to train the DQN to apply to this use case.

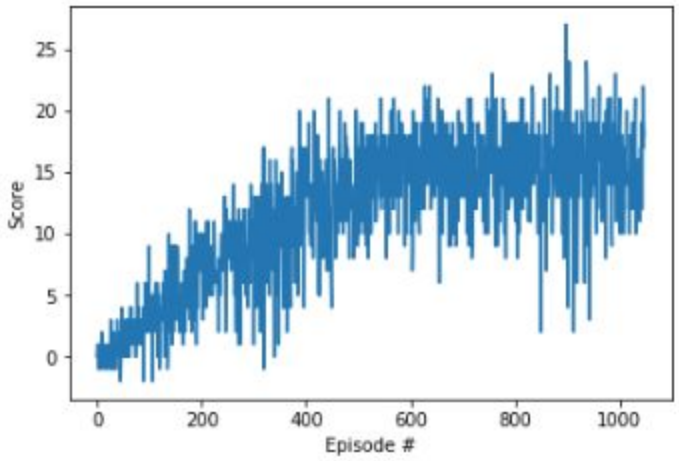
I found good success by using a relatively simple neural network that also happens to be quick to train. The Neural Network consists of three fully connected layers. The first layer has 64 neurons, the second has 32 neurons and the last has 4 neurons. RELU is used as an activation function b/w the first two layers of the neural network. The input into the neural network is the state of the game as described by the 37 dimensions. The output from the neural network is the score for the 4 possible action outcomes.

The mean square loss is used as the loss function to train this neural network. Adam is used as the optimizer with a learning rate of 5e-4. For the agent class, a batch size of 64 is used with a replay buffer size of 1e5. The weights of the target neural network is updated every 4 steps.

Training the Neural Network

The Deep Q-Network was trained for max of 2000 steps or till the reward of 16 was reached. It took the model about 1100 steps to get to a reward of 16. The plot of score by epoch is shared below:

```
Episode 100     Average Score: 1.29
Episode 200     Average Score: 4.83
Episode 300     Average Score: 8.07
Episode 400     Average Score: 9.98
Episode 500     Average Score: 13.25
Episode 600     Average Score: 14.84
Episode 700     Average Score: 15.33
Episode 800     Average Score: 15.08
Episode 900     Average Score: 15.48
Episode 1000    Average Score: 15.77
Episode 1048    Average Score: 16.04
Environment solved in 948 episodes!     Average Score: 16.04
```



Once training was finished, the best weights were saved. I then tested the trained agent on Unity Environment by playing games. The agent was always able to get a score of at least 13 points.

Opportunities for Improvement

As explained in the lecture video, the performance of DQN can be improved by modifying the architecture, training process a bit. Some of these improvements include:

1. Prioritized Experience Replay - The idea behind Prioritized Experience Replay is to sample events from the replay buffer based on importance probabilities instead of randomly. The intuition is that some past experiences that occur infrequently may be more important for learning. Past experiences with higher TD error delta could be given higher priority
2. Dueling DQN - The Dueling DQN extends the standard DQN architecture by including two streams one more predicting the state value and the other for predicting the advantage values. The two streams may share some base layers. The final Q-value is calculated by combining the two streams. The intuition is that the value of many states doesn't vary over time and one of the branches can learn the state values. When the value of state does vary by action, then the advantage stream can estimate this
3. Noisy DQN - As explained in the paper, Noisy DQN adds noise to the weights and finds that this added randomness can be used to improve network exploration. The parameters of the noise are learned with gradient descent along with the remaining network weights.