Project:Navigation

Introduction:

This report provides the details regarding the procedures and techniques implemented to complete the Udacity Deep Reinforcement Learning Nano-degree first project named as "**Navigation".** This project involves using Deep Reinforcement Learning technique to train an agent to be able to navigate itself through the path to collect bananas. A reward of +1 was given to collect yellow bananas and -1 for purple bananas. The controller or the number of actions that an agent was eligible to take were four which are as follows:

1) Move straight
2) Move down
3) Move left
4) Move right

The observation space that was given was the space in which the agent was to collect these bananas. The task was an episodic task where the agent when collect +13.0 rewards was considered to be a successful agent in the window of 100 episodes. The simulation for collecting bananas by the agent was provided by Udacity team.

Model:

The model developed for this agent's training was a Deep Reinforcement Learning model. It uses a deep neural network to train the agent. The deep learning neural network comprises of three linear units and two rectified linear units layers. The first linear unit layer consists of 1024 neurons and the other layer consists of 256 neurons. The Rectified Layer Unit function acts as an activation function in the neural network. The model used gradient descent with ADAM optimizer to update the weights. The training code is as follows:

```
import torch.nn as nn

import torch.nn.functional as F

class QNetwork(nn.Module):

    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed):

        """Initialize parameters and build model.

        Params

            state_size (int): Dimension of each state

            action_size (int): Dimension of each action

            seed (int): Random seed
```

```
        super (QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        fc1_units = 1024
        fc2_units = 256
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
    def forward(self, state):
      """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        #x = F.dropout(x, p=0.0, training=self.training)
        x = F.relu(self.fc2(x))
        #x = F.dropout(x, p=0.0, training=self.training)
        x = self.fc3(x)
        return x
```

Agent:

The main agent that is used to train the algorithm as stated above is based on 3-layered neural network. The Q-Learning method is one of the most famous method in Reinforcement Learning that used policy methods to deduce actions. The major advantage of Q-Learning method over typical SARSA methodology is that it directly learns optimum q-values rather than evaluating between exploitation and exploration. Non-Linear functions suffer from the problem of instability and in order to overcome this problems, there are two modifications that are being inserted into the environment. These two modifications are as follows:

1 Experience Replay:

A typical RL-agent proposes an agent interaction with the environment and get back a (state-action-reward, next-state) learn from it and then discard it. This can be a bit wasteful sometimes as some of the experiences seem to be very important to deduce useful information from it. So, this useful information should be stored in some memory and this memory is called Replay buffer and this phenomenon is termed as Experience Replay.

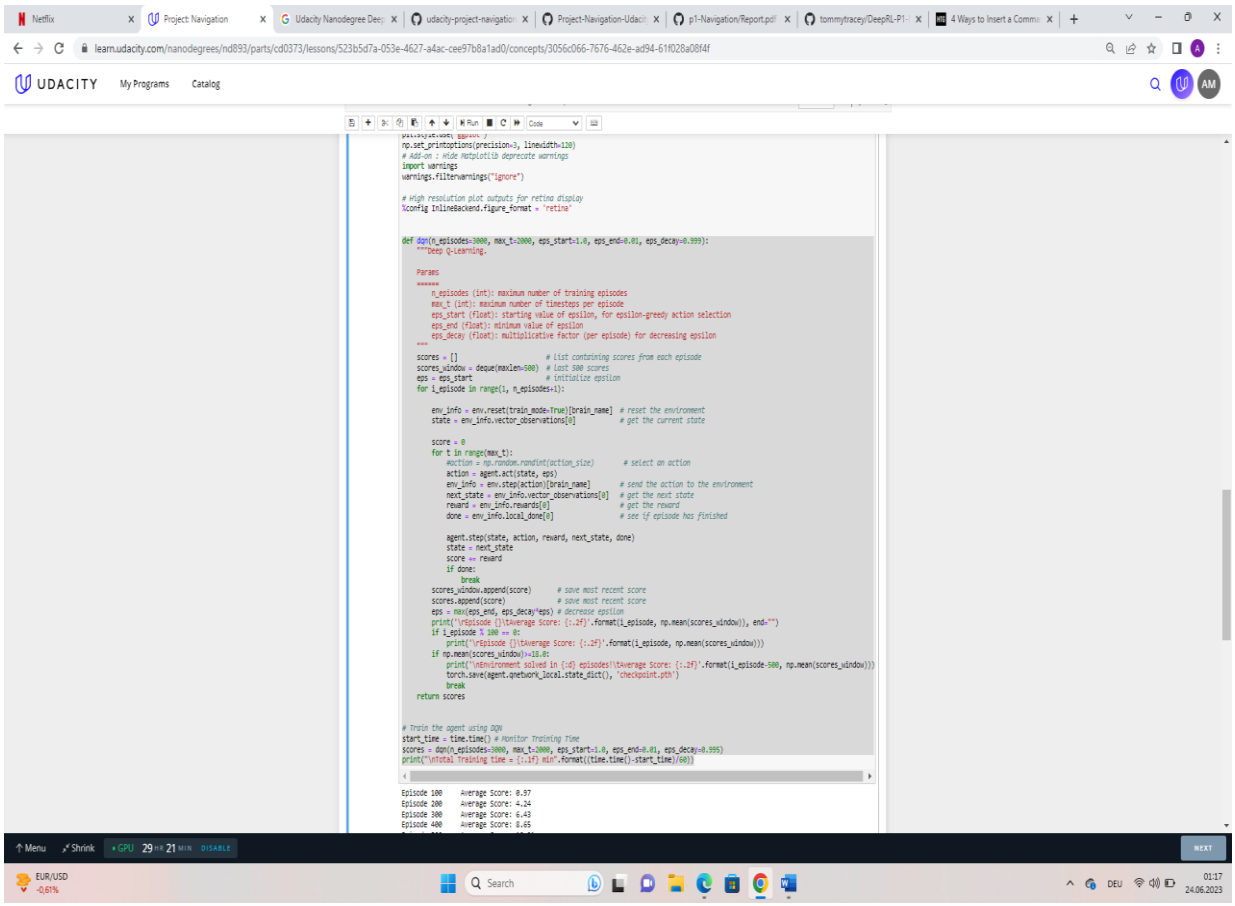
2 **Fixed Q-Targets:**

The temporal difference method is dependent on the parameter “w” that we ought to learn. This can lead to instability. That’s why we use a separate artificial neural network. The target network gets updated slowly with hyperparameter TAU and local network gets updated slowly which is called as soft update.

This deep neural network architecture also uses epsilon greedy method to get the most out of the particular action taken by an agent. In this architecture we have defined an artificial replay buffer. An experience of replay buffer initializes a deque length of specified size to store episodes length.

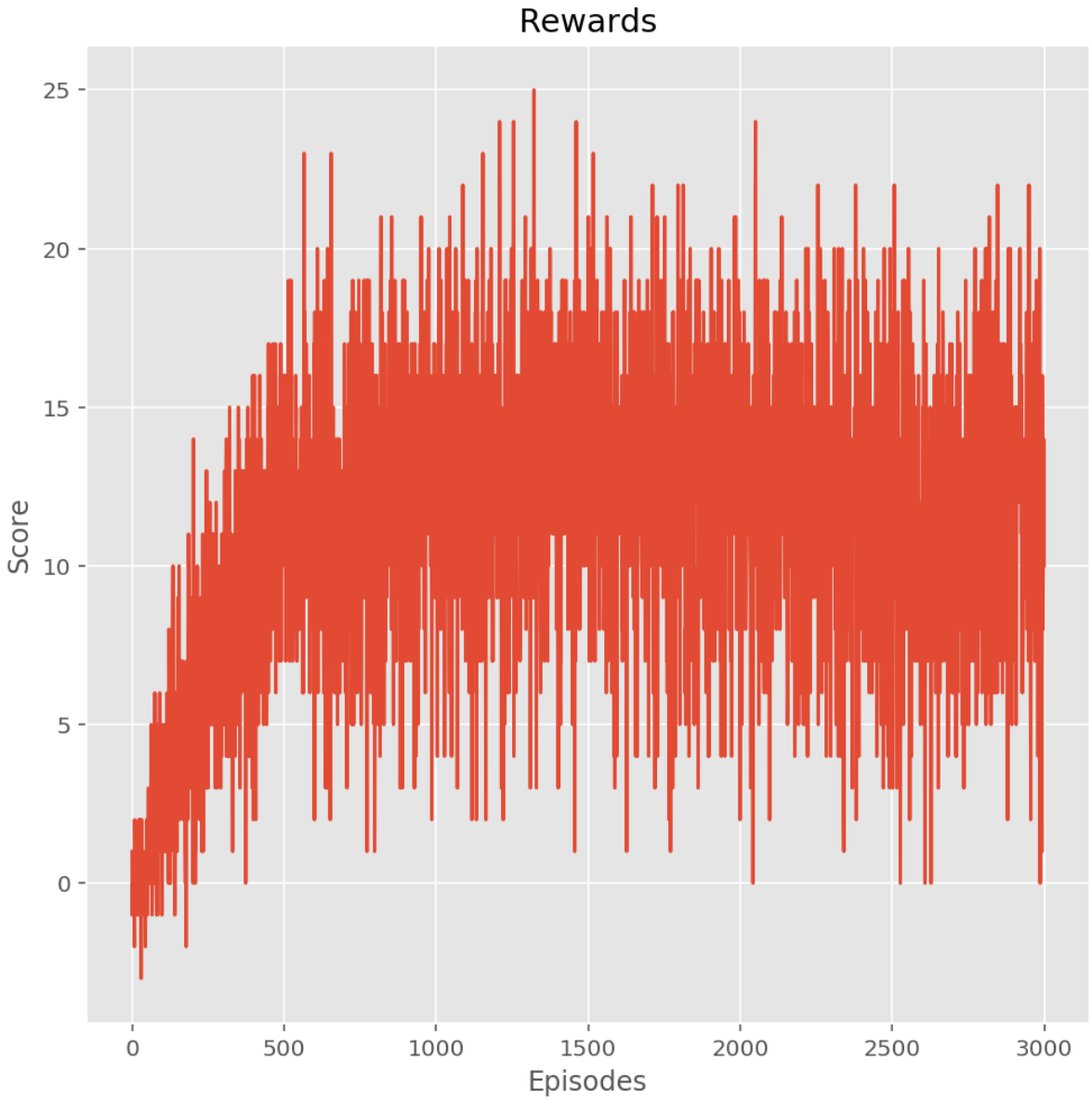
DQN NETWORK:

Here is the DQN network designed by me which comprises of 3000 episodes and records the average score and their values for each timestep at around 100 episodes:



Results:

Here is the Results of these networks:



Improvements-Ideas:

Following are the ideas for the improvements that can be undertaken in this architecture to make it more efficient and useful:

1) Try to implement Prioritized Experience Replay or Duel DQN (even Rainbow DQN).
2) Create a general DQN architecture that could be used for other architectures.