Deep Reinforcement Learning Navigation Project Report

Project Overview

The goal of this this project is to train an agent in a game environment to play the game and earn a good average score. The game environment used for the project is similar to **Banana Collector** created by **Unity Machine Learning Agents (ML-Agents)**. The agent should navigate a square world in which there are yellow and blue bananas. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. The goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space provided by environment 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 task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

Algorithms and Techniques

To train the agent for playing the game, we use Value-Based Reinforcement Learning techniques which its basic form, the agent interacts with the environment and builds a table named **Q-Table** which maintains **Action Values** for each **state action** combination. The algorithm alternates between **policy evaluation** and **policy improvement** steps to recover the optimal policy π_*.

For this project we use **Deep Q-Network (DQN)** to maintain the states and actions relationship. DQN uses a deep neural network as a function approximator. The input to the neural network is the state values and the output of the network is the action values. As the reinforcement signal, the change in game score is fed back in each time step.

The neural network structure used for the agent is the same as the one used in classroom. It has 2 hidden fully connected layers followed by Relu and one fully connected layer as output. The layers sizes are 37*64, 64*64 and 64*4 to match the states and actions space size.

The other technique which is important to consider is **Experience Replay**. When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation. By instead keeping track of a replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

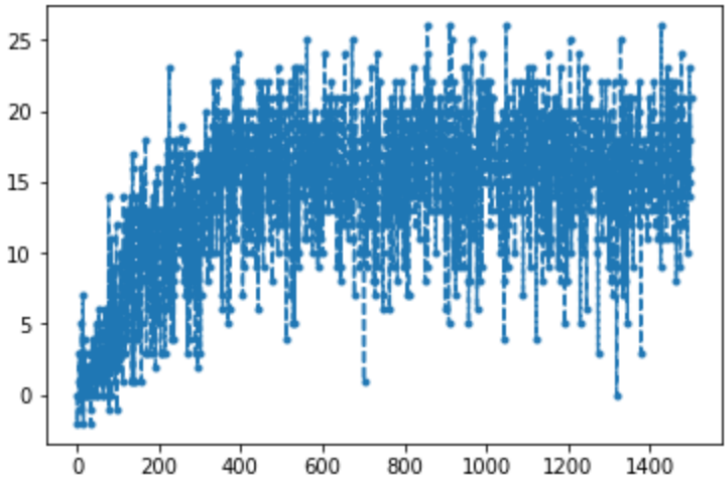
The code is implemented in Jupyter Notebook. Q-Network is coded using PyTorch library. The Agent class is similar to the agent in classroom samples which buffers state values in replay buffer and feeds them to the neural network.

To train the agent, the code loops over many episodes, 1500 to 1800, and applies the actions and gets the observations and reward from the environment. It also decays the value of epsilon in each episode for the agent starts from higher **exploration** at the beginning and moves to higher **exploitation** at the end of the training.

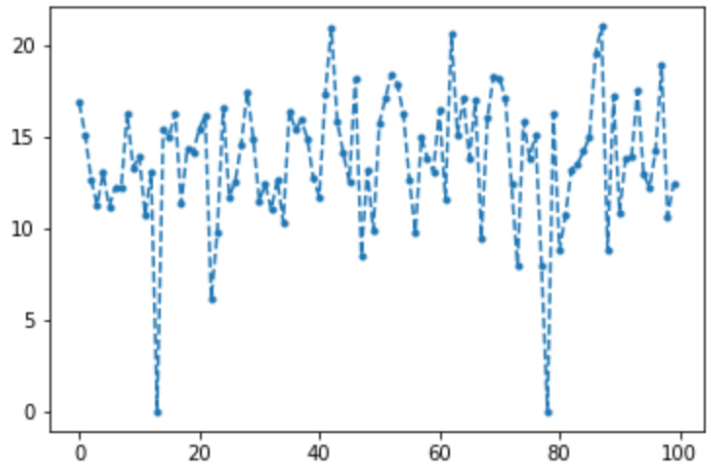
Training results

I tried different hyperparameters and the best result I could get was for changing **eps_decay** to 0.70. I could get average score higher than 13 with 1500 episode training. Here are the results for 1500 episode training:

Training scores for 1500 episodes:
Average Score: 14.24



Scores of the trained agent for 100 episodes:
Total average: 13.6122



Possible improvements

Ideally we would like an agent which can play as well or better than a human. To implement that we need to create a neural network which gets image frames of the game and learns to create optimal actions to maximize the reward, in other words it can learn from pixels. For this particular game there is an agent which produces images as the environment state. To train that we should use a **Convolutional Neural Networks** followed by some fully connected layers to learn actions based on the input image.

I tried to build a neural network based on **Deep Mind paper, Human-level control through deep reinforcement learning.** Since the state space is very large (84*84*1 for gray scale image) compared to previous state space (37) and also the neural network is deeper, the training takes a long time without GPU. Here's the simple CNN I built based on the paper:

```
class QNetwork(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        super(QNetwork, self).__init__()

        self.conv1 = nn.Conv2d(1, 32, 8 , stride = 4)
        self.conv2 = nn.Conv2d(32, 64, 4 , stride = 2)
        self.conv3 = nn.Conv2d(64, 64, 3, stride = 1)
        self.fc1 = nn.Linear(64 * 7 * 7 , 512)
        self.fc2 = nn.Linear(512,4)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

I am trying to setup an AWS instance to train the CNN. I expect with enough training for the CNN I can get human level performance.