Report for Project 1: Navigation

January 2023
Rajiv Krishnakumar

This document is a brief report on how my code for Project 1 of the Deep Reinforcement Learning Course was implemented.

1 Problem Statement

The aim of the project is to create an agent that can successfully play the UnityML Banana Game Figure 1. The game consists of a landscape with yellow and purple bananas. The agent came move forward, backwards, turn left and turn right, and will pick up a banana everytime it walks over one. The agent receives a score of +1/-1 everytime it picks up a yellow or purple banana respectively. The agent is deemed successful if it can get an average score of 13 or more in 100 consecutive games.

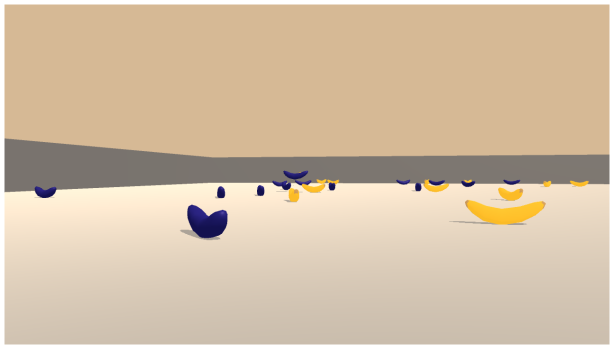

Figure 1: A screenshot of the UnityML banana environment take from this github repo
.

2 The Learning Algorithm

To play this game, we use a Deep Q-Network (DQN) agent. At the heart of this DQN lies a neural network. At every time step, the enviromnet outpus a state consisting of a vector of 37 real numbers. Given that the state vector has a relatively small (¡100) dimension, we can work with a fairly small vanilla fully-connected neural network (FCNN). We use a newtork with 4 layers: 1 input layer of 37 nodes, two hidden layers of 32 and 16 nodes respectively and finally an output node with 4 outputs. After each hidden node we use a Rectified Linear Unit (ReLU) activation function. This implementation can be seen in the `model.py` file.

Once constructed, we use this DQN as part of our agent. The agent learns through TD learning using experience replay and fixed Q targets. This is done by first initializing the agent with the FCNN mentioned above with random weights. Then we make the agent play the game, where first computes which action is likely to give the highest expected returns by inputing the state vector into the FCNN and performing the action (0 to 3) according to which output node (0 to 3) has the highest value. Here 0 is for forward, 1 for backward, 2 is for turning left and 3 is for turning right. In fact, each of the outputs stands for the Q-value of the state-action pair. E.g. if the environmnet is currently in state S, then the 0th ouput node of the FCNN gives the approximation of the value $q(S, A = 0)$. Then, with probability $(1 - \epsilon)$ it chooses that action, and with probability ϵ it chooses a random action. We start with an $\epsilon = 1$ and then decay it by 0.995 at every step (since the agent is becoming better and better) until we reach $\epsilon = 0.01$. At every timestep, we save the state, action, reward and next state, up to a total of 10^5 states (in order to avoid using too much memory). Then every 4 steps, we stop and ask the agent to learn.

During the learning stage, the agent learns through TD learning with a fixed Q target. This means that we compute the mean squared loss between our expected rewards $q(S, A|w)$ and our target rewards $r + \arg\max_A \gamma q(S', A|w^-)$ for a batch of 64 randomly sampled timesteps from our set of stored state, action, reward and next state tuples. Here r indicates the reward, γ indicates the discount factor, w indicates the weights for the local FCNN and w^- indicates the weights of the target network. Both networks have identical architectures and in the beginning, w and w^- are initialized to the same values. However, after computing the mean squared loss, the two newtorks are updated in different ways. First, weights of the local network are optimized using the Adam optimizer (with a learning rate of 5×10^{-4}). Then the weights of the target network are slightly "nudged" in the direction of the local network by changing them to a weighted average of the weights of the new local network and the current target network. In particular we use a weight of .001 of the local network. The use of this fixed Q target creates a more stable learning algorithm by decoupling the approximations with the targets. This is all implemented in the `dqn_agent.py` file.

3 Results and further directions

Our DQN agent manages to successfully paly the UnityML Banan Game. It manages to get an average of 13 or higher for 100 consecutive rounds after 600 iterations of the game. **??** shows the plot of rewards per episode for 1800 episodes. Further improvements can be made to this agent. One could try different architectures for

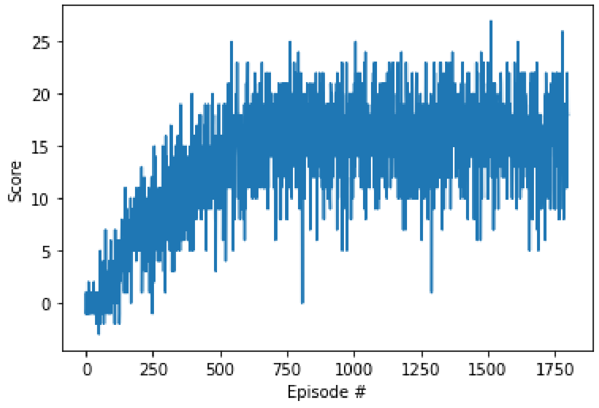

Figure 2: Plot of rewards per episode for the DQN agent for 1800 episodes

.

the FCNN. However doubling the number of nodes and layers did not seem to have such a large effect. More interesting improvements could be to implement a Double DQN agent and Prioritized Experience Replay to see how a fundamental change in the learning process can help improve the performance of the agent.