Project 1: Navigation

Dinh, Tien

May 31, 2020

1 Introduction

This project is part of Udacity Deep Reinforcement Learning Nanodegree. Our main directive was to implement a Reinforcement Learning agent to solve the Banana environment.

The environment was provided by Udacity, built on top of Unity Machine Learning Agents Toolkit.

In this environment, the agent's task is to collect as many yellow bananas as possible, while avoiding blue bananas. Collecting a yellow banana grants the agent +1 reward, and -1 reward if the agent collects a blue banana.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around 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.

2 Getting Started

After downloading the environment file and unzipping it to the correct location. You can start by creating and activating a new virtual environment (Python 3.6) to run my project. I used `virtualenv`

```
$ virtualenv drl
$ source drl/bin/activate
```

Then we can start installing the dependencies needed

```
$ cd python/
$ pip install .
```

Then to train the agent, you need to make a small modification to `main.py` file. Open it up and under line 22 where I defined the environment, the `file_name` parameter to the function call need to have the correct path to the environment file. After that is done, you can watch the agent trains in real time with

```
$ python main.py <Double>
```

Where `<Double>` can take either `true` (using Double DQN) or `false` (regular DQN)

Alternatively, you can download my pretrained weights from here, place it in the same directory with `evaluation.py` and run

```
$ python evaluation.py
```

to watch a fully trained agent plays the game.

3 Technical Details

DQN and Double DQN were implemented. In DQN, I used two identical neural networks (described below) to update weights and improve the accumulated return. In Double DQN, I also used two identical neural networks, but I use one to update the other.

Experience Replay (EP) was also implemented, the idea of EP is to interact with the environment, store the experiences in a dictionary, and sample randomly from this dictionary to learn.

For neural network architecture, I used a 3 layers fully connected perceptron network to map from the state to action values. Each hidden layer consists of 128 fully connected units. The activation function used was ReLU activation function.

Hyperparameters

Full list of hyperparameters used:

- Replay Buffer size: 10^5
- Training sample (batch) size: 64
- Discount rate: 0.99
- Soft update constant: 10^{-3}
- Learning rate: 5×10^{-4}
- Network update interval: 4 episodes
- Total episodes trained: until solved or maximum of 2000 episodes, whichever comes first.

- Maximum timesteps per episode: until the environment sends a terminating signal or 1000 timesteps, whichever comes first.
- Epsilon (for ϵ-greedy policy): Starts with $\epsilon = 1.0$, ends with $\epsilon = 0.01$, decays with constant 0.995.

4 Results

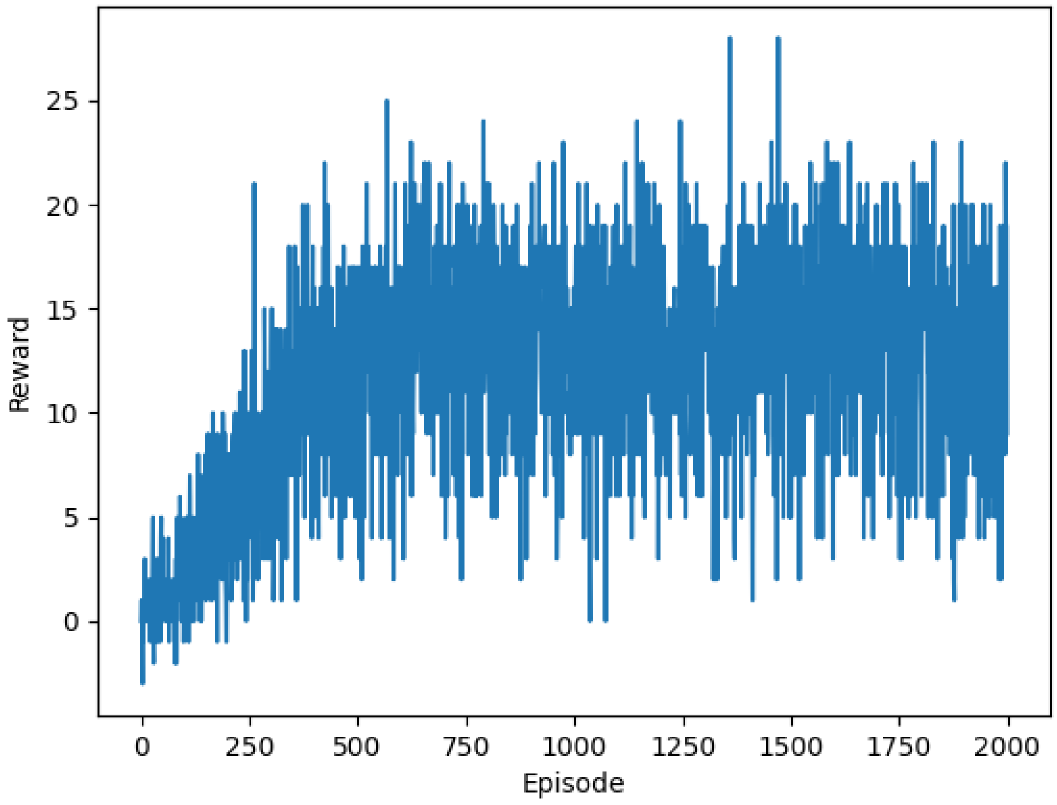

Figure 1: Accumulated return over 2000 episodes

My agent was able to solve the environment in 503 episodes for regular DQN, and 381 episodes for Double DQN network.

5 Future Work

There are many aspects in which I can improve this project. Implementing dueling DQN or prioritized experience replay would likely improve the number of episodes until solved.

Now that this environment is solved, I should try out solving the same game, but with a different state representation. Representing the state as raw pixels would be more realistic since raw pixels are what we see, and we should allow the agent to see the same.