Banana Navigation Project Report

Patrick Sowinski

This project implements a Deep Q-Learning Network (DQN) to solve the Banana Navigation environment from Unity ML Agents.
The project is part of Udacity Nanodegree program on Deep Reinforcement Learning.

The Environment

The environment is implemented within Unity ML Agents. More info about these environments can be found here: https://github.com/Unity-Technologies/ml-agents

In this case, a single agent has to navigate a square world (bounded by visible walls). The goal is collect as many yellow bananas as possible, while avoiding blue bananas. The bananas spawn at random locations. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana.

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

The Solution

The environment was solved with a DQN (Deep Q-Network) agent.
The agent has an “act” function, which determines the next action to take given the environment’s state. It also has a “step” function, which adds experiences to the replay buffer. This replay buffer is used for learning, which includes updating the network weights.
The agent uses a local and a target Q-Network to compute losses. The target network weights are updated after every learning step with a soft update from the local network.

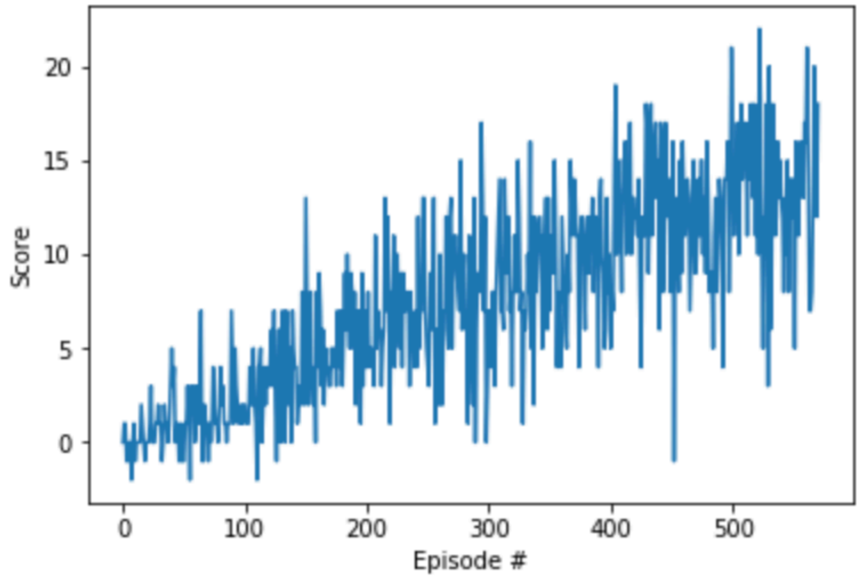
The local and target network each consist of 5 fully connected layers with ReLU activations between them (input and output layers + 3 hidden layers). The final layer output is directly taken as the network’s output with no activation function.
The input size corresponds to the size of the state space, given by the environment, which in this case has 37 dimensions. The output size of the networks corresponds to the number of available actions, which is 4 here. When the agent selects an action, it takes the one corresponding to the highest network output. The output could also be used for a stochastic policy, but we only use the highest output here, making this a deterministic policy.
The hidden sizes of the layers are 300, 200 and 100.

The replay buffer has a size of 100,000, so it can store a variety of different experiences.
The batch size for a learning step is 64, which reduces the noise of the training curve well enough for progress to be visible.
The factor gamma used for discounting rewrds is set to 0.99, which is probably irrelevant in this case. 1.0 should work as well.
The soft update factor tau is set to 0.001, which means it takes roughly 1000 steps to completely overwrite the target network weights.
The learning rate for the local network is set to 0.0005, which just works.
After every 4 steps in the environment, a learning step happens with a random batch of experiences from the replay buffer.

Results

Below is the plot of rewards per episode during training. The environment was solved as soon as the agent reached an average reward of +13 over the previous 100 episodes.
The required average was reached between episodes 471 and 571.

```
Episode 100     Average Score: 1.11
Episode 200     Average Score: 4.25
Episode 300     Average Score: 7.03
Episode 400     Average Score: 9.26
Episode 500     Average Score: 11.98
Episode 571     Average Score: 13.06
Environment solved in 471 episodes!     Average Score: 13.06
```



A video of the trained agent performing can be seen here: https://youtu.be/QgCX-zGajT8

Ideas for future work

When evaluating the trained agent, the agent sometimes gets stuck if we use a greedy policy. This mostly happens when a blue and a yellow banana are visible at equal viewing angles and distances. This can be solved by using an epsilon-greedy policy with a small value epsilon for random actions. However, a better approach might be to build the agent so it uses a stochastic policy instead of a deterministic one. This would also allow it to get out of a situation where it is stuck.

Another improvement would be to use prioritized experience replay, so we learn more often from uncommon experiences that provide a lot of information.

On another note, the agent would be able to achieve even higher scores if it could select to move at different speeds. It would need to slow down when it is searching for a new target or turning around and could go faster when it is focused on reaching a target. This can only be achieved by modifying the given environment and redefining the action space. A realistic physics simulation for cornering speeds would make this task even more applicable to real-life solutions.