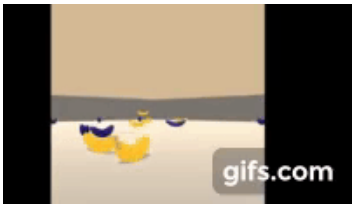
Report for Project 1 - Deep Reinforcement Learning Nanodegree



This report contains the details of the code used to solve the first project in Udacity's Deep Reinforcement Learning Nanodegree program.

The algorithm

The agent uses a policy to move in the environment and find the optimal policy to maximize its reward. The optimal policy is the result of a trial and error effort through which the agent leans by iteration. This kind of algorithm is called Q-Learning and the optimal policy is found by running the Q-function which calculates the expected reward for all possible actions and all possible states.

The agent uses a deep neural network to approximate the Q-function. The Deep Q-Network contains three fully connected layers of 64, 64, 4 nodes. The learning rate is set to: 0.0005. This learning rate has given the best results during training.

The agent uses also an Replay buffer, the experience is stored in a buffer and the agent samples randomly form this buffer during the learning steps.

Hyperparameters tuning

Agent hyperparameters:

- BUFFER_SIZE = int(1e5) # replay buffer size
- BATCH_SIZE = 64 # minibatch size

- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR = 5e-4 # learning rate
- UPDATE_EVERY = 4 # how often to update the network

I have tested different hyperparameters combinations but this one gave the best results, it is the basic combination provided in a previous lesson. I did many experiments changing values the eps_decay, eps_end and decreasing the eps_decay during training gave good results. The decreasing epsilon value was clearly giving the greedy agent a way to learn more by opening up new possibilities with and increasing probability.

The experiments and hyperparameters

I have tried different learning rates and found out an interesting thing: the agent learns from previous trainings as well!

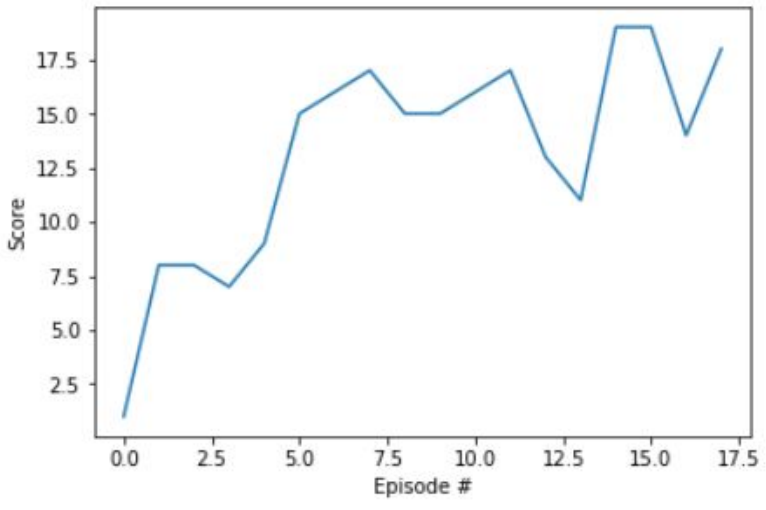
```
scores, checkpoint = dqn(n_episodes=300, eps_decay=0.98, eps_end=0.02)
```

```
Episode 100     Average Score: 7.45
Episode 177     Average Score: 13.00
Environment solved in 77 episodes!      Average Score: 13.00
```

By starting the agent with the trained weights and decreasing the decay, the training results even more effective and the training speeds up, **resolving the environment in 18 episodes**! Please see my training results below.

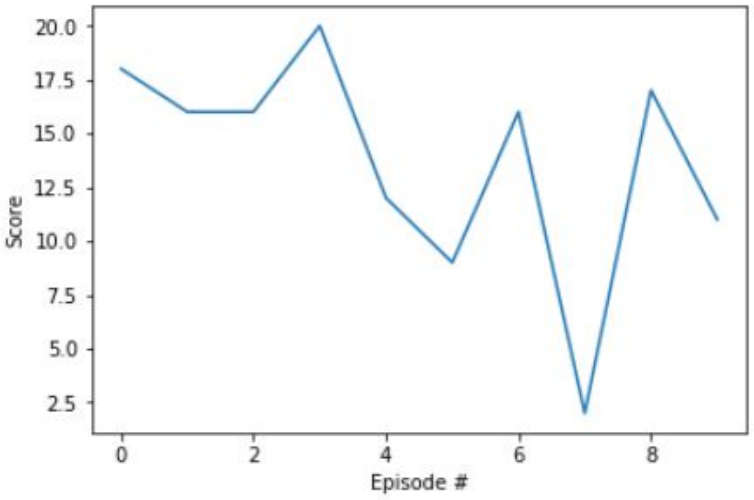
```
bestscores2, checkpoint2 = dqn(n_episodes=300, eps_decay=0.55, eps_end=0.01)
```

```
Episode 18      Average Score: 13.22
Environment solved in -82 episodes!     Average Score: 13.22
```



The trained agent achieves a high score also in the testing part as shown in the tested code and in

```
Episode 1       Average Score: 18.00
Episode 2       Average Score: 17.00
Episode 3       Average Score: 16.67
Episode 4       Average Score: 17.50
Episode 5       Average Score: 16.40
Episode 6       Average Score: 15.17
Episode 7       Average Score: 15.29
Episode 8       Average Score: 13.62
Episode 9       Average Score: 14.00
Episode 10      Average Score: 13.70
```



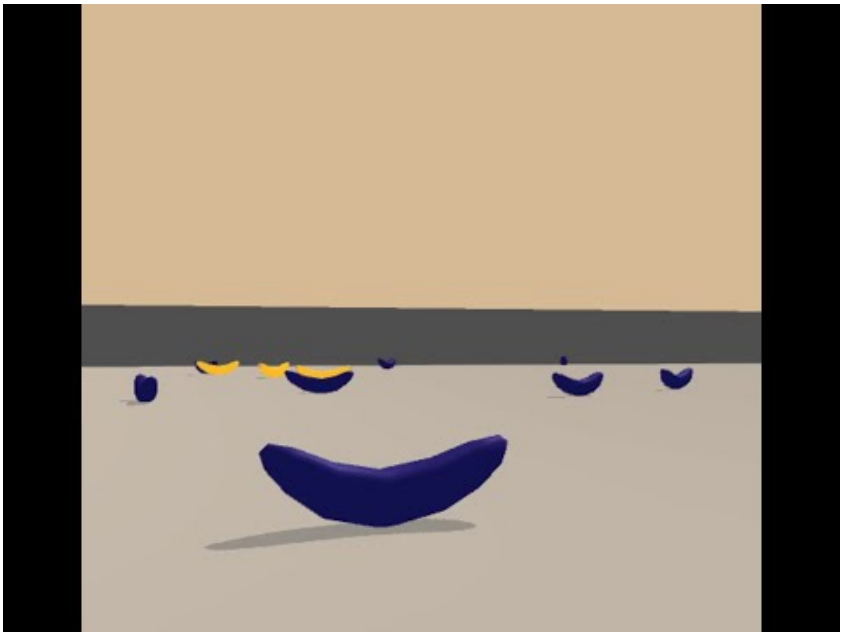
the video.

As I went on with my experiments I realized the agent could show signs of overfitting for this simple task. But training it longer, the agent showed it was getting better also with higher scores 20 and 30, just by using the same technique. Therefore I did not need to make any additions to the algorithm as it was solving the environment really easily.

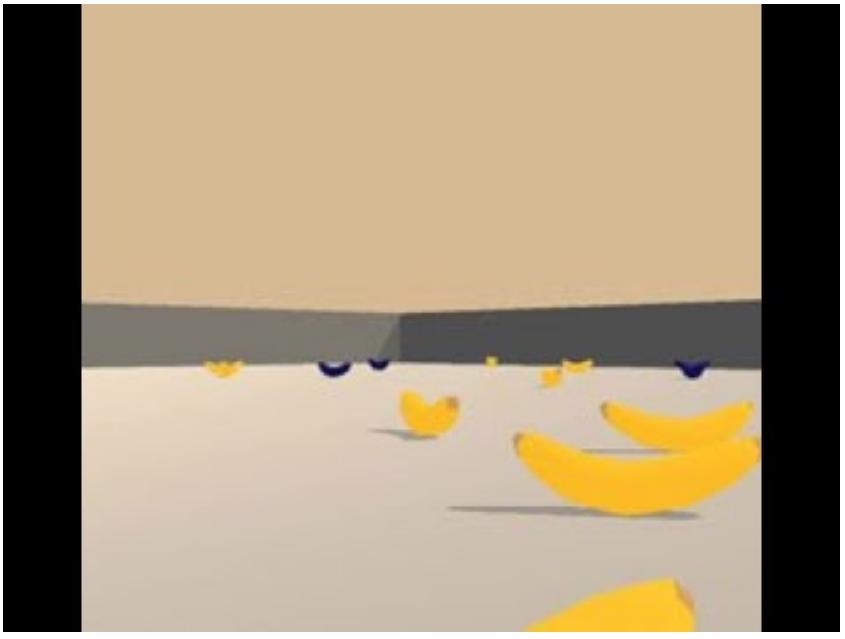
Video of the agent: watch the results

I recorded two videos of my agent, the untrained and trained agent.

Video of the untrained agent



Video of the trained agent



Algorithm improvements

The algorithm solves a simple game but if it were used for a more difficult task several improvements could be added:

- a Double DQN
- a Dueling DQN
- Prioritized experience Replay

Prioritized experience Replay

The prioritized experience replay adds significance values to each stored tuples to give a priority to reuse the tuples according to a level of priority assigned on the base of previous experience.

Double DQN

Q-learning is prone to overestimation of the action values because is based on the maximization steps which selects higher values without trying other possible values. Are these values trustful? The mechanism of double DQN provides a robust counterpoint, with an evaluation step which relies on another set of values. If the two mechanisms are not agreeing on the maximum, the value set is lower. These prevents propagating of false maximum values obtained by chance.

Dueling DQN

The dueling DQN uses two streams, one estimates the state value function, the other the advantage for each action. The two branches are in the fully-connected layers. The Q-value is a combination of the two values.

Project Starter Code

The project starter code of the original Udacity repo for this project can be found here.