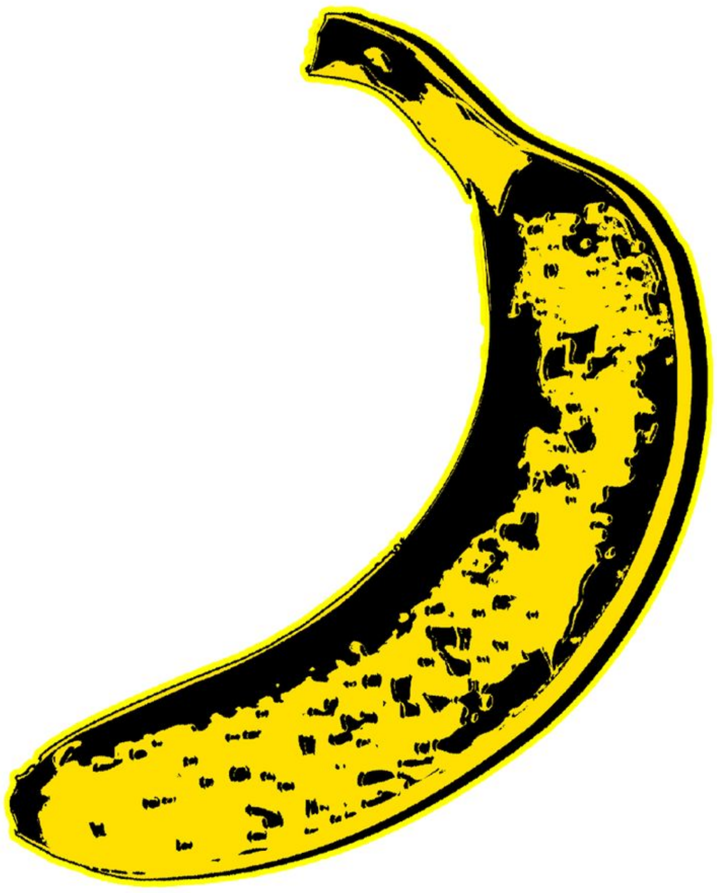
NAVIGATION PROJECT REPORT

Train an agent to navigate (and collect bananas) in a large, square world



Dimitris Poulopoulos

08.25.2018

Udacity Deep Reinforcement Learning Nanodegree

* Picture by Telrúnya (https://commons.wikimedia.org/wiki/File:VU-Banana-1000x1000.png)

INTRODUCTION

Unity Machine Learning Agents (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents.

For game developers, these trained agents can be used for multiple purposes, including controlling NPC behaviour (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release.

In this project, we develop a Deep Q-Learning agent that utilises its newly acquired skills to navigate in a large, square example world and collect bananas. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

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:

- move forward
- move backwards
- turn left
- turn right

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

LEARNING ALGORITHM

This project leverages a Deep Q-Network (DQN) to realise the Deep Q-Learning algorithm, pioneered by Deep Mind[1]. This network takes as input a state space of 37 dimensions,

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

that describe the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Then, it passes this information through a relatively simple network architecture, consisting of two fully connected, hidden layers, with *ReLU* activation functions. The output and third layer of the network is a pure linear layer, with the identity activation function. It consists of four nodes, one for each available action. The value of every node is the q value of the corresponding action, given the input state. Figure 1 depicts the architecture of the network.

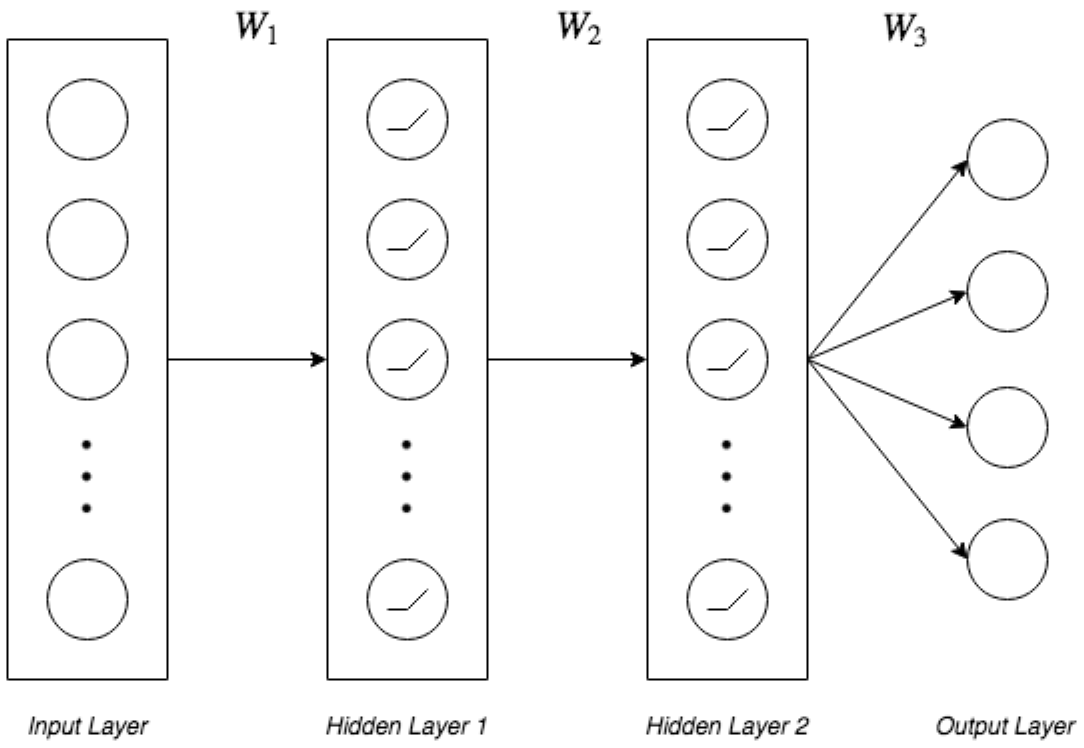


Figure 1 - Network Architecture

PROCEDURE

The output layer of the neural network consists of four nodes, each for every possible action. The node with the highest value indicates the action to take (in a greedy setting). As a reinforcement signal, it receives a reward of +1 for a yellow banana and a reward of -1 for collecting a blue banana.

In the beginning, when the neural network is initialized with random values, the actions it takes are also random. But given time, it learns the associations between states and sequences with the appropriate actions.

EXPERIENCE REPLAY

To avoid a naive Q-Learning algorithm getting swayed by the correlation between a sequence of tuples, we use a technique called *Experience Replay*. Thus, instead of training the agent while it observes the environment and takes action, we are keeping track of a *replay buffer* and later, during training, we sample from it at random, breaking those correlations and preventing action values from oscillating or diverging catastrophically.

The *replay buffer* is a collection of *experience* tuples - S , A , R , S' . The act of sampling a small batch of tuples from the *replay buffer* in order to learn is known as *Experience Replay*. Moreover, this technique allows us to reuse individual tuples to learn more or recall rare occurrences.

FIXED Q-TARGETS

In Q-Learning, the targets that we are trying to optimize against are also a guess. Thus, we update a guess with a guess and this, apart from being mathematically incorrect[2], this can lead to harmful correlations. To avoid this we are updating the parameters w of the network q with a formula as shown below:

$$\Delta w = \alpha \,(R + \gamma \, max_a \, q\,(S', \alpha, w^-) - q\,(S, A, w))\, \nabla_w \, q(S, A, w)$$

where w^- is a copy of w, that gets updated periodically and not on every iteration. For example, w^- gets updated every four iterations with the newly calculated weights.

RESULTS

The results of the experiment are summarised in table 1 and depicted in figure 2.

EPISODE	AVERAGE SCORE
100	0.95
200	4.38

[2] The targets also depend on the weights of the network, and during optimization, we are calculating the gradients with respect to those weights

300	7.17
400	10.11
500	12.66

Table 1 - Results

To consider the environment solved, the agent must get an average score of +13 over 100 consecutive episodes. This, for the specific network architecture, happens in 414 episodes according to the results. To achieve this, we built the network with the following hyperparameters:

- Learning rate: 0.001
- Batch size: 64
- Discount rate (γ): 0.99
- Soft update (τ): 0.001
- Initial epsilon/Final epsilon (ε): 1/0.1
- Epsilon decay: 0.995
- Hidden layers: 2
    - First/Second hidden layer units: 64
- Buffer size: $1 \cdot 10^5$

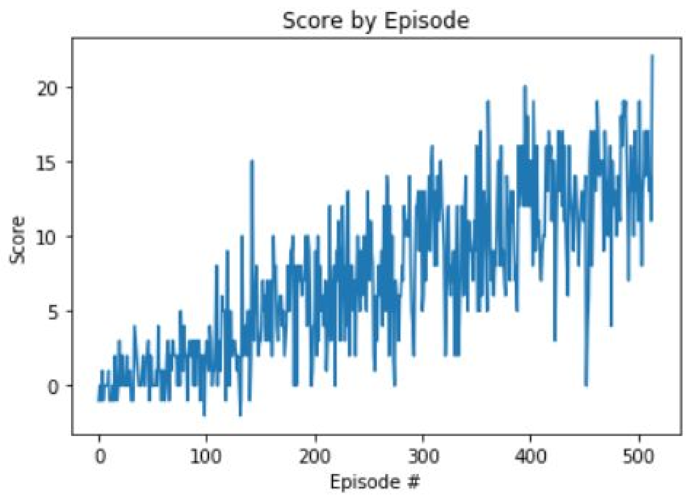


Figure 2 - Results

FUTURE IMPROVEMENTS

For future work, we could Implement a double DQN[3], a duelling DQN[4], and/or prioritized experience replay[5], to made the agent learn faster and go beyond its current abilities. We could also try and learn from raw pixels, but this would also mean that we should change the network architecture and implement several convolutional layers, to take advantage of the spatial information.

[3] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. Vol. 2. 2016.

[4] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581(2015).

[5] Schaul, Tom, et al. "Prioritized experience replay." arXiv preprint arXiv:1511.05952 (2015).