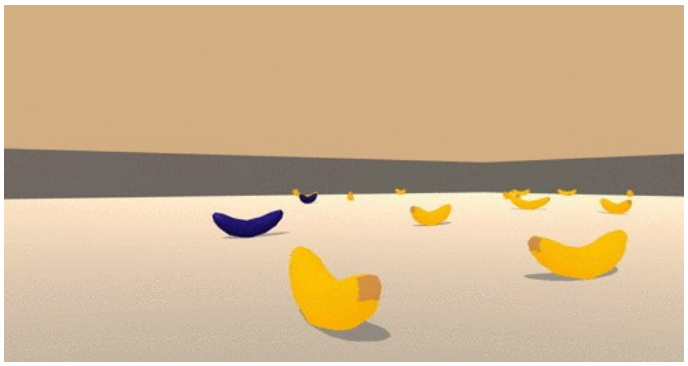
21.03.2020

**Udacity Nanodegree Deep Reinforcement Learning**
**Sören Klingner Report**

# Navigation

## 1. Introduction

The goal of the project is to train an agent to navigate a virtual world and collect as many yellow bananas as possible while avoiding blue bananas.



To achieve this goal, I implemented a Value Based method called Deep Q-Networks[1], which had been presented in the previous Udacity lesson. Modifications had been made to the implementation to work with the provided unity environment. To meet the projects requirements the hyperparameters and the average of rewards needed had been modified.

Deep Q-Learning combines 2 approaches:

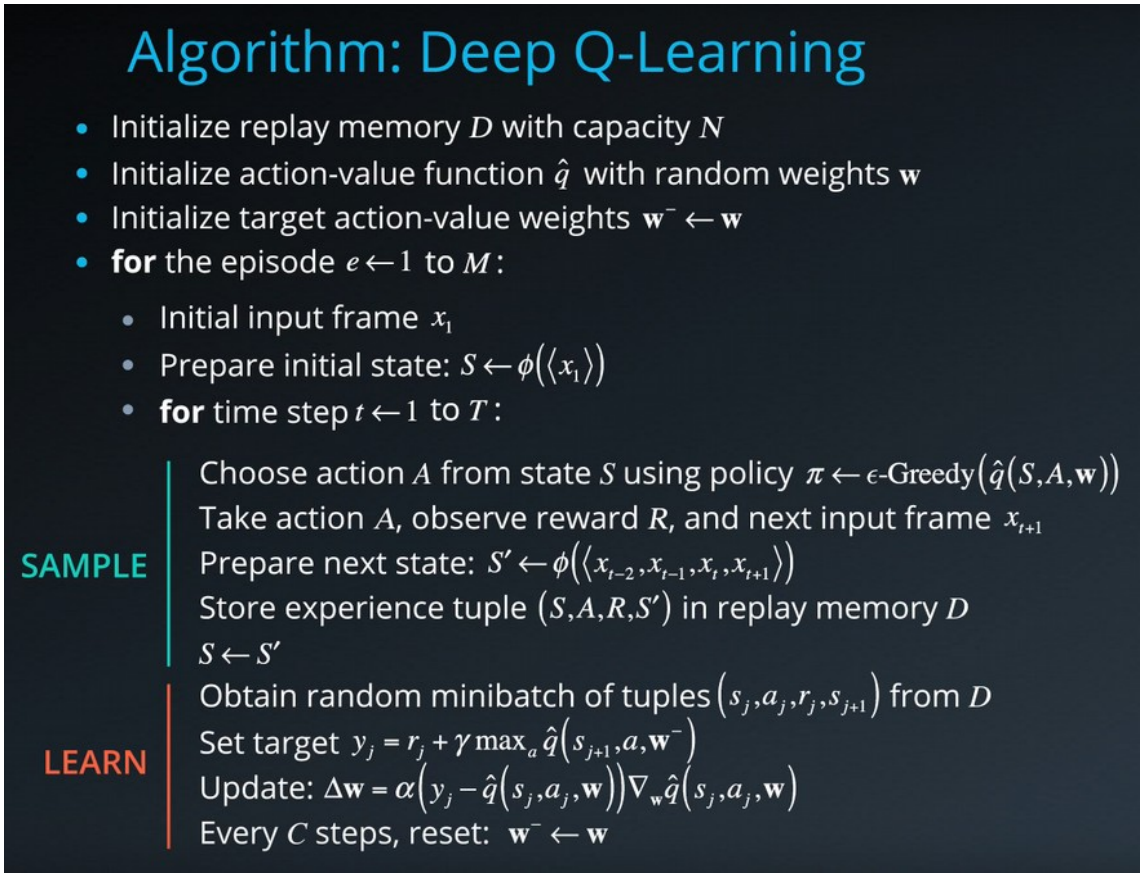
- A Reinforcement Learning method called Q Learning[2] (SARSA max)
- A Deep Neural Network to learn a Q-table approximation (action-values)

This implementation includes 2 major training improvements by Deepmind[3] and described in their publication: "Human-level control through deep reinforcement learning (2015)"[4].

- Experience Replay
- Fixed Q Targets

1 https://deepmind.com/research/dqn/
2 https://en.wikipedia.org/wiki/Q-learning
3 https://deepmind.com
4 https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

# 2. Algorithm




The screenshot is taken from the Udacity Nanodegree “Deep Reinforcement Learning”[5].

5 https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

# 3. Implementation

The code is derived from the "Lunar Lander" lecture from the Deep Reinforcement Learning Nanodegree, and has been slightly adjusted for being used with the banana environment. First of all, all necessary packages are been imported and state and action spaces are been examined. Next, the model and agent gets implemented, following by training the DQN agent and plotting the scores.

### The Model

As the model, a Q-Network had been implemented. This is a regular fully connected Deep Neural Network using PyTorch. This network will be trained to predict the action based on the states.

This Neural Network is used by the DQN agent and is composed of:

- The input layer which size depends of the state_size parameter
- 2 hidden fully connected layers of 1024 cells each
- The output layer which size depends on the action_size parameter

### The Agent

The agent is constructed out of a DQN agent and a Replay Buffer memory. The DQN agent class is implemented, as described in the Deep Q-Learning algorithm. It provides several methods:

### DQN Agent

- **constructor**
  - Initializes the memory buffer (Replay Buffer).
  - Initializes 2 instance of the Neural Network: the target network and the local network.

- **step()**
  - Allows to store a step taken by the agent (state, action, reward, next_state, done) in the Replay Buffer/Memory.
  - Every 4 steps and if their are enough samples available in the Replay Buffer, update the target network weights with the current weight values from the local network.

- **act()**
  - Returns actions for the given state and current policy.
  - The action selection use an Epsilon-greedy selection to balance between exploration and exploitation for the Q Learning.

- **learn()**
  - Updates the Neural Network value parameters using given batch of experiences from the Replay Buffer.

- **soft_update()**
  - It is called by learn() to softly update the value from the target Neural Network from the local network weights.

**ReplayBuffer**

It implements a fixed-size buffer to store experience tuples (state, action, reward, next_state, done)

- **add()**
  - Adds an experience tuple to the buffer memory.

- **sample()**
  - Randomly samples a batch of experience tuples for learning.

# 4. Hyperparameters

The following hyperparameters were used during training:

| | |
|---|---|
| n_episodes=2000 | Total number of episodes to train for |
| max_t=500 | Max number of time steps per episode |
| eps_start=1.0 | Epsilon starting value |
| eps_end=0.01 | Epsilon decay minimum value |
| eps_decay=0.995 | Epsilon decay rate |
| | |
| 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 (aka alpha) |
| UPDATE_EVERY = 4 | How often to update the network |
| | |
| state_size = 37 | Dimension of each state |
| action_size = 4 | Dimension of each action |
| seed = 42 | Random seed |
| fc1_units = 64 | Number of nodes in first hidden layer |
| fc2_units = 64 | Number of nodes in second hidden layer |

These parameters were adjusted via trial an error based on the agents ability to learn the task well enough to meet the rubric spec, an average reward of 13.0 over 100 episodes.

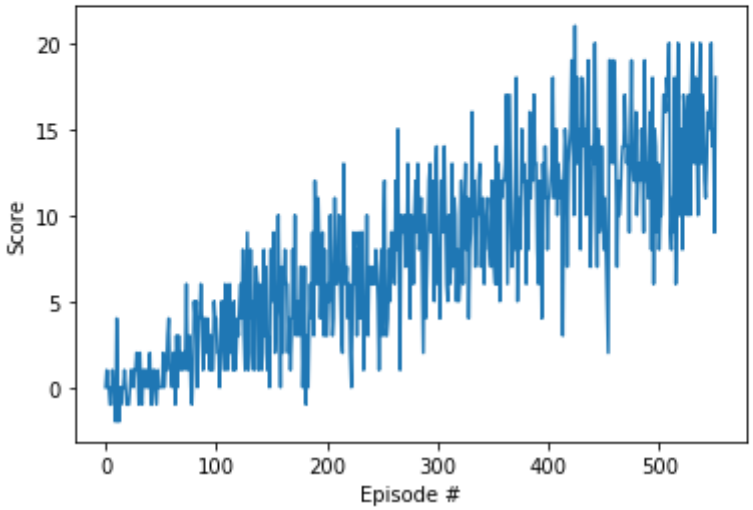
The Neural Networks use the following architecture:

The input layer consists of 37 units. Then we have two fully connected hidden layers each with 64 units and Relu activation. The output layer consists of 4 units.

# 5. Results

Given the chosen architecture and parameters, our results are:

```
Episode 100     Average Score: 1.18
Episode 200     Average Score: 4.47
Episode 300     Average Score: 7.20
Episode 400     Average Score: 9.973
Episode 500     Average Score: 12.92
Episode 552     Average Score: 13.51
Environment solved in 452 episodes!     Average Score: 13.51
```




These results meets the project's requirements as the agent is able to receive an average reward (over 100 episodes) of at least +13, from episode 452 to 552.

# 6. Ideas for Future Work

As discussed in the Udacity lecture, a further improvement would be to train the agent directly from the environment's observed raw pixels instead of using the environment's internal states. For this, a Convolutional Neural Network could be added at the input of the network to process the pixel values.

Other improvements could be:

- Double DQN
- Dueling DQN
- Prioritized experience replay

Lastly, hyperparameters could be further tuned to find better combinations. Automated grid search to find the hyperparameters could be applied.