Report

September 22, 2019

1. Report

The aim of this report is to describe the learning algorithm method used in the Navigation's Project, provided by Udacity, along with its chosen hyperparameters.

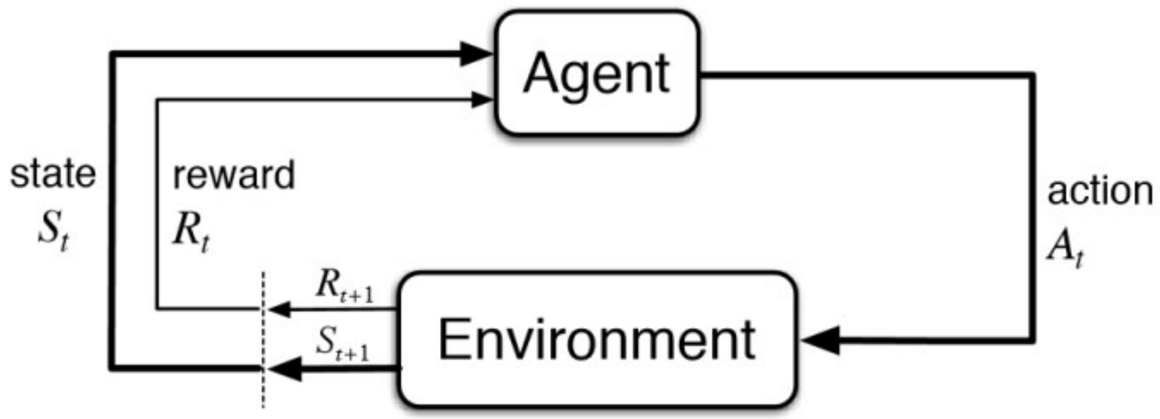
2. Background

2.1 – Markov Decision Process (MDP)

The Markov Property means that each *state* is dependent solely on its preceding state, the selected *action* taken from that state and the *reward* received immediately after that action was executed.
Mathematically, it means: $s' = s'(s,a,r)$, where s' is the future state, s is its preceding state and a and r are the action and reward. No prior knowledge of what happened before s is needed — the Markov Property assumes that s holds all the relevant information within it. A Markov Decision Process is decision process based on these assumptions.

2.1 – Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a machine learning approach to artificial intelligence concerned with creating computer programs that can solve problems requiring intelligence. The peculiar property of DRL algorithms is the learning through trial and error from feedback, that is simultaneously sequential, evaluative and sampled by leveraging powerful non-linear function approximations. DRL algorithms learns what to do, how to map situations to actions, so as to maximize a numerical reward signal.



Reinforcement Learning Illustration (https://i.stack.imgur.com/eoeSq.png)

2.1.1 – Policy (**π**)

The policy, denoted as **π**, is a mapping from some state *s* to the probabilities of selecting each possible action given that state. An <u>optimal</u> policy π_* is the solution to the Markov Decision Process and it is the best strategy to accomplish its goal.

2.1.2 – Greedy Policy, ***ε***-Greedy Policy

A greedy policy means the Agent constantly performs the action that is believed to yield the highest expected reward. Such policy will not allow the Agent to explore the environment at all. In order to still allow some exploration, an ***ε***-Greedy Policy is often used instead: a variable (named ***ε***) in the range of [0,1] is selected and prior selecting an action, a random number in the range of [0,1] is selected. If that number is larger than ***ε***, the greedy action is selected, however if it is lower, a random action is selected. Note that if ***ε***=0, the policy becomes the greedy policy and if ***ε***=1, always explore.

2.1.3 – State action value function (Q function)

A state-action function is also called the Q function. It specifies how good it is for an agent to perform a particular action in a state with a policy π.

We can define Q function as follows:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right]$$

This specifies the expected return starting in state *s* with the action *a* according to policy π.

2.1.4 – Q Learning Algorithm

Q-Learning is an *off-policy Reinforcement Learning* algorithm. In its most simplified form, it uses a table, also known as Q-table, to store all Q-values of all possible *state-action* pairs. It updates its table using the *Bellman equation,* while action selection is usually made with an **ε**-greedy policy. The optimal Q-value, denoted as Q^* can be expressed as:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \lambda \max_{a'} Q^*(s',a')|s,a]$$

Optimal Q-value (https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit)

	Action	
State	Left	Right
0	0	0
1	-100	65.61
2	59.049	72.9
3	65.61	81
4	72.9	90
5	81	100
6	0	0
7	100	81
8	90	72.9
9	81	0

Q-table Example
(https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8)

2.1.5 – Bellman Equation

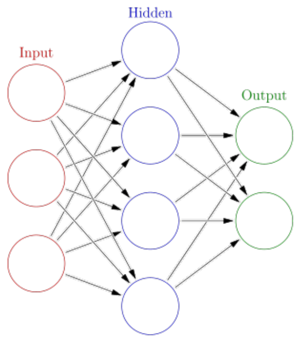
Defines the relationships between a given state (or state-action pair) to its successors. While many forms exist, the most common one usually encountered in *Reinforcement Learning* tasks is the Bellman equation for the optimal Q-value, which is given by:

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

Bellman Equation

2.2 – Artificial Neural Networks

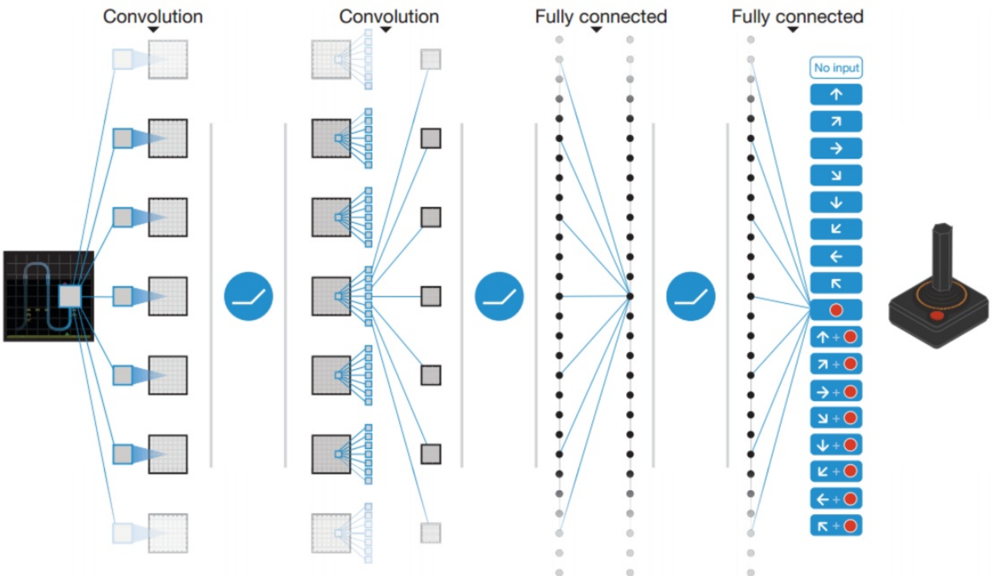
Artificial neural networks are computing systems that are inspired by biological neural networks that constitute animal brains. Such systems “learn” to perform tasks by considering examples, generally without being programmed with task-specific rules.



ANN architecture

2.3 – Deep Q Network Algorithm (DQN)

The DQN algorithm is a variant of the Q-Learning that replaces the state-action table with a neural network in order to cope with large-scale tasks, where the number of possible state-action pairs can be enormous. The Deep Q-Learning algorithm represents the optimal action-value function q_* as a neural network.



DQN Atari Example (https://zhuanlan.zhihu.com/p/25239682)

The forward pass goes through several layers including convolutional layers as well as fully connected layers. The output is the Q-value for each of the actions that the agent can take.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Other techniques are also essential for training DQN:

1. Experience Replay:

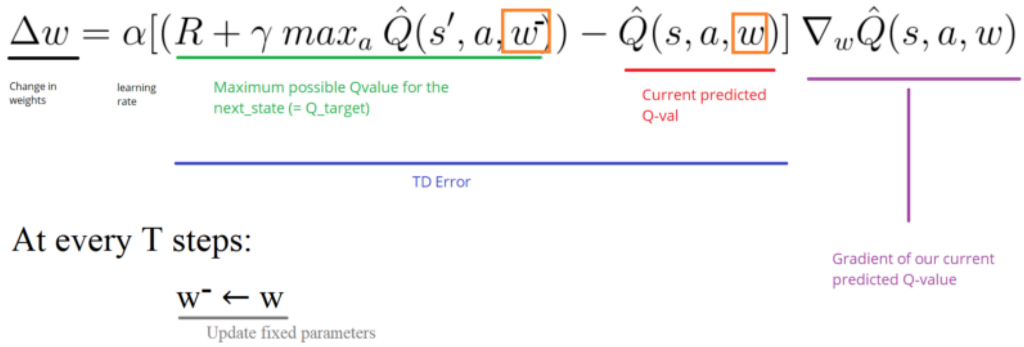
The idea of experience replay is storing each experienced tuple into a **replay buffer** as we are interacting with the environment and then sample a small batch of tuples from it in order to learn. As a result, we're able to learn from individual tuples multiple times, recall rare occurrences and in general make better use of experience.

2. Separate Target Network:

The target Q Network has the same architecture as the one that estimates value. Every C steps, according to the above pseudo code, the target network is reset to another one. Therefore, the fluctuation becomes less severe, resulting in more stable trainings.

3. Fixed Targets:

In DQN, we update a guess based on a guess and this can potentially lead to harmful correlations. To avoid this, we can update the parameters θ in the network to better approximate the action value corresponding to state s and action a with the following update rule:



Fixed Targets update rule
(https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/)

PS: w = $\boldsymbol{\theta}$ and w- = $\boldsymbol{\theta}^-$

2.4 – Environment

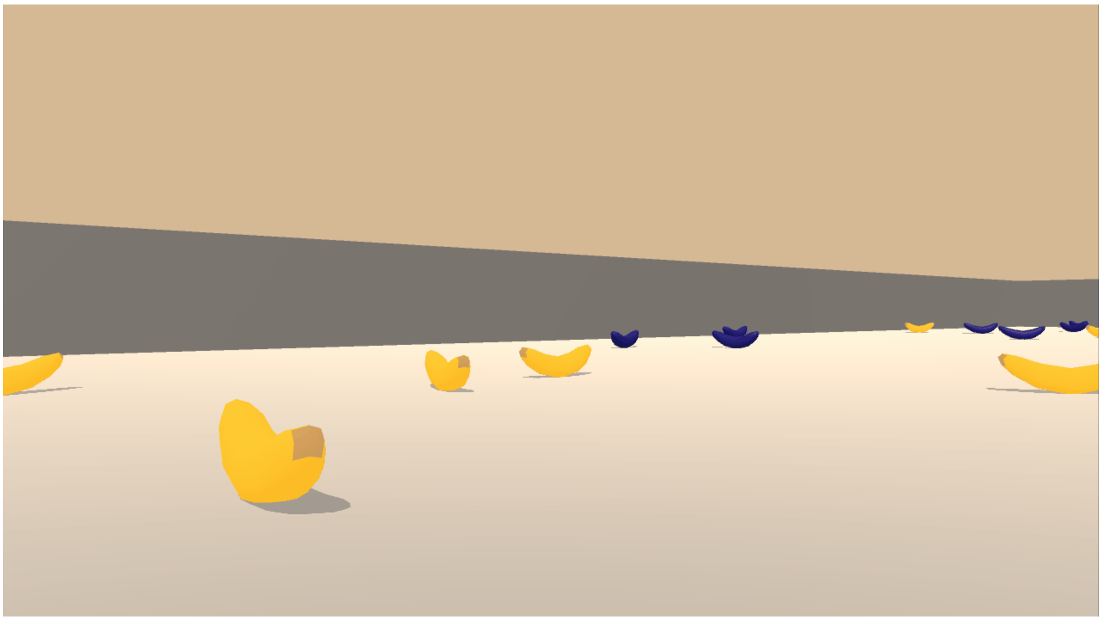
The goal of this project is to train an agent to navigate and collect bananas in a squared world! It has to collect as many yellow bananas as possible while avoiding purple bananas.

Details

This environment was developed on Unity Real-Time Development Platform and it has some peculiarities:

- A reward of +1 is provided for collecting a yellow banana
- A reward of -1 is provided for collecting a purple 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.



Udacity's Environment
(https://github.com/lucastakara/NavigationProject-DRLND/blob/master/README.md)

3.1 – Solution

The solution was implemented using Python 3 language programming. 3 scripts (model.py, dqn_agent.py, navigation.py) were created in order to maintain the code organized. The first script (model.py) contains the neural network architecture used for the solution. The Second (dqn_agent.py) contains the **replay buffer** as well as agent's class. Finally, the last script contains the function that will train the agent and plot scores in order to evaluate its performance.

3.2 – Neural Network Architecture

3.2.1 – Dueling Network

The dueling architecture, explicitly separates the representation of state values and (state-dependent) action advantages. The dueling architecture consists of 2 streams that represent the state value V(s) and advantage functions A(s, a). While sharing a common convolutional feature learning module. Automatically produces separate estimates of the state value function and advantage function, without any extra supervision. The implementation of this architecture can be seen in "model.py" file.

$$Q(s,a) = A(s,a) + V(s)$$

$$Q(s,a,w) = V(s;w) + (A(s,a;w) - \frac{1}{A}\sum_{a'} A(s,a';w))$$

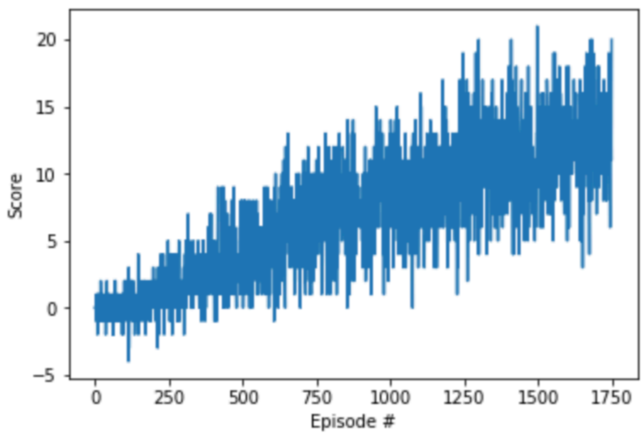
2.4 – Hyperparameters

In order to gain optimal performance, hyperparameters tuning is very important. The following hyperparameters were set in the solution:

```
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
```

```
def dqn(agent, n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.999, train=True):
    """Deep Q-Learning.

    Args
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
        train (bool): flag deciding if the agent will train or just play through the episode
    """
```



2.3 – Plot of Rewards

The environment considered as being solved if an agent would get an average of +13 reward over 100 consecutive episodes. The result is +13.00 reward in 1751 episodes, implemented by DQN Dueling Network combination.

At the first 100 episodes, the agent presented a very weak average score (0.05), however after increasing the number of episodes, it started to converge faster and increased +1 average score at every 100 episodes

4 – Ideas for Future Work

For further performance improvement, the implementation of Double DQN and Prioritized Experience Replay would be a great start in order to increase the learning speed.