Phase 3 Deep Reinforcement Learning Nanodegree Scholarship from Facebook
NANODEGREE PROGRAM by Udacity

PROJECT
NAVIGATION

By Sabrina Palis

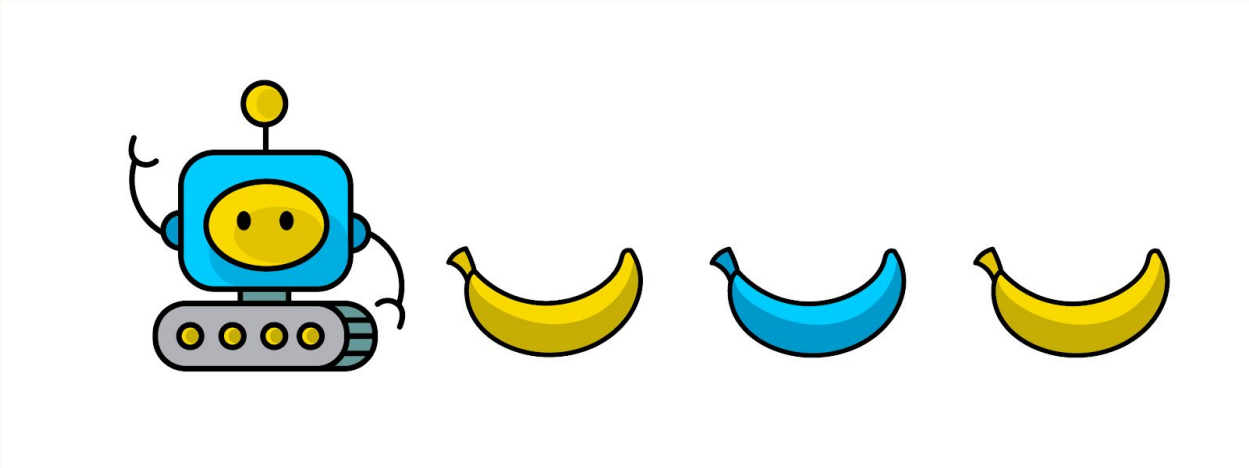


Image credit: Sabrina Palis

INTRODUCTION

For this project, I will train an agent to navigate and collect bananas in a large, square world.
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 the 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:

- **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.

Screenshot of the Unity Environment

Implementation

1. Deep Q-learning

Q-learning is an effective Temporal Difference Learning method that uses an update rule to approximate the optimal value function at every time-step.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \right)$$

Deep Q-learning

The issue with Q-learning is that it is better applied to discrete states and actions. In a continuous environment, neural networks provide a solution to this limitation by allowing the use of function approximations.

The Q-network architecture

In the Navigation project, I will use a neural network made of 3 fully-connected linear layers also called dense layers, the first two of these followed by a ReLU activation function, as shown in the pseudo-code summary below.

```
QNetwork

(fc1): Linear (in_features=37, out_features=256)

ReLU

(fc2): Linear(in_features=256, out_features=128)

ReLU

(fc3): Linear(in_features=128, out_features=4)
```

The in_features of the first fully-connected linear layer correspond to the 37 dimensions of the environment. The layers are used to build up progressively more abstract representations. The out_features of the third fully-connected linear layer correspond to the 4 possible actions of the agent.

Therefore the deep Q-learning algorithm represents the optimal action-value function q* as a neural network. It takes the state as input and returns the corresponding predicted action values for each possible action.

2. Stabilizing Reinforcement Learning

Reinforcement Learning is notoriously unstable when neural networks are used to represent action values. In order to address these instabilities, we use **experience replay** and **fixed Q-targets**.

Experience replay

Experience replay is implemented by keeping track of a replay buffer and using experience replay to sample from buffer at random. It prevents action values from oscillating or diverging catastrophically.

In addition to breaking harmful correlations, experience replay allows to learn more from individual tuples multiples times, recall rare occurrences, and make better use of the experience.

Fixed Q-Targets

In Q-Learning, we update a guess with a guess, and this can potentially lead to harmful correlations. To avoid this, we can update the parameters w in the network q^ to better approximate the action value corresponding to state S and action A with the following update rule:

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

The fixed target implementation is a way to solve the moving target problem and reducing the loss Δw. Small updates are made to the weights of the target network by introducing a hyperparameter tau.

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha[\underbrace{(\underbrace{r_t + \max_a \hat{Q}(s_{t+1}, a; \theta')}_{\text{target}} - Q(s_t, a_t; \theta))}_{\text{TD-error}}]$$

We sample the environment by performing actions and store away the experienced tuples in a replay memory.

We select the small batch of tuples from this memory randomly, and learn from that batch using a gradient descent step.

These two processes are not directly dependent on each other, so you could perform multiple sampling then one learning step or even multiple learning steps with different batches. The rest of the algorithm is designed to support these steps.

Algorithm

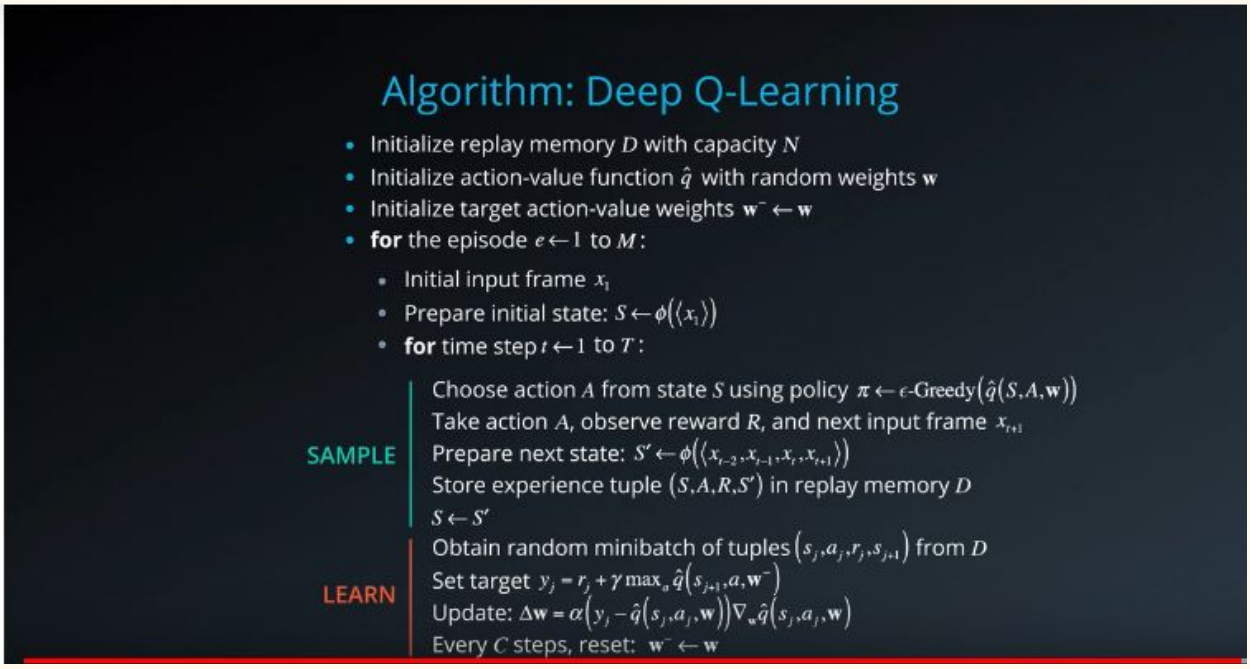


Image Credit: Udacity Deep Reinforcement Learning for Enterprise Part 2 Lesson 2.5

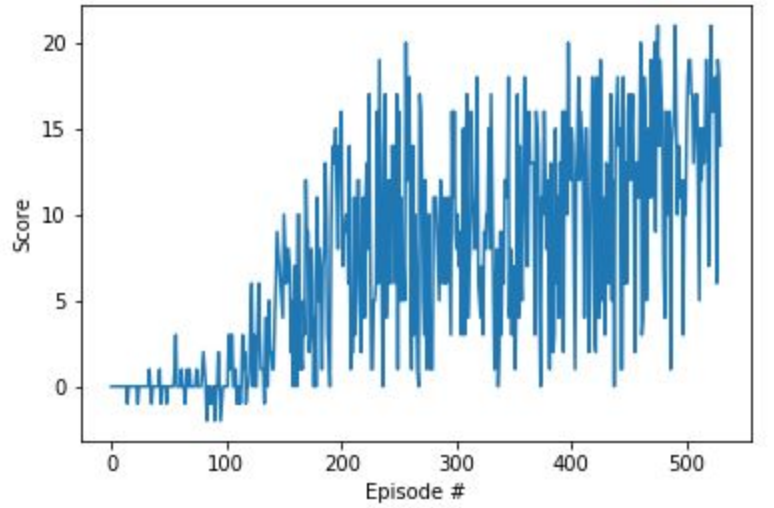
3. Chosen hyperparameters

- **BUFFER_SIZE = 1e5** **Number of samples in the memory buffer**
- **BATCH_SIZE = 64** **Minibatch size**
- **GAMMA = 0.99** **Discount factor**
- **TAU = 1e-3** **For soft update of target parameters**
- **Adam optimization**
- **LR = 5e-4** **Learning rate**
- **UPDATE_EVERY = 4** **How often to update the network**
- **eps_start=1.0** **Epsilon start for the epsilon greedy policy**
- **eps_decay=0.995** **Epsilon decay for the epsilon greedy policy**
- **eps_end=0.01** **Epsilon end for the epsilon greedy policy**

Plot of rewards

The following plot of rewards per episode is included illustrates that the agent is able to receive an average reward (over 100 episodes) of at least +13.

430 episodes were needed to solve the environment.



Ideas for Future Work

Double Q-Learning would help with the tendency of Deep Q-Learning to overestimate action values.

Prioritized experience replay could help the agent to learn more effectively from some transitions than from others.

Dueling architecture would allow to assess the value of each state without having to learn the effect of each action.