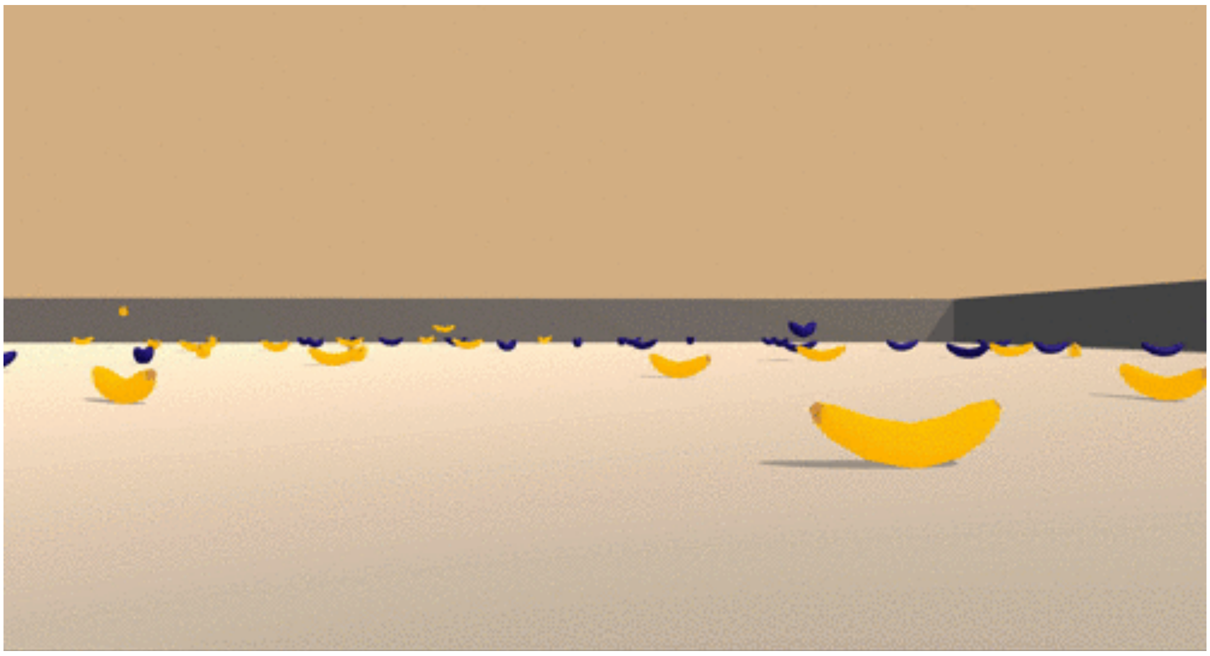
Deep reinforcement learning nanodegree
Project 1 - Navigation

1. Summary

This project required the training of an agent to navigate (and collect bananas) in a large, square world.



The environment is such that 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 (300 steps) and, in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

The learning algorithm used was the deep Q network (DQN). I also implemented an improvement to the original DQN algorithm (double-Q).

2. Learning algorithm

The DQN algorithm addresses the fundamental limitation of the Q-learning method, which struggled to deal with situations when the number of observable states is very large. A good example of such a situation is an Atari game, with many different screens defined by raw pixels.
The solution presented by DQN involves the use of a nonlinear representation that maps both the state and action onto a Q value. The best way to build such a representation is, of course, the use of a deep neural network.
In order to make DQN work satisfactorily, three adjustments are necessary:

I. Introduction of the epsilon-greedy method, which means switching between a random policy and a greedy Q policy according to a probability ε. By varying ε, we can control the degree of randomness. The best approach is to start with total randomness (ε = 1) and decrease ε as learning progresses. Generally, we use a terminal ε value between 0.02 and 0.05. In essence, the introduction of the hyperparameter ε in the epsilon-greedy method helps us solve the dilemma exploitation versus exploration.
II. The SGD (stochastic gradient descent) optimization used to calculate the weights of a deep neural network requires training data that is both independent and identically distributed. In our case, data samples from a same episode tend to be very close to each other and, thus, highly correlated. To address this issue, we use a replay buffer of a fixed size. In other words, we simply sample from the replay buffer while fresh data is added to it and stale data is removed. This allows the use of relatively fresh data that has a higher degree of independency. The size of the replay buffer represents the introduction of another additional hyperparameter (buffer size) and defines the degree of in independency among samples.
III. The error term from the Bellman equation is given by $(r + \gamma Q_{max}\,(s', a')) - Q(s, a)$, where γ is the discount factor for future actions. If we use the same neural network to map both the target and the local term, we're going to have a situation in which we try to get to a moving target every time the weights are updated. To solve this issue, we use a second network (target network) with the same architecture for the term $Q_{max}\,(s', a')$. The weights of the target network are updated less often, according to a new hyperparameter τ. The term $Q(s, a)$ continues to be mapped with the original network (local network)

The steps used in the DQN algorithm are as follows:

I. Initialize both the target and the local network with random weights and set ε=1. At this point, the replay buffer is completely empty.
II. Reset the environment and observe state s.
III. Select either a random action or action = argmaxQ(s, a) with probability ε.
IV. In the environment, execute action a, observe reward r and move to next state s'.
V. Store (s, a, r, s', done) in the replay buffer. The flag done indicates whether or not the episode has reached a conclusion.
VI. Sample a mini batch of (s, a, r, s', done) from the replay buffer. If the size of the buffer is smaller than the size of the minibatch, wait for the buffer to grow.
VII. For every (s, a, r, s', done) in the mini batch, calculate the target $(r + \gamma Q_{max}\,(s', a'))$. Remember that, if done=True, the target is just the value of the reward r.

VIII. Calculate $Q(s, a)$ for every (s, a, r, s', done) in the mini batch.

IX. Calculate the loss (target - $Q(s, a)$)^2

X. Update the local network via SGD (or Adam) for every n steps (in our case, n = 4)

XI. Partially copy the updated weights to the target network, according to the τ hyperparameter: $\text{weights\_target}_{t+1} = (1-\tau)\ \text{weights\_target}_t + \tau\ \text{weights\_local}_{t+1}$

XII. Update ε according to a decay term if ε is greater than the minimum ε (do this at the end of every episode)

XIII. Repeat from step III.

In addition to algorithm above, I also implemented the double-Q improvement. The idea behind double-Q is that the standard DQN algorithm tends to overestimate Q values, which can be detrimental to training performance and lead to suboptimal policies. In order to solve this issue, instead of using the max Q values in the target term $(r + \gamma Q_{max}\ (s', a'))$ we calculate the best actions for s' using the local network but evaluate the respective Q values using the target network.

3. Neural network architecture

Given that the states have only 37 dimensions, a network with three fully connected hidden layers of size 512, 128 and 32 is more than enough to handle the task. After each hidden layer, I applied a Relu activation function.

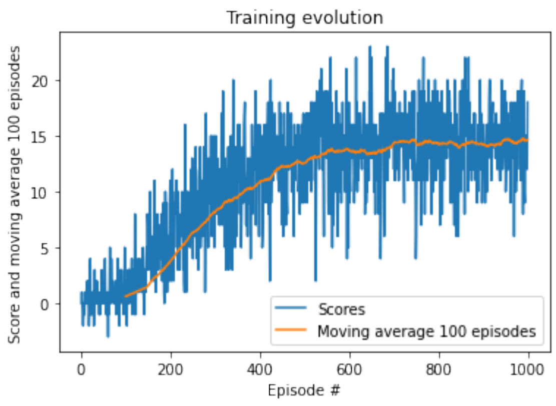
4. Hyperparameters

Hyperparameter	Description	Value
Initial ε	Initial value of the ε parameter. A value of 1 means a totally random policy.	1.0
ε decay	Defines the rate of decrease for ε after each episode.	0.995
Minimum ε	Keeps a minimum level of exploration and help to avoid a situation in which we can become stuck in the local peak of a lousy policy.	0.05
γ (gamma)	Discount factor for future actions.	0.99
τ (tau)	Defines the lag in the updated of the weights of the target network.	1e-3
Buffer size	How many samples we're going to keep in the replay buffer.	100,000
Batch size	Size of the mini batch sampled from the replay buffer	64
Update_every	Defines the network update for every n steps	4
Learning rate	Hyperparameter of the deep neural network	2.5e-4

5. Training results

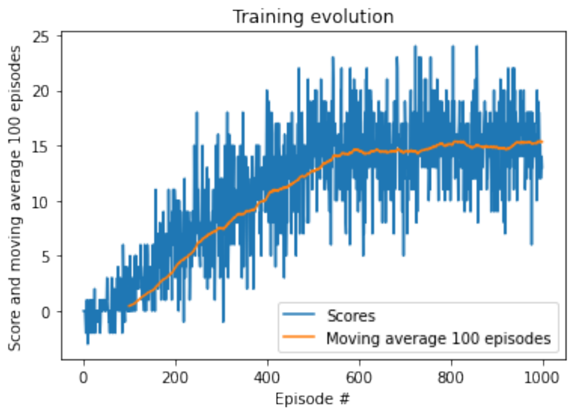
The vanilla agent achieved an average score greater than 13 over 100 episodes in the episode number 515. The training progression is shown below.

Vanilla agent solved environment after 515 episodes

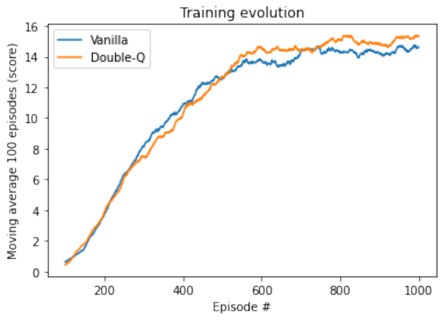


The double-Q agent solved the environment after 518 episodes. So, there was no improvement over the vanilla agent.

Double-Q agent solved environment after 518 episodes



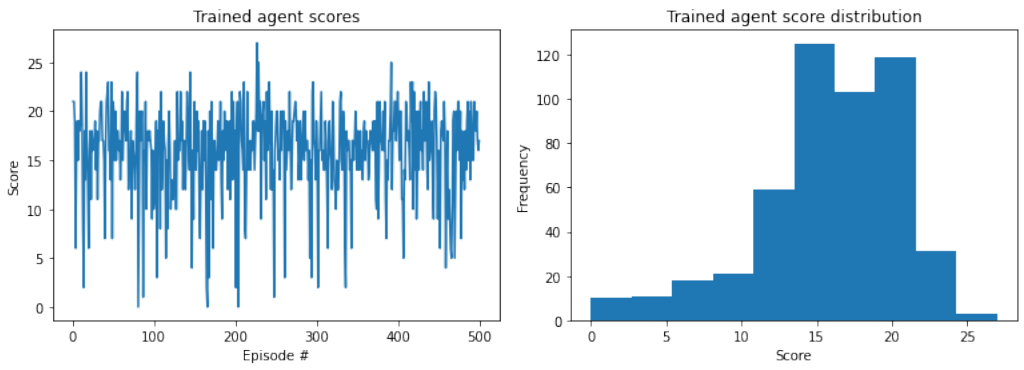
Plotting the moving averages of the training scores for both agents, we can clearly see that, for this environment and network architecture, the use of a double-Q agent does not introduce any significant improvement in training performance.



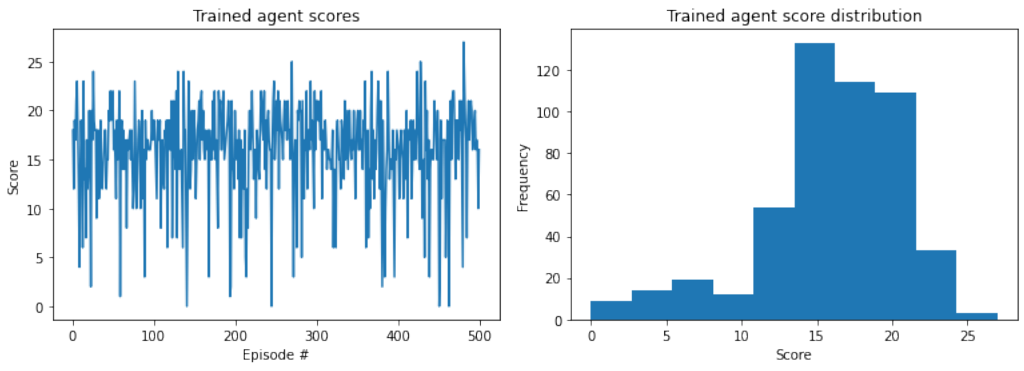
6. Performance of trained agents

Observing the two agents over 500 episodes, we can see that the vanilla agent achieved an average score of 15.88 while the double-Q agent achieved an average score of 15.92. Given the high standard deviation observed for both agents we cannot say categorically that the vanilla agent has an edge. In fact, the performances were almost identical.

Vanilla agent performance over 500 episodes – Average score 15.88 and standard deviation 4.69



Double-Q agent performance over 500 episodes – Average score 15.92 and standard deviation 4.70



7. Ideas for future work

- Find the optimal size of the neural network;
- Implement the prioritized replay buffer improvement;
- Implement the dueling DQN improvement;
- Implement the pixel method.