Robot Navigation using Deep Q-Learning

By

Wael Farag

1. Goal

In this project, I built a reinforcement learning (RL) agent that navigates an environment. This environment is similar but not identical to the Unity's Banana Collector environment [1].

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. The goal of the developed agent is to collect as many yellow bananas as possible while avoiding blue bananas. In order to solve the environment, the agent must achieve an average score of +13 over 100 consecutive episodes.

2. Approach

The following are the high-level steps that has taken to build the agent that solves this environment:

1. Evaluating the state and action space by randomly selecting actions and aggregating the collected rewards till the end of the episodes.
2. Establishing a baseline using the above random action policy.
3. Implementing a learning algorithm using a Deep Q-Learning Network.
4. Running experiments to measure the implemented agent performance.
5. After training, testing the learned agent using random episodes.

3. Evaluating the State-Action Space

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:

- `0` move forward
- `1` move backward
- `2` turn left
- `3` turn right

4. Implementing the Learning Algorithm

Agents use a policy to decide which actions to take within an environment. The primary objective of the learning algorithm is to find an optimal policy, i.e., a policy that maximizes the collective rewards for the agent. Since the effects of possible actions aren't known in advance, the optimal policy must be discovered by interacting with the environment and recording observations. Therefore, the agent "learns" the policy through a process of trial-and-error that iteratively maps various environment states to the actions that yield the highest reward. This type of algorithm is called “Q-Learning” [2].

Q-learning is a model-free reinforcement learning algorithm [3]. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

As for constructing the Q-Learning algorithm, the general approach is to implement a handful of different components, then run a series of tests to determine which combination of components and which hyper parameters yield the best results.

In the following sections, we'll describe each component of the algorithm in detail.

The Epsilon Greedy Algorithm:

One challenge with the Q-function above is choosing which action to take while the agent is still learning the optimal policy. Should the agent choose an action based on the Q-values observed thus far? Or, should the agent try a new action in hopes of earning a higher reward? This is known as the **exploration vs. exploitation dilemma** [2].

To address this, an **ε-greedy algorithm** is implemented. This algorithm allows the agent to systematically manage the exploration vs. exploitation trade-off. The agent "explores" by picking a random action with some probability epsilon **ϵ**. However, the agent continues to "exploit" its knowledge of the environment by choosing actions based on the policy with probability (1-**ϵ**) [2].

Furthermore, the value of epsilon is purposely decayed over time, so that the agent favors exploration during its initial interactions with the environment, but increasingly favors exploitation as it gains more experience. The starting and ending values for epsilon, and the rate at which it decays are three hyperparameters that are later tuned during experimentation.

Deep Q-Network (DQN):

With Deep Q-Learning, a deep neural network is used to approximate the Q-function. Given a network F, finding an optimal policy is a matter of finding the best weights "*w"* such that $F(s, a, w) \approx Q(s, a)$.

The neural network architecture used for this project can be found in the "model.py" file of the source code. The network contains four layers and implemented with different topologies (will be discussed later).

As for the network inputs, rather than feeding-in sequential batches of experience tuples, random samples from a history of experiences are fed to the network during training in an approach called Experience Replay [4].

Experience Replay:

Experience replay allows the RL agent to learn from past experience [4]. Each experience is stored in a replay buffer as the agent interacts with the environment. The replay buffer contains a collection of experience tuples with the state, action, reward, and next state (s, a, r, s'). The agent then samples from this buffer as part of the learning step. Experiences are sampled randomly, so that the data is uncorrelated. This prevents action values from oscillating or diverging catastrophically, since a naive Q-learning algorithm could otherwise become biased by correlations between sequential experience tuples.

Also, experience replay improves learning through repetition. By doing multiple passes over the data, our agent has multiple opportunities to learn from a single experience tuple. This is particularly useful for state-action pairs that occur infrequently within the environment.

The implementation of the replay buffer can be found here in the "agent.py" file of the source code.

5. Experimentations and Results

The Deep Q-Learning Network (DQN) is implemented with several sizes and trained with these topologies. Table 1 presents the training results which shows clearly that the DQN size does not need to be big in size, in the contrary the small size networks (#2, #3, and #4) shows significantly faster and better performance than the big ones (#1). All the networks are trained with the hyper parameters listed in Table 2.

Table 1 Training Results for Different DQN Topologies.

#	No. Layers	Size	Episodes to reach 13	Training Time	Comment
1	4	Input Layer: 37 1st Hidden Layer: 1024 2nd Hidden Layer: 1024 Output Layer: 4	963 960	21.3 min 111.9 min	With GPU With CPU
2	4	Input Layer: 37 1st Hidden Layer: 48 2nd Hidden Layer: 24 Output Layer: 4	506	10.1 min	With GPU
3	4	Input Layer: 37 1st Hidden Layer: 24 2nd Hidden Layer: 12 Output Layer: 4	524	10.6 min	With GPU
4	4	Input Layer: 37 1st Hidden Layer: 12 2nd Hidden Layer: 6 Output Layer: 4	511	10.3 min	With GPU

Table 2 Hyper-parameters Values.

Hyper parameter	Value
Replay Buffer Size	100,000
Batch Size	64
Discount Factor (γ)	0.99
Soft-update for Target Parameters (τ)	0.001

Learning Rate	0.0005
How often to update the network	Every 4

The DQN (#4) has been then trained to reach another prominent goal (a collective reward of 16.0 instead of 13.0). Figure 2 and Figure 2 depict the training performance of the DQN to reach the goal of 16. The goal of 13.0 was already reached before reaching 600 Episodes.

```
Episode 100     Average Score: 0.74
Episode 200     Average Score: 2.59
Episode 300     Average Score: 4.59
Episode 400     Average Score: 8.05
Episode 500     Average Score: 10.73
Episode 600     Average Score: 13.22
Episode 700     Average Score: 13.37
Episode 800     Average Score: 13.87
Episode 900     Average Score: 14.49
Episode 1000    Average Score: 13.81
Episode 1100    Average Score: 15.46
Episode 1200    Average Score: 15.06
Episode 1300    Average Score: 14.14
Episode 1400    Average Score: 14.61
Episode 1500    Average Score: 14.90
Episode 1600    Average Score: 15.33
Episode 1700    Average Score: 14.86
Episode 1800    Average Score: 15.94
Episode 1900    Average Score: 14.75
Episode 1949    Average Score: 16.01
Environment solved in 1849 episodes!    Average Score: 16.01

Total Training time = 42.2 min
```

Figure 1 Training Performance of the DQN(#4) to reach a 16.0 reward.

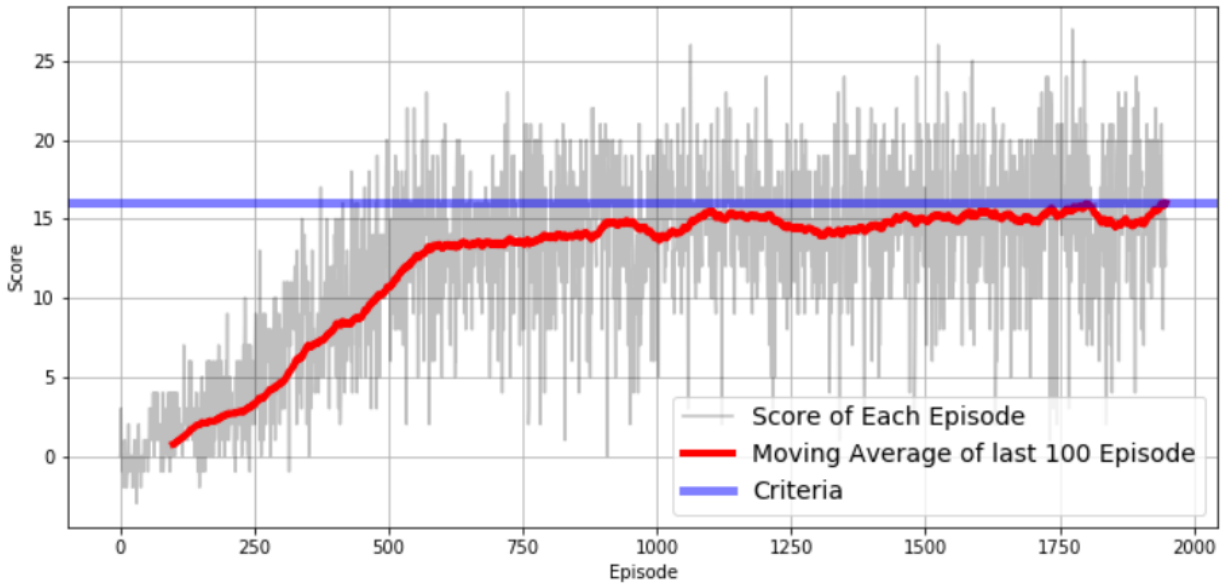

Figure 2 Training Performance of the DQN(#4) to reach a 16.0 reward.

4. Evaluation and Testing

The DQN(#4) is also tested after being trained using 10 randomly generated episodes with a different seed (30). The network was already trained with seed (10). Figure 3 and Figure 4 depict the testing results, showing that some of the episodes collect a reward pf 20.0.

```
Episode 1       Average Score: 11.00
Episode 2       Average Score: 15.50
Episode 3       Average Score: 16.33
Episode 4       Average Score: 17.00
Episode 5       Average Score: 17.60
Episode 6       Average Score: 17.17
Episode 7       Average Score: 17.43
Episode 8       Average Score: 17.38
Episode 9       Average Score: 16.67
Episode 10      Average Score: 17.00
```

Figure 3 Testing Performance of the DQN(#4) for 10 individual Episodes.

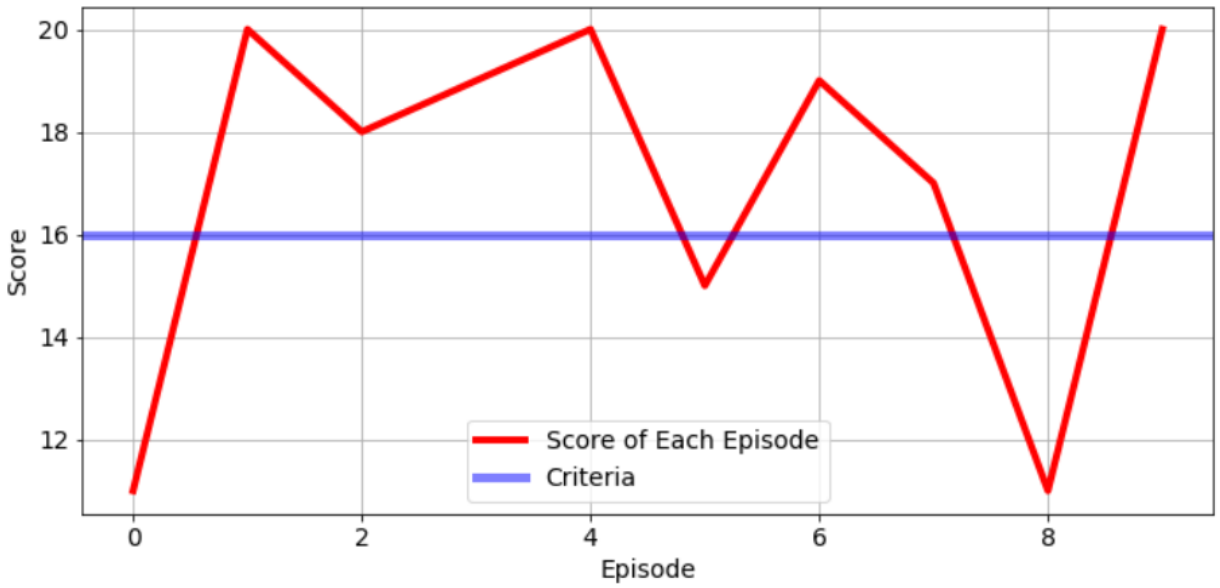

Figure 4 Testing Performance of the DQN(#4) for 10 individual Episodes.

4. Future Improvement

There are several suggestions that can be carried out to further understand and improve the performance of the agent:

1. Using another network topology like Dueling DQNs, and Double DQNs.
2. Test the replay buffer — Implement a way to enable/disable the replay buffer to measure the impact the replay buffer on performance.
3. Add prioritized experience replay — Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare and important experience vectors are sampled.
4. Replacing conventional exploration heuristics with Noisy DQN — This approach improves exploration phase. The key point is that parametric noise is added to the weights to induce stochastic nature to the agent's policy, yielding more efficient exploration.

References

[1] Unity's Banana Collector environment, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#banana-collector, retrieved on Oct. 5th, 2019.

[2] Richard S. Satun, Andew G. Barto, "Rienforcement Learning – An Introducation", 2nd Edition, The MIT Press, 2018.

[3] Q-learning, wikipedia, https://en.wikipedia.org/wiki/Q-learning, retrieved on Oct. 5th, 2019.

[4] Ruishan Liu, James Zou, "The Effects of Memory Replay in Reinforcement Learning", arXiv:1710.06574v1 [cs.AI] 18 Oct 2017.