Deep Reinforcement Learning Nanodegree – Navigation

Udacity Nanodegree Program

Prabowo Setiawan

May 19, 2020

Introduction

This report aims to examine and discuss the navigation project, both from the implementation and the results. The navigation project entails training an agent in navigating a large square world. In this large square world, there are two different bananas with corresponding rewards associated with them. The agent is able to obtained rewards as follows:

- +1 reward for collecting yellow bananas
- -1 reward for collecting purple bananas

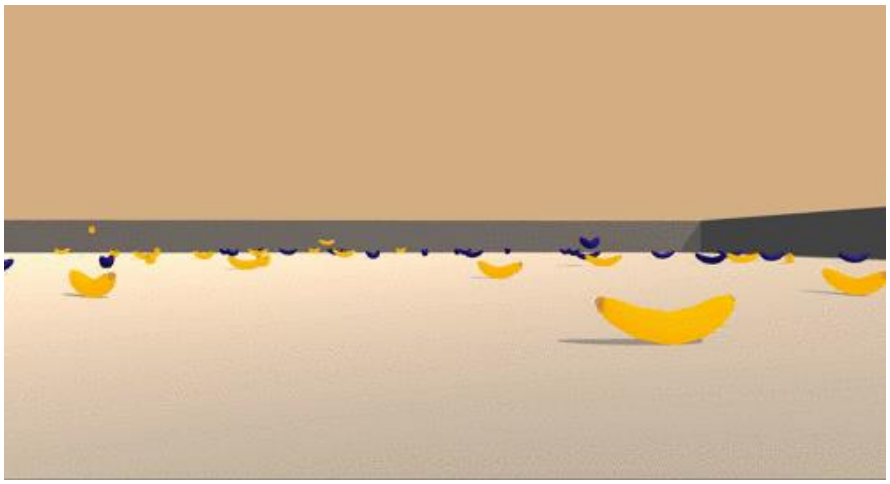

Figure 1: The Unity environment

In each episode, an agent is expected to obtain as many rewards or scores as possible. There are several ways that the agent can navigate or move through this environment. These are consisted of four discrete actions:

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

These actions can be taken once the agent observes the environment. The state space of the environment consists of 37 dimensions; these include the agent's velocity and ray-based perception of objects surrounding the agent's forward direction. The agent is considered to solve the environment once it attains an average score of at least 13 over the past 100 consecutive episodes.

Project Approach

In order to solve the environment, there are two implementations of value-based methods that are explored in this project:

- Deep Q-Network (DQN)
- Double Deep Q-Network (DDQN)

DQN is a Q-learning algorithm which uses deep neural network to approximate the Q-value function. The architecture accepts the state as the input and provides the Q-values of the possible discrete actions as the output. DDQN on the other hand, uses another deep network to address the issue of DQN overestimating the Q-values. Theoretically, the result of DDQN should be superior in comparison to the DQN's. Both methods are implemented using the same network. The network's architecture is as follows,

- Input: size of state space
- Layer 1: 128 layers with ReLU activation function
- Layer 2: 64 layers with ReLU activation function

- Layer 3: 32 units with ReLU activation function
- Output: size of possible discrete actions

This combination is chosen after experimenting with several other variants (2 hidden layers, different number units for each layer, etc.) Although all of these configurations perform relatively well, this specific architecture yields the lowest number of episodes required to finish the training process.

The next parameter to be determined is the epsilon, ε, for epsilon-greedy choosing actions. There are three different parameters needed to specify ε:

- ε_0 (initial epsilon) = 1.0
- ε_{decay} (decaying factor of epsilon) = 0.97
- ε_{min} (minimum value of epsilon) = 0.005

The epsilon value initially starts with ε_0 to enforce equal amount of exploration within the training environment. As the agent is being trained, the value of epsilon is decayed by the factor until it reaches the minimum value of epsilon. This allows the algorithm to explore at the beginning of the training process and proceed to exploit actions with highest expected rewards. Other parameters such as the learning rate, batch size, and others are determined through trial and error.

The next aspect of the approach is the implementation of replay buffer. After each action taken, the states, actions, rewards, next states and done condition of the training status is appended to the memory. Once this number equals to or larger than the batch size, the agent decides to learn from those random sampled memory. The update on the Q-values is executed every 4 steps taken. On the other hand, random action is selected if the size of the memory is lower than the batch size.

Results

Based on the agent trained using the chosen hyperparameters and random seed number, the regular DQN performs better than its variant, DDQN. The consistency of both methods is promising, yet the DQN has shown its learning capabilities to be more consistent. The following graphs show these differences.

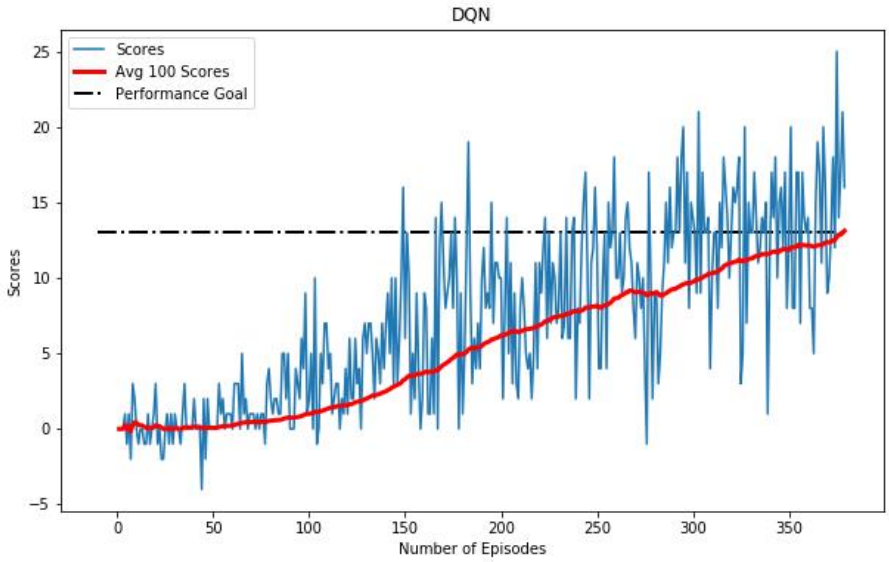

Figure 2: DQN Scores and Average Scores

A regular DQN with implementation of replay-buffer performs quite well. It only needs 379 episodes to attain the training criteria. It finishes the training with an average score of 13.12. It can be seen that the agent learns

consistently with increasing average scores most of the time. It is noted that the scores per episode are fluctuating since the underlying action taken by the agent can still be determined by random actions due to the epsilon-greedy policy.

The next method to be examined is the DDQN. Surprisingly, even though DDQN should address the problem of DQN's tendency to overshoot, the agent solved the environment in 452 episodes with average final score of 13.00. This is 73 more episodes in comparison to the regular DQN. Although it wasn't able to complete the environment as fast as the regular DQN, it is essential to note that this performance is still impressive.

This result can be seen from Figure 3 below. The average scores growth for this particular method is not as consistent as the regular DQN. This occurs especially around episodes 250-320.

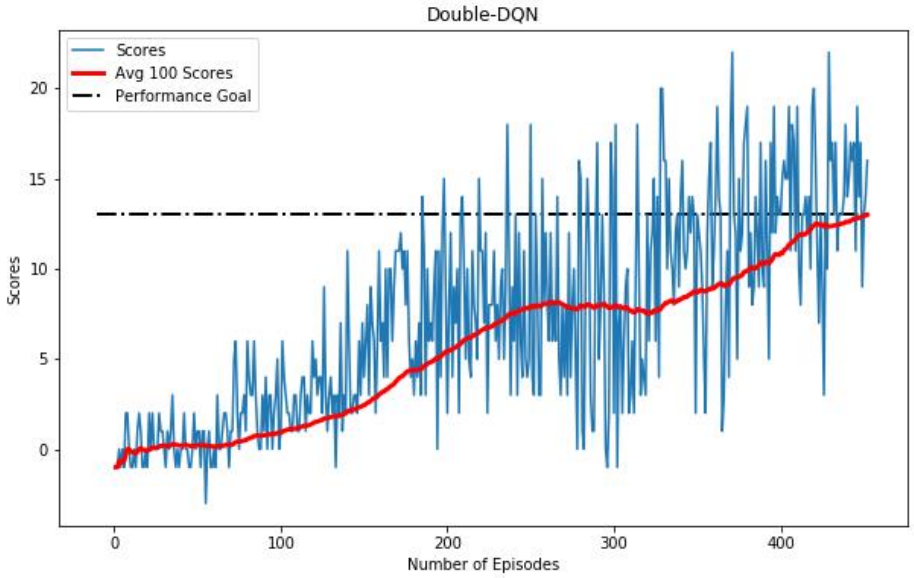

Figure 3: DDQN Scores and Average Scores

Discussion

From the results above, it is concluded that the DQN outperforms DDQN, although not by a large margin. The plots available suggest that DQN has a better consistency in terms of learning from the environment. Although this is a substantial comparison between the performance of the two methods, there is another aspect of this project that needs to be addressed. This important aspect is the parameter of the DQN and DDQN.

In the process of completing the project, numerous combinations of hyperparameters were implemented for DQN. Among 20 or so combinations, this hyperparameters were found to be the most optimal:

1. epsilon = 1.0
2. eps_min = 0.005
3. eps_decay = 0.97
4. gamma = 0.99
5. tau = 0.001
6. learning rate = 0.0005
7. Deep QNetwork:
    - hidden_layer_1 = 128, with ReLU
    - hidden_layer_2 = 64, with ReLU
    - hidden_layer_3 = 32, with ReLU

- output_layer has no activation function

8. targetUpdateNet = 4
9. seed = 1

These hyperparameters can be easily tweaked and resulting in the Double-DQN outperforming the regular DQN. Although with the nature of complexity, the DQN is slightly easier to implement and therefore would still most likely to be the more efficient option.

Future Improvements

One thing to note is that using different seed numbers sometimes lead to a drastic increase of number of episodes needed in finishing the training process. Therefore, it might be in the best of interest to look into stabilizing the training process of this method. One way to implement this would be to introduce the Dueling Network Architectures. This architecture provides two separate estimators, one for state-value function and the other for state-dependent action advantage function, which leads to a better generalization in training the agent (Wang et al, 2016). Another interesting research subject to look into would be the prioritized experience replay so that the network can sample more important experience from the replay memory instead of sampling it randomly (Schaul et al, 2016).

References

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. 2016. Dueling Network Architectures for Deep Reinforcement Learning. [Online]. Google DeepMind, London, United Kingdom. [Accessed 19 May 2020]. Available from: https://arxiv.org/pdf/1511.06581.pdf

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. 2016. Prioritized Experience Replay. In: *ICLR 2016*. [Online]. [Accessed 19 May 2020]. Available from: https://arxiv.org/pdf/1511.05952.pdf