Deep Reinforcement Learning Nanodegree
Project 1 – Navigation Report

Chris Cadonic
chriscadonic@gmail.com

July 11, 2020

1 Introduction

As part of the first project for the deep reinforcement learning nanodegree, I built and trained an agent that learns how to navigate through a world with yellow and blue bananas in an effort to maximize how many yellow bananas it can acquire.

1.1 Environment Overview

In the Unity-ML environment provided, the agent navigates through a world filled with yellow and blue bananas. As the agent moves through the world, if it comes into contact with a banana it will pick it up, receiving a reward of +1 if it picks up a yellow banana and -1 if it picks up a blue banana. Its goal is to maximize its reward for a single play session where it navigates and tries to pick up as many yellow bananas while avoiding blue bananas. The game is considered solved if it achieves an average score over 100 games of $+\mathbf{13}$.

As the agent interacts, it receives information from the environment of the form: **insert state info**

Interactions available to the agent include moving in one of four directions:

- move forward,
- move backward,
- turn left,
- turn right,

with these four actions forming the action space $\mathcal{A}$. The agent solves the game by understanding what the optimal action is to take in each of the possible states that it finds itself in. An example image of the game is shown below in Fig. 1.

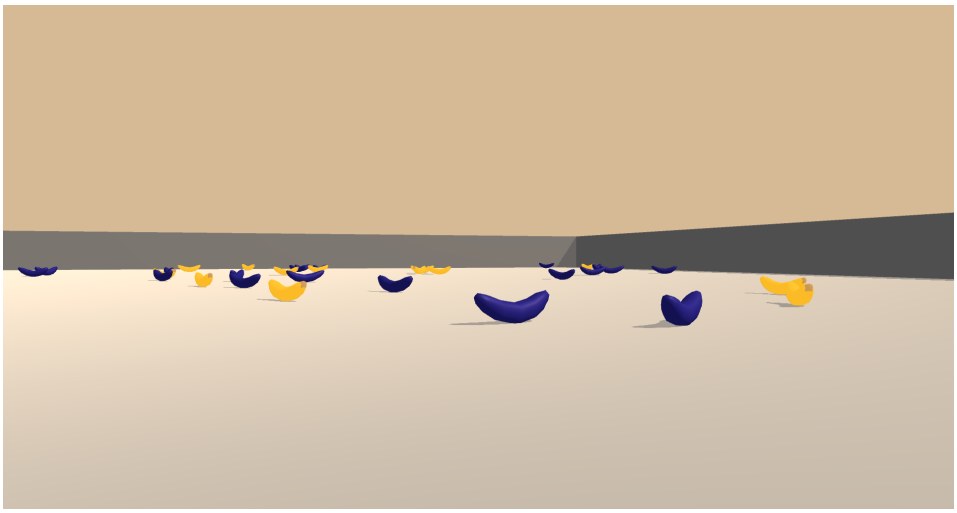

Figure 1: A snapshot of the banana navigation game.

2 Approach

Initially I had approached this problem by first attempting to use Q-learning as a first benchmark, but given the continuous nature of the state space $\mathcal{S}$ this would also necessitate discretization or tile coding the space. For this reason, I instead chose to implement a vanilla DQN (Deep Q-network) and iteratively improve it to achieve better performance.

2.1 DQN

My first complete algorithm for DQN leveraged a simple multi-layer perceptron using the following architecture:

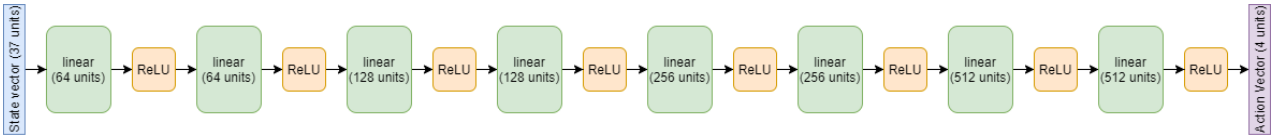

Figure 2: The 8-layer MLP used in the DQN algorithm.

I chose to experiment with a deeper model that leveraged increasingly wider linear layers, ending up with an 8-layer neural network with units of sizes [64, 64, 128, 128, 256, 256, 512, 512] and *ReLU* activation functions between each layer. This architecture was experimented with and ultimately chosen to capture ever-increasingly complex relationships between the state values in the input, determining more high-order information between ray projections as states are passed further into a deeper network.

I also needed to experiment hyperparameter tuning for the following hyperparameters:

hyperparameter	**values**
ϵ_{decay}	$[0.99, 0.996, 0.999, 0.9999, 0.99999]$
α	$[0.00005, 0.0001, 0.0005, 0.001, 0.005]$
τ	$[0.005, 0.01, 0.1]$
γ	$[0.8, 0.9, 0.95, 0.99]$

Table 1: Hyperparameters experimented with to train an agent using DQN.

Even though learning performance from the vanilla DQN was quite good (see results in Fig. **??**), I wanted to see explore options to determine if performance could be improved.

2.2 Improving DQN

Since I had the least experience with implementing prioritized experience replay, this was the first method I chose to expand on my vanilla DQN.

2.2.1 Prioritized Experience Replay

Using prioritized experience replay expands on the idea of sampling from the replay buffer to provide experience tuples for the agent to learn from. In the original DQN algorithm, this sampling is uniform across all stored experience tuples in the replay buffer, whereas prioritized experience replay *prioritizes* experience tuples that resulted in a larger loss from expected Q-values, capturing that these tuples may have been more impactful during the learning process and will likely be more useful to sample more often than lesser impactful experience tuples.

In order to expand on my initial implementation, I changed the following: - storage of

3 Results and Discussion

3.1 Learning Performance

The results in Fig. 3 show that the the vanilla DQN algorithm with the architecture shown in Fig. 2 achieved quite good performance, solving the navigation problem in 1185 episodes. That is, achieving an average score of +13 over 100 episodes after the 1185th episode.

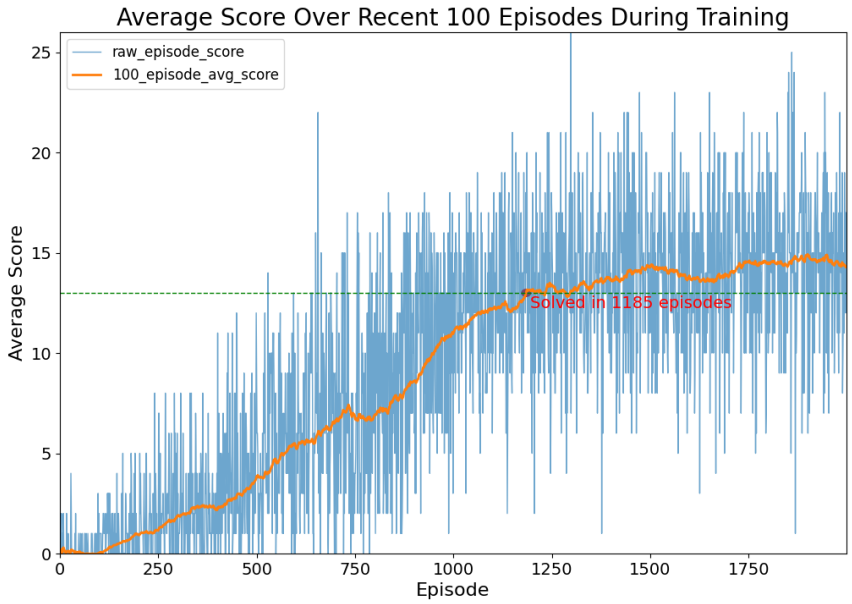

Figure 3

After running experimentation over the parameters listed in Tbl. **??** the optimal parameters were:

- $\epsilon_{decay} = 0.996$,
- $\alpha = 0.001$,
- $\tau = 0.01$,
- $\gamma = 0.99$.

Evident in the learning performance of the agent, not only was the agent able to solve the navigation problem fairly quickly, but it was able to continue to improve beyond this, likely because I also enforced a minimum value on $\epsilon_{min} = 0.05$ to ensure that the agent is never *entirely* acting greedily.

Although performance tends to improve consistently over the 2000 episode training period, there is quite a bit of variation in the raw episode scores indicated by the light blue line. This is likely due to the aforementioned lower bound of 0.05 on ϵ, which means that the agent still acts completely random 5% of the time. This lower bound is reached fairly rapidly with $\epsilon_{decay} = 0.996$, at approximately episode 747 (i.e., $log_{0.996}(0.05) \approx 747$).

3.2 Next Steps

There remains room for improvement for the algorithm, although through experimentation with a vanilla DQN I was able to achieve fairly good results. Some logical next steps, however, would include:

- implementing double DQN to assist with any value overestimation that can occur in vanilla DQN,
- implementing duelling DQN to provide better separation between favourable actions in states, though this may not provide an extreme improvement other than more rapid learning due to the small action space.

With regard to experimentation and overall improvement of the DQN algorithm at hand, it may make sense to experiment more with the rate at which the target network is updated a bit more. Currently, the target network is not only updated on a static basis of every 100 iterations, but I also use a soft-update that doesn't completely transfer weights from the Q-network to the target network but rather transfers as in:

$$\theta_{target} \leftarrow (1 - \tau)\theta_{target} + \tau\theta_q \tag{1}$$

for target network weights θ_{target} and Q-network weights θ_q.

Additionally, it may be interesting to determine if the algorithm may be able to improve if the learning rate α was adaptive rather than static, which may enable the algorithm to learn less and less. This could provide a better overall agent in the end, particularly if ϵ is decayed to a smaller lower bound, since then the agent could learn from exploration sufficiently well and when it has reached a point that it will no longer be learning much from new actions, it has little to gain from taking completely random actions.

4 Conclusion

References