Udacity Navigation Project Report

1. Introduction

The goal of this project was to train an agent to navigate and collect bananas in the Unity banana environment – a large square box with 4 walls and yellow and blue bananas distributed throughout the box. The agent receives a +1 reward for collecting a yellow banana, and a -1 for collecting a blue banana. So the agent must navigate the box to collect as many yellow bananas as possible while avoiding blue bananas at all costs.

The state space has 37 dimensions with 35 ray-based vectors that detect objects and obstacles in the agent's 180 degree view, and 2 dimensions for forward/backward velocity. The agent has 4 actions available; 0: forward | 1: backward | 2: left | 3: right.

This is an episodic task that is solved if the agent achieves an average score of +13 over 100 consecutive episodes.

2. Model

For my work I leveraged information from this article;

Amit Patel, Sep 23, 2018 – Navigation: Banana Collection Agent
https://medium.com/@amitpatel.gt/double-dqn-48562b5f31c1#:~:text=Using%20a%20simplified%20version%20of,for%20collecting%20a%20blue%20banana.

I initially implemented a 'vanilla' DQN with experience replay, fixed Q-target, and epsilon-greedy action selection. This DQN has 2 hidden layers with 128 units and 32 units, using Relu activation.

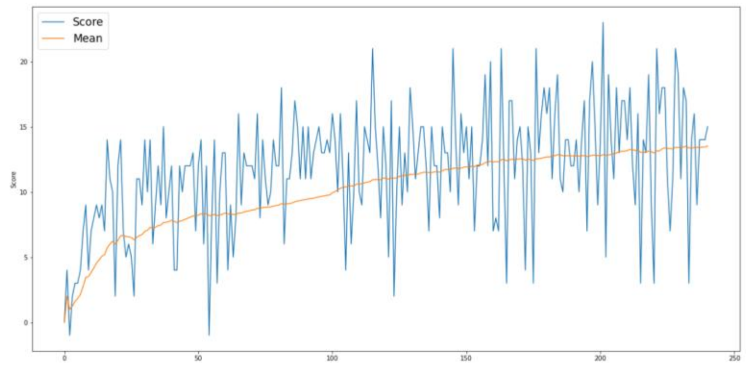
I then implemented advanced DQN techniques that were referenced earlier in the DQN lessons, with the help of associated papers and knowledge articles.

- Double Deep Q Network - Goal of this technique is to reduce the overestimation in the standard DQN architecture. I added this as an option to the dqn_agent with local and target qnetworks, and when implemented it performed considerably better than the standard DQN.
- Dueling Network - Goal of this technique is to use two estimators, one for the state value function and one for the state-dependent action advantage function. This helps generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. This dueling network has 2 hidden layers with 64 and 32 units, also using Relu.

Leveraging these techniques improved performance by over 2x compared to the 'vanilla' DQN network.

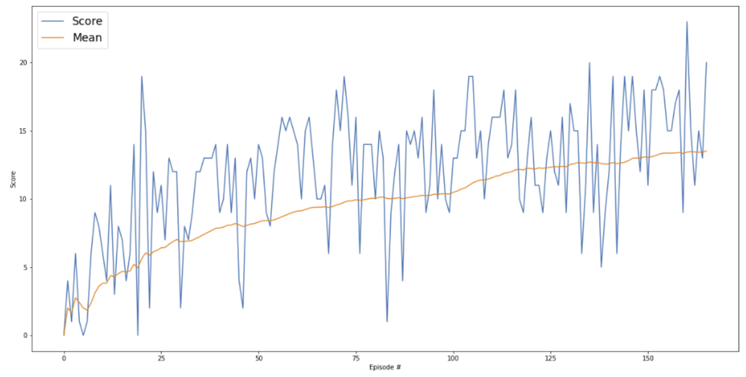
Vanilla DQN – not bad!

```
Running on: cpu
Episode 100     Average Score: 9.77     eps: 0.0270     LR: [0.00022709051475243257]
Episode 200     Average Score: 12.79    eps: 0.0100     LR: [0.00010726594079740074]
Episode 241     Average Score: 13.52    eps: 0.0100     LR: [7.886959582952152e-05]]
Environment solved in 141 episodes!     Average Score: 13.52
```



Double DQN with Dueling Networks – even better!!

```
Running on: cpu
Episode 100     Average Score: 10.36    eps: 0.0270     LR: [0.00022709051475243257]
Episode 166     Average Score: 13.53    eps: 0.0115     LR: [0.00013842434484267883]
Environment solved in 66 episodes!      Average Score: 13.53
```



3. Hyperparameters

This phase required additional research to determine how to tweak the hyperparameters optimally for my DQN. I used the ADAM optimizer and a LR scheduler and starting LR decay of 0.9999, along with the following parameters for the dqn_agent;

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-2              # for soft update of target parameters
LR = 4.8e-4             # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

Epsilon_start = 1.0
Epsilon_end = 0.01
Epsilon_decay = .995

4. Conclusion

It took a lot of experimentation, tweaking hyperparameters, and turning off/on advanced DQN techniques to find a strong solution. Additional methods I could consider in the future would be adding Prioritized Experience Replay, Noisy DQN architecture, or Distributional DQN architecture.