Navigation Project

Udacity Deep Reinforcement Learning Nano-Degree

Jeremy Vila

August 25, 2018

1 Introduction

In this project, I overview my Deep QN approach to solve the navigation project in Udacity's Deep Reinforcement (RL) Learning Nano-degree, and show the results. The goal of the Deep RL Navigation project is to train and agent navigate a square environment and collect as many yellow bananas and avoid as many blue bananas as possible. Specifically, the agent receives a reqard of $+1$ for a yellow banana, and -1 for a blue banana. The agent is considered trained after getting an average score of $+13$ over 100 consecutive episodes.

At any given time t, the agent can take one of four discrete actions:

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

Additionally, at every time t, the agent receives state information from its environment. The state space is continuous of dimension 37, that contains the agent's velocity and other ray based perception of objects in it's forward direction.

2 Learning Algorithm

This implementation employs Deep Q Learning (DQN) to train the agent to navigate the environment. A standard ϵ-greedy approach was used to also encourage exploration of the action space. However, to avoid correlation issues across adjacent time-steps, we use a technique called **experience replay** that allows to train on random batches of "experiences" from a large queue. In experience replay, we maintain a buffer of the current state, current action, current reward, and next state (S, A, R, S').

In addition, we employ **fixed Q-targets** that employs two identical architectures (but usually with unequal weights). One architecture represents the current estimate of the q function, while the other represents the TD target function. Typically, the TD target network is updated much more slowly than the q function network. In this framework the update step for a weights w is

$$\Delta w = \alpha \left(R + \gamma \max_a \hat{q}(S', A, w^-) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(S, A, w). \tag{1}$$

Here, w^- represent the set of weights for the target function, while w represents the weights of the current estimate of the q function. Meanwhile, hyperparameters α controls the step size for updating the weights, while γ is the scalar controlling discounted future reward.

For the model we selected a standard Deep Learning architecture consisting of $L = 3$ fully connected layers. The first hidden layer contains 32 output neurons that is activated with the ReLu function. The second hidden layers contains 16 neurons that is activated with the Relu function as well. The final layer is a standard fully connected layer that outputs to the 4 possible actions.

The hyperparameters, e.g., the number of layers and the number of hidden neurons, were selected using a hand-tuning procedure. The number of outputs in the hidden layers was constrained

to be less than 64 to avoid overfitting. Other hyperparameters were set as $\gamma = 0.99$ and $\alpha = 10^{-4}$. ϵ was initialized as 1 (random) and gradually reduced to 0.01 over the course of training. Batch size was set to 64, though it did not have much of an impact on results.

3 Results

The results for the training process is shown in Figure 1. The training process was completed after 720 episodes, where it averaged a score great than +13 over the previous 100 episodes. One interesting observation is that though the average score increases across episodes, so does the variance and volatility.

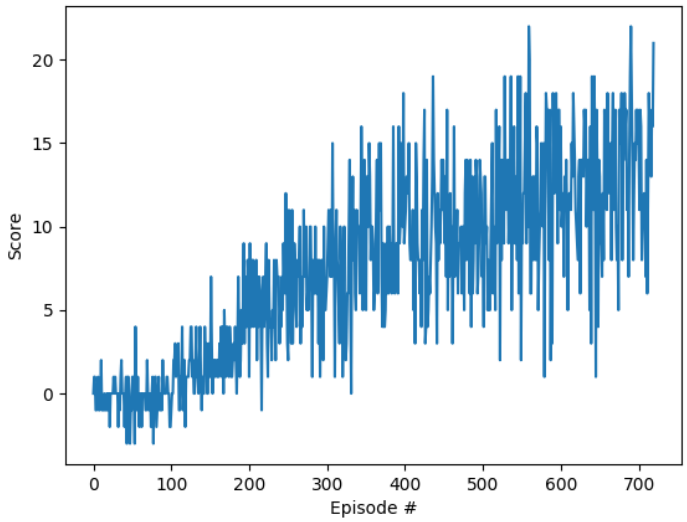

Figure 1: Episode number vs. score.

4 Future Improvements

In the proposed approach, we used a standard DQN model to train the agent. To improve stability and overall learning, we employed experience replay and fixed Q-targets. However, we could possibly see improved performance if we leverage

1. Prioritized experience replay to sample experiences that contribute to better learning with a higher probability
2. Double Q Networks that decompose the max operator in (1), by using the target and q-function networks weights separately again. This stabilizes the learning in the early iterations, where the estimates can vary wildly.
3. Dueling Networks, one of which learns the state value function, and the other the advantage values.

Finally, we can consider using alternate sensor data entirely, e.g., a camera. In this scenario, a convolutional neural network is likely to be a better choice of model for the q function, as they are shown to reliably handle image data.