

Project 1 - Navigation

Aditya Chukka

February 2, 2019

1 Introduction

For this project, we will train an agent to navigate (and collect bananas!) in a large, square world.

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

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

The task is episodic, and in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

2 Implementation

Before we go into the actual implementation details, let me give an overview of the methods I used to solve the project. Let us understand how Deep Q Network works.

2.1 Deep Q Network

At the core of our training lies, the Deep Q-Learning, an off-policy learning algorithm. Here the policy being learned is different from the policy on which we evaluate the agent.

Q-learning is a variant of Temporal Difference (TD) learning where we can learn from each step rather where as Monte Carlo methods needs an entire

episode to learn. The idea of Q-learning algorithm is to learn the optimal action-value function denoted by $\mathbf{Q}^*(\mathbf{s}, \mathbf{a})$ where s is the state and a is action. The update at each step goes this way

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (1)$$

where r_t is the reward at the current time-step t , γ is the discount factor, under policy $\pi = P(a_t|s_t)$ after observing state (s_{t+1}) by performing action a_t on state s_t .

The above state works fine when the state and action space is limited. It gets difficult when the state space and action space grow larger. Things go out of hand when the actions are continuous. Hence we use a *Function Approximator*. This is where we introduce a parameter θ to find the approximate value of $Q(s, a)$, $\hat{Q}(s, a)$. To make things interesting we turn this into a supervised problem where $\hat{Q}(s, a)$ is the expected value and $r_t + \gamma \max_a Q(s_{t+1}, a)$ is the target. Using mean square we reduce the difference between the expected and target values, i.e loss and update the parameter θ using optimization techniques like gradient descent.

Here we use a three layered feed-forward neural network with 2 hidden layers of size 64 units, a rectified linear unit. Adam optimizer has been used to find the optimal weights (or parameters). The update step for a simple Q network looks something like this

$$\theta \leftarrow \theta + \alpha(r_t + Q^{hat}(s', a', \theta) - Q^{hat}(s, a, \theta))\delta Q^{hat}(s, a, \theta) \quad (2)$$

These updates are highly unstable both due to mathematical reasons and initial estimations. Two techniques have been proposed to stabilize training.

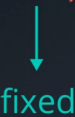
2.1.1 Experience Replay

During training the sequence of observations might be highly correlated. This might make the network sway to a specific behavior and doesn't learn all scenarios. We maintain a buffer of tuples $(s_t, a_t, r_{t+1}, s_{t+1})$ known as **Replay Buffer** of a fixed length and randomly sample tuples to update the network parameters. This way our network restricts parameters from diverging catastrophically in-turn stabilizing the action space. We also get an additional benefit of reusing experiences instead of simulating every time.

2.1.2 Fixed Q-Targets

As we can see from the above equation, we are training and updating on the same parameter θ . This might never lead to convergence since the targets are always moving. How can we fix this? The plan is to use two separate networks one with fixed targets and the other learns the q-value. We freeze the target networks parameters and learn weights for the others. After every C steps we update the target network parameters to coincide with the online network. The fixed target update equation as shown in the lecture.

$$\Delta \mathbf{w} = \alpha \left(R + \gamma \max_a \hat{q}(S', a, \mathbf{w}^-) - \hat{q}(S, A, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$



fixed

Figure 1: Fixed Target Update Equation

You can learn more about Q-learning in this paper.

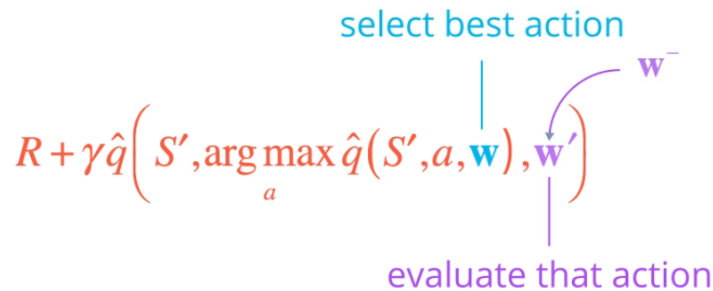
In addition to normal DQN, the course presents two variants of it.

2.2 Double DQN

DQN's tend to overestimate action value function due to *max* operator and that is expected since the network doesn't have enough information in the earlier stages. Double DQN overcomes this by finding the best action and the computes Q-values for that action. It achieves this by computing best actions using local network and evaluates that action using the target network. The rationale is that if one network makes a mistake in computing the best action, the other network might rectify since it will have lesser Q-value and treat it as sub-optimal. You can find more details about this approach in this paper.

The update equation as shown in the lecture.

$$R + \gamma \hat{q} \left(S', \arg \max_a \hat{q}(S', a, \mathbf{w}), \mathbf{w}' \right)$$



select best action

 evaluate that action

Figure 2: Double DQN Update Equation

2.3 Dueling Network

DQN have single output where the number of output nodes is equal to the number of actions. This might lead into a case where we calculate actions for

states that are unstable or not suitable. Or we might encounter scenarios where the action is fixed for a certain state.

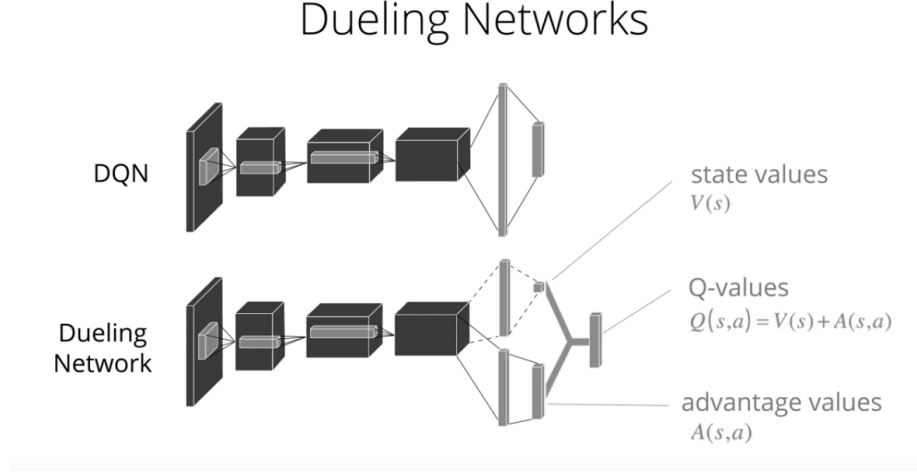


Figure 3: Dueling Network Architecture

Here comes, Dueling network that has two output streams with a shared feature extractor, one stream computes the value function $V(s)$ and other calculates the advantage function $A(s, a)$, you can treat this as the quality of action (a) at state (s). These two are aggregated by the following equation.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a^t} A(s, a^t; \theta, \alpha) \quad (3)$$

This approach addresses the issue of identifiability, i.e given V and A we can recover Q values uniquely. You can read more about Dueling Network architecture in this research paper.

3 Results

Before running make sure you've the necessary dependencies, i.e Unity ML-Agents and NumPy. You might want to install PyTorch depending your device architecture.

I have struggled a lot getting things right for GPU, you might want to cross-check versions of all the libraries and try to get the configuration correct. One a side note, this version of project can run without a GPU.

Network	Notebook	Model Weights
Deep Q Network (DQN)	Navigation_DQN.ipynb	dqn.pth
Double DQN	Navigation_Double_DQN.ipynb	double_dqn.pth
Dueling DQN	Navigation_Dueling_DQN.ipynb	dueling_dqn.pth

Table 1: Network and Source Code

3.1 DQN

The implementation of DQN lies in *Navigation_DQN.ipynb* and model weights are placed in *Models/dqn.pth*.

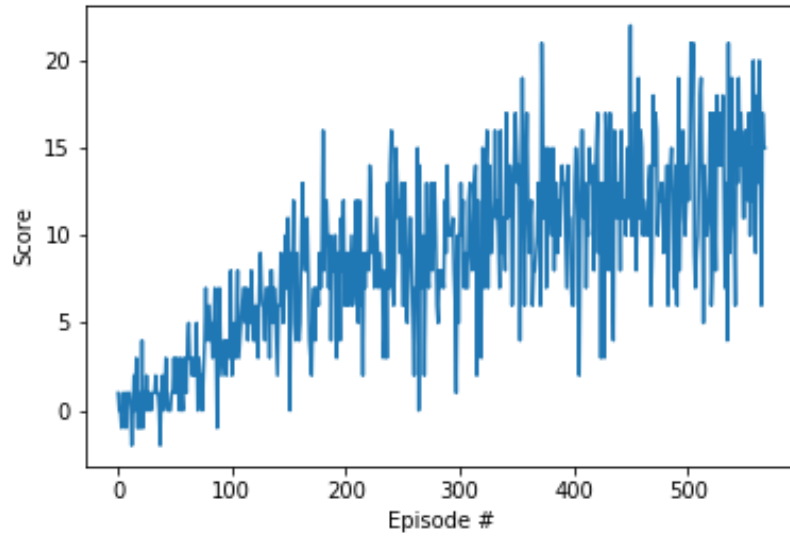


Figure 4: Scores vs Episodes for DQN

3.2 Double DQN

The implementation of Double DQN lies in *Navigation_Double_DQN.ipynb* and model weights are placed in *Models/double_dqn.pth*.

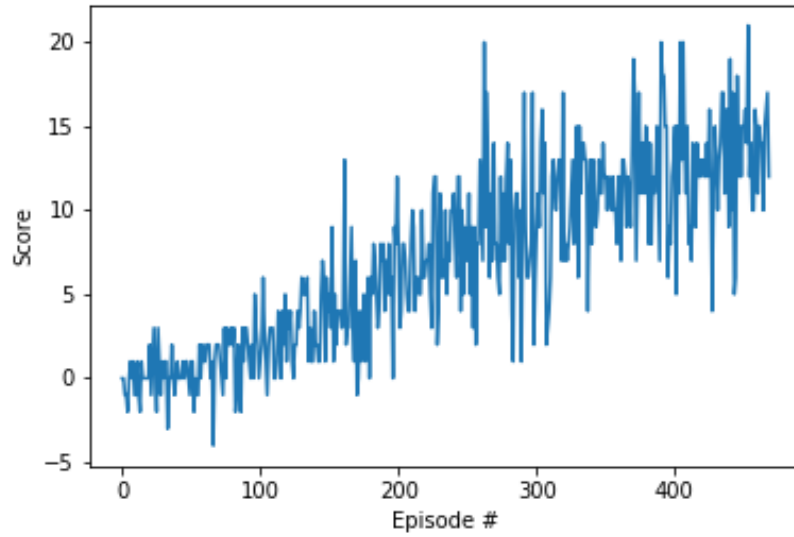


Figure 5: Scores vs Episodes for Double DQN

3.3 Dueling DQN

The implementation of Dueling DQN lies in *Navigation_Dueling_DQN.ipynb* and model weights are placed in *Models/dueling-dqn.pth*.

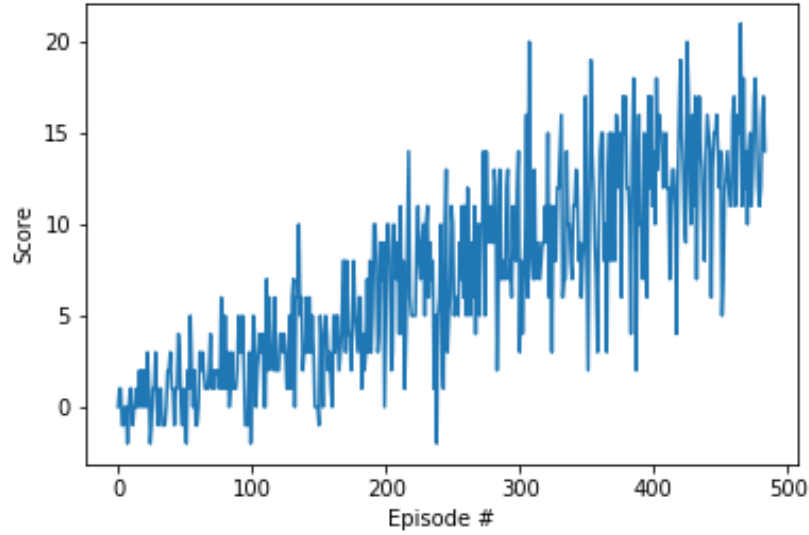


Figure 6: Scores vs Episodes for Dueling DQN

All the above approaches did converge quickly, however Double DQN solved the task in **369** episodes. From the plots you can see that agent learns slowly at start, and the scores start to fluctuate in middle (i.e. change drastically) and converge towards the end. I feel Dueling DQN is smoother compared to the other two and can be applied in real world scenarios.