Xiaofei Zheng

Udacity Deep Reinforcement Learning Project 1

Navigation – Banana Collector Robot

Table of Contents

[Introduction 2](#_Toc14010726)

[Environment 2](#_Toc14010727)

[Methodology 2](#_Toc14010728)

[1. Deep Q-Networks (DQN) [1] 2](#_Toc14010729)

[2. Double Deep Q-Networks (DDQN)[5] 4](#_Toc14010730)

[3. PRE-Experience Replay 4](#_Toc14010731)

[Experiments 5](#_Toc14010732)

[Future Work 8](#_Toc14010733)

[References 9](#_Toc14010734)

Navigation – Banana Collector Robot

## Introduction

This project 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 the 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 is considered solved when the Agent reaches an average score of +13 over 100 consecutive episodes.

## Environment

Unity Machine Learning Agents (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. For game developers, these trained agents can be used for multiple purposes, including controlling [NPC](https://en.wikipedia.org/wiki/Non-player_character) behavior (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release.

In this project, we will use [Unity's rich environments](https://github.com/Unity-Technologies/ml-agents) to design, train, and evaluate our own deep reinforcement learning algorithms.

To focus on the model training itself, we use the banana environment directly from one of the links below.

* Linux: [click here](https://s3-us-west-1.amazonaws.com/udacity-drlnd/P1/Banana/Banana_Linux.zip)
* Windows (64-bit): [click here](https://s3-us-west-1.amazonaws.com/udacity-drlnd/P1/Banana/Banana_Windows_x86_64.zip)

## Methodology

### Deep Q-Networks (DQN) [1]

Instead of using [Monte Carlo Method or Temporal - Difference Methods](https://www.linkedin.com/pulse/brief-introduction-reinforcement-learning-xiaofei-zheng/) [2] to estimate the action-value function, we will go further: use non-linear approximation to estimate value function based on observations from the environment by neural network or deep learning.

The basic idea is that using the environment states as input (it can be the states of the agent or it can be the pure pixels as input), predict a Q value for every possible action in a single forward pass, and choose the action which the neural network thinks will maximize the reward.

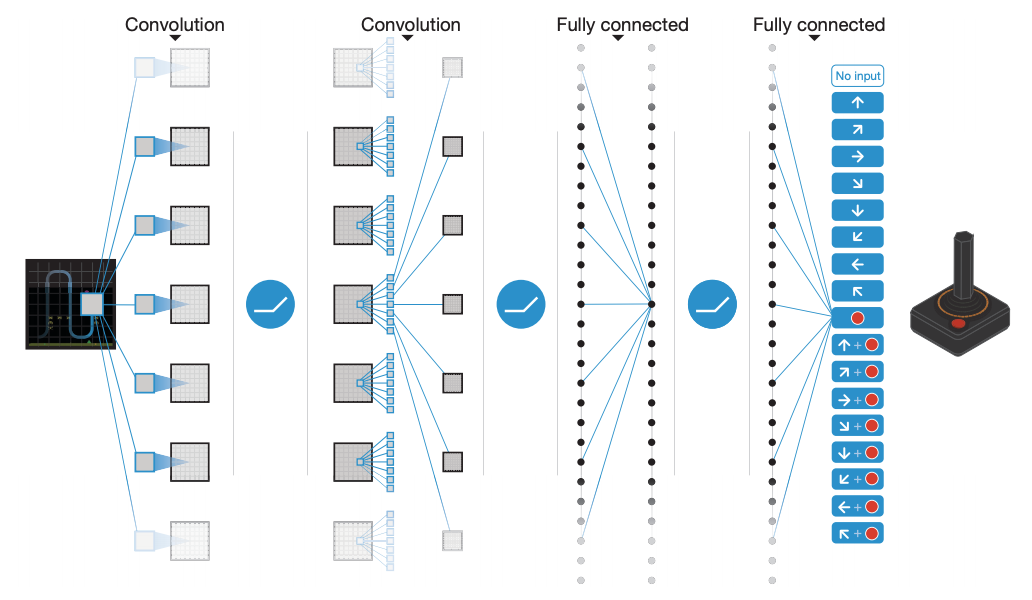


Figure 1: Schematic illustration of the convolutional neural network

Training such a neural network requires a lot of data. Unfortunately, due to the large amount of combination of states and actions, the result may fluctuate and not converge to the optimal Q values. In other words, reinforcement learning is [notoriously unstable](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.3097&rep=rep1&type=pdf)[3] when neural networks are used to represent the action values. In this project, we will address these instabilities in the Deep Q-Learning Network (DQN) by using two key features:

* Experience Replay
* Fixed Q-Targets

#### What is Experience Replay?

When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation.

By instead keeping track of a replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically. The replay buffer contains a collection of experience tuples (S, A, R, S'). The tuples are gradually added to the buffer as we are interacting with the environment.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as **experience replay**. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

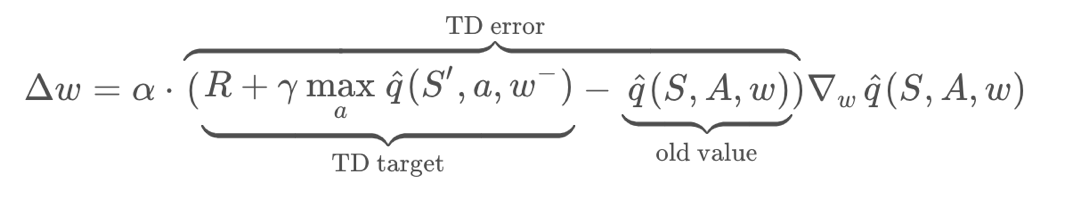
#### Fixed Q-Targets

In Q-Learning, we update a guess with a guess by using the formula below:

 (1.1)

and this can potentially lead to harmful correlations.

To avoid this, we can update the parameters (weights) in the network to better approximate the action value corresponding to state S and action A with the following update rule:

(1.2)

For the whole project, we will fix Q-Targets: Deep Q-Learning algorithm uses two separate networks with identical architectures. The target Q-network’s weights are updated less often than the primary Q-Network.

### Double Deep Q-Networks (DDQN)[5]

[From experiments, Deep Q-Learning tends to overestimate action values](https://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf) [3]. Recall in DQN (1.1), we always take the maximize value of  , but it is not accurate especially at the early stage of training, in which way DQN tends to overestimate action values.

Double Q-Learning (DDQN) has been shown to work well in practice to help with this. Let’s rewrite (1.1) to be:

(2.1)

### PRE-Experience Replay

Recall that it is a waste of resource if we throw away an experience tuple <S\_t, A\_t, R\_t+1, S\_t+1> after using it. The idea of experience replay is to use replay buffer to store all tuples and then train the model using experience sampled from the buffer. This helps us make better use of experience. Sample can also break the order of the highly correlated experiences.

However, when treating all sample the same, we are not using the fact that we can learn more from some transitions than from others. **Prioritized Experience Replay** (PER) [4] is one strategy that tries to leverage this fact by changing the sampling distribution.

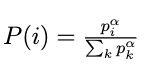
The main idea is that we prefer transitions that does not fit well to our current estimate of the Q function, because these are the transitions that we can learn most from. One criterion to measure the importance of an experience is the temporal differential error (TD error), which is defined as

(2.2)

We will store this error in the agent’s memory along with every sample and update it with each learning step. The distribution of the sampling then is defined to be

(2.3)

The extra term e in (2.3) is to avoid that the probability becomes 0. Some experiences may be over selected, and it leads to overfit on a subset. To avoid this, we define the probability to be its power where power ranges from 0 to 1. Thus, the probability of the ith tuple is selected becomes

(2.4)

When a = 0, it becomes the experience replay since the distribution is uniform. When a = 1, its distribution is from td-error. In all, parameter determines how much prioritization is used. Correspondingly, the updated rule will no longer use the uniform weights and we modify the update rule as below:

(2.5)

where b is a paramete determining the importance of the sampling weights.

The algorithm is simple - during each learning step we will get a batch of samples with this probability distribution and train our network on it. We only need an effective way of storing these priorities and sampling from them.

Sumtree is a good data structure to store the experience and priorities. As for how do we store the experience and effectively sample from it, those posts give a good explanation: [Let’s make a DQN: Double Learning and Prioritized Experience Replay](https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/#fn:4) and [Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed…](https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/)

## Experiments

We test DQN + Replay Experience, DDQN + Replay Experience with the following parameters:

|  |
| --- |
| parameter list: |
| n\_episodes=2000 |
| max\_t=1000 |
| eps\_start=1.0 |
| eps\_end=0.01 |
| eps\_decay=0.999 |
| fc1\_units=64 |
| fc2\_units=64 |
| BUFFER\_SIZE = 1e5 |
| BATCH\_SIZE = 64 |
| beta\_incremement = 0.01 |
| priority\_epsilon = 0.001 |
| max\_error = 1 |
| GAMMA = 0.99 |
| TAU = 1e-3 |
| Learing Rate = 1e-4 |
| UPDATE\_EVERY = 4 |
| ALPHA = 0.5 |
| BETA = 0.4 |

The results are shown as below:

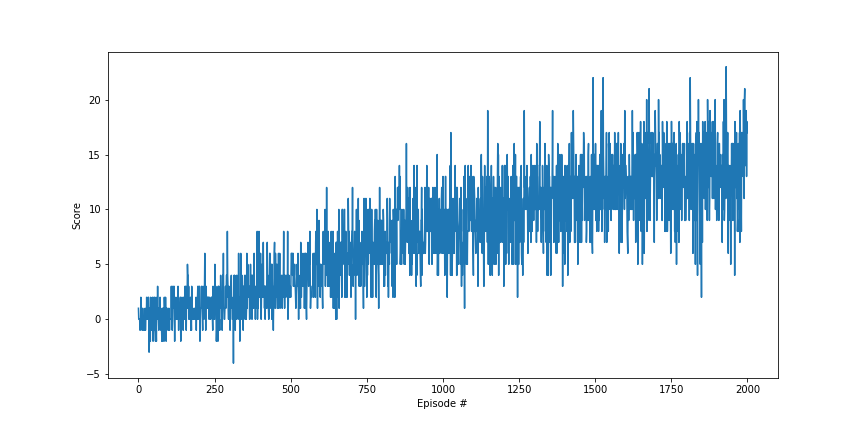


Figure2: DQN + ReplayBuffer

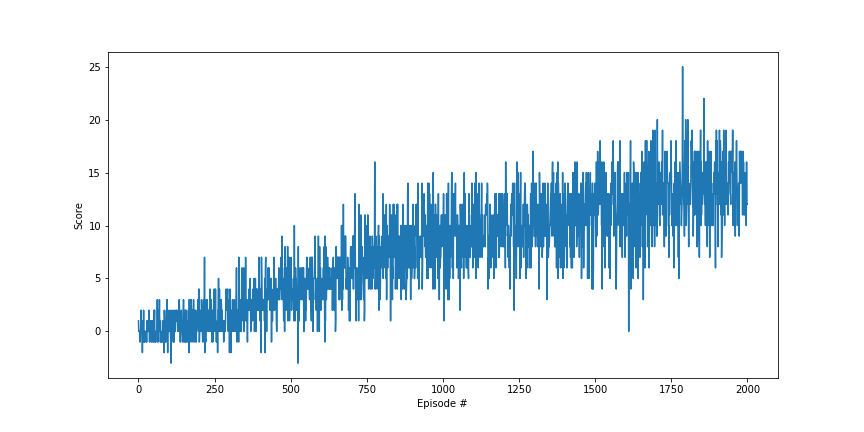


Figure3: DDQN + ReplayBuffer

With the same parameters, DQN + PRE\_Replay Experience, and DDQN + PRE\_Replay Experience don’t have a better result. Hence, we adjust the learning rate to be 5e-4 and have the following results:

Figure4: DQN + Prioritized Experience Replay

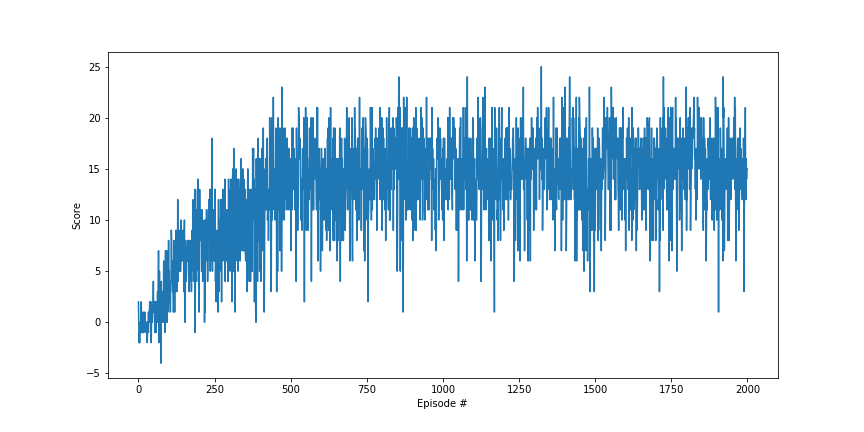


Figure5: DDQN + Prioritized Experience Replay

BY DDQN + Prioritized Experience Replay, it reaches the goal of average score above 13 as early as 400 episodes. For Experience Replay, it needs around 1800 episodes.

#### Comparison of the four experiments

From the experience results, we can see that by applying DDQN, there is a slight improvement of the number of episodes one needs to achieving the average score above 13.0. When implement prioritized experience replay, there is a huge improvement in term of the number of episodes from 1800 reduced to 400.

## Future Work

Due to time limit, I didn’t tune the parameter a lot considering that the current parameters are good enough for the agent to achieve the goal. Later, I will tune the parameters to see which ones have more influence on the performance.

In this project, we test Double DQN and Prioritized experience replay. Dueling DQN [6] would also be good to explore further. We cite the introduction of [6] to give an idea of the Dueling DQN method. “The proposed network architecture, dueling architecture, explicitly separates the representation of state values and (state-dependent) action advantages. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q as shown below. This dueling network should be understood as a single Q network with two streams that replaces the popular single-stream Q network in existing algorithms such as Deep Q-Networks. The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision. Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state.”

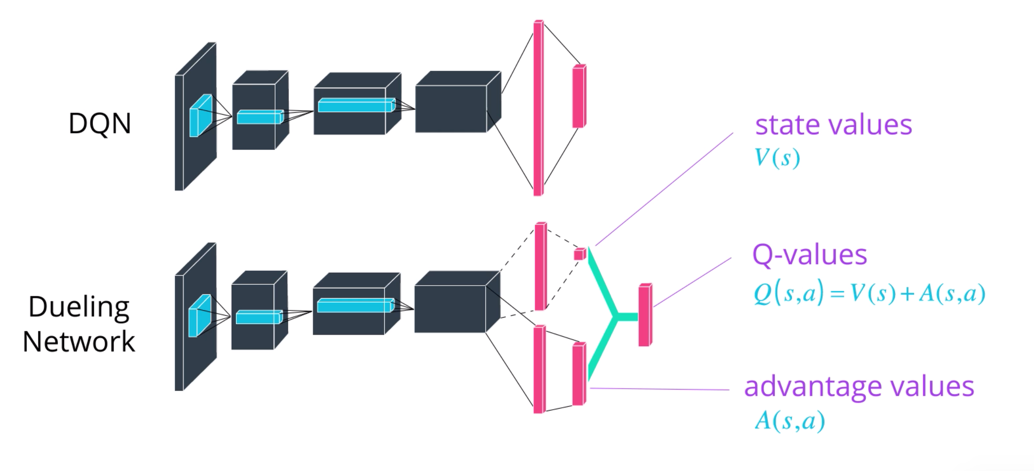


Figure6: Dueling DQN structure

Besides Double DQN (DDQN), Prioritized experience replay and Dueling DQN, there are other extensions of DQN, including

* Learning from [multi-step bootstrap targets](https://arxiv.org/abs/1602.01783)
* [Distributional DQN](https://arxiv.org/abs/1707.06887)
* [Noisy DQN](https://arxiv.org/abs/1706.10295)

Each of the six extensions address a **different** issue with the original DQN algorithm.

[Rainbow](https://arxiv.org/abs/1710.02298) algorithm is an algorithm incorporate the six modifications, which outperforms each of the individual modifications. We will have an exploration of it in the future to the banana collector robot.

Visual/Pixel based training will be explored later as well.

## References

[1] Mnih, Volodymyr, et al. "[Human-level control through deep reinforcement learning](https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf)." *Nature* 518.7540 (2015): 529.

[2] Xiaofei, Zheng. [“Brief Introduction to Reinforcement Learning.”](https://www.linkedin.com/pulse/brief-introduction-reinforcement-learning-xiaofei-zheng/)

[3] Thrun, Sebastian, and Anton Schwartz. "[Issues in using function approximation for reinforcement learning](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.3097&rep=rep1&type=pdf)." *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*. 1993.

[4] Schaul, Tom, et al. "[Prioritized experience replay](https://arxiv.org/abs/1511.05952)." *arXiv preprint arXiv:1511.05952* (2015).

[5] Van Hasselt, Hado, Arthur Guez, and David Silver. "[Deep reinforcement learning with double q-learning](https://arxiv.org/abs/1509.06461)." Thirtieth AAAI conference on artificial intelligence. 2016.

[6] Wang, Ziyu, et al. "[Dueling network architectures for deep reinforcement learning](https://arxiv.org/abs/1511.06581)." *arXiv preprint arXiv:1511.06581*(2015).