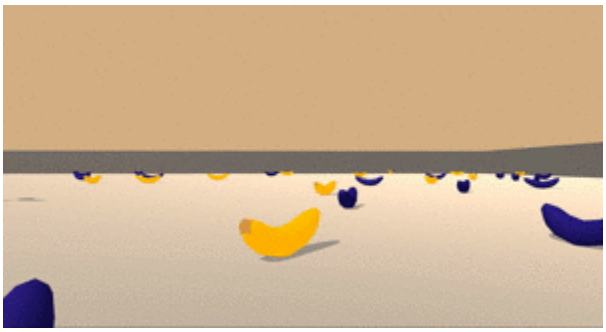
Project 1: Navigation

Introduction

This project is designed to train an agent to navigate (and collect bananas!) in a large, square world. Two different color bananas existed in this world. We will adjust the agent's action to collect more yellow bananas and get a reward of +1 for each yellow banana. If the agent collects the blue bananas, we will get reward of -1.



In the big world, agent's state space has 37 dimensions, such as the agent's velocity, ray-based perception of objects around agent's forward direction. There are four discrete actions in total, including:

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

We will train the agent to collect more yellow bananas and avoid the blue bananas. This needs the agent to make the best action under certain state. We use the Deep Q Network (QDN) as the training model. In the model, the input will be the 37 dimensions state information, the output will be the four actions. Model architecture designed as:

1. Fully connected layers + Relu activation function
2. Fully connected layers + Relu activation function
3. Fully connected layers

In this report, we will train models based on this model architecture and test different parameter settings and compare the model performance. Model performance is the mean reward value of every 100 episodes. Each episode has been limited to 1000 steps.

Exploration of the DQN

1. Comparing the DQN size

As mentioned above, we use three fully connected layers and the number of nodes per layer needs to be determined. We compared the following number of nodes:

	1st fully connected layer	2nd fully connected layer
Model1	128	64
Model2	64	64
Model3	64	32

Usually, there is no clear rule about how many nodes shall we chose, but we usually chose the 2*n (n is an integer). Here we compared different size of neural networks. For each of them, we saved out their best model's weights according to the score values.

Next, we will compare their scores by number of episodes.

Figure 1: Score by episodes in different size neural networks

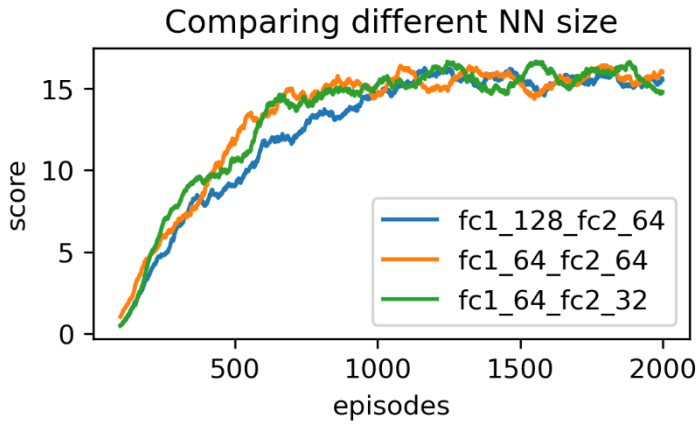


Figure 1 shows how the scores change by episodes. We can see that all modes reach to relatively stable around 1000 episodes. The scores show some level of variance along the episodes. There is no clear difference among these three models. If we further compare the time used by each model and the highest score

	Time used	Highest average score per 100 episodes
Model1	2325s	16.3
Model2	2416s	16.44
Model3	2416s	16.67

Surprisingly, the model 3 reaches to the highest scores though no big difference. In short, we will suggest the model 3 architecture which is good enough in our project.

2. Comparing learning rate

Except of the neural network size, learning rate is another key parameter. If the learning rate is too high, we will see the scores oscillating; if the learning rate is too low, the learning process will be too slow to converge. Therefore, we have tested three different commonly used learning rate values: 0.001, 0.0005, and 0.0001.

Figure 2: Score by episodes with different learning rate

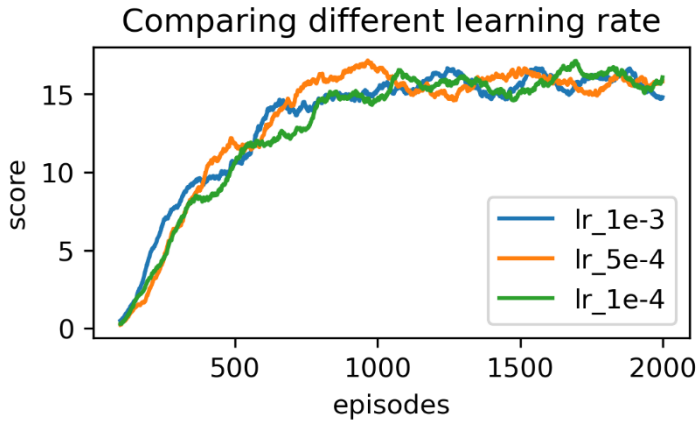


Figure 2 shows how the scores changes by episodes with different learning rate value. We can clearly see that when learning rate is 0.001, the model learns the fast at the first 500 episodes. However, learning rate equals to 0.0004 achieve the first peak in terms of scores. If we explore the time used and the highest score model ever achieved, we obtained following table.

	Learning rate	Time used	Highest average score per 100 episodes
Model3	0.001	2416s	16.67
Model4	0.0005	2426s	17.16
Model5	0.0001	2487s	17.13

We can see model with smaller learning rate took a little bit longer time to finish but the obtained to higher score. In consideration of both time and model performance, I think learning rate 0.0005 is a good choice.

3. Comparing updating frequency

The DQN model we used here uses one local network and one target network. The local network will keep collecting more bananas and all steps will be saved to our Buffer. At the same time, target network will update its weight at a certain frequency. This updating frequency may also affect how good the model performs and how fast it converges.

Here, we compared the updating frequency as 4, 8, and 16 steps.

Figure 3: Score by episodes with different updating frequency

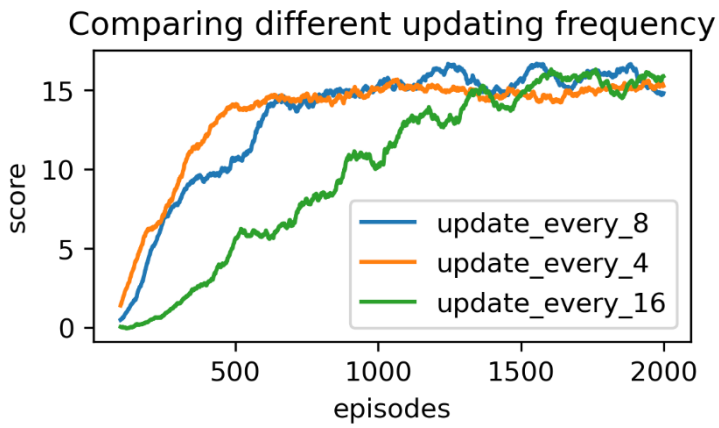


Figure 3 shows how the score changes by episode with different updating frequency. It is clearly that when updating the target network every 16 steps, the learning speed is significantly lower than the other two. Updating every 4 steps looks like the best in the first 1000 episodes. However, if we further compare the best scores ever in table below, we found that model 6 has the lowest score.

	Update frequency	Time used	Highest average score per 100 episodes
Model3	8	2416s	16.67
Model6	4	2990s	15.66
Model7	16	2001s	16.31

Updating the target network too infrequent can reduce the running time and the learning rate is lower. We may need more episode to compensate the low learning rate when updating infrequently.

Future work and direction

Except of the parameters tuning of neural networks, we can have more improvements in terms of the network structures. For example, double DQN, prioritized experience replay, and dueling DQN.

Double DQN: solve the issue of overestimation of action values in some conditions. When updating the weights, instead of using the same weights for the both the action selection and evaluation, we use the weights from target network as the action evaluation weight. In this way, we can avoid the inflation of action values by always selecting the maximum values.

Prioritized experience replay: in this approach, models will prioritize the saved experiences: the more important, the more frequent it will be sampled.

Dueling DQN: in the neural networks, dueling DQN will generate two sets of layers before the output layer, one for state values, one for state-dependent action advantage values.

Then the output layer will be the Q-values which is the sum of state values and advantage values. This approach can generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm.