Project: Navigation

1 Overview

In this report, I will briefly summarize the learning algorithms I used for the navigation project as well as their performance. I will also comment on some ideas for future improvement.

2 Environment and Task

The goal of this project is to build an agent which moves inside a two-dimensional plane with yellow and blue bananas dropped, and collect efficiently the yellow ones while avoiding the blue ones. A state is 37-dimensional and the number of possible actions is 4 (move forward, move backward, turn left and turn right). In this report, we denote state, action and reward at time step t by s_t, a_t and r_t, respectively. The agent receives a reward +1/-1 if it collects a yellow/blue banana. The environment is regarded as being solved once the average score over the last 100 episodes reaches 13 or higher.

3 Learning Algorithm

For a given policy π, the action-value function for a state s and an action a at time step t is defined as an expectation value of the accumulated rewards under the current policy after this time step, i. e. $Q_\pi(s,a) = \mathbb{E}[r_t + \gamma r_{t+1} + \cdots | s_t = s, a_t = a, \pi]$ where $\gamma(= 0.999)$ is the discount factor. For a given state, the next action is taken by using the ϵ-greedy algorithm based on this action-value function. The parameter ϵ for the ϵ-greedy algorithm is initially set to 1.0 and decays as $\epsilon \to 0.99 \times \epsilon$ when an episode ends until it reaches 0.01.

In this project, the action-value function is build as a neural network, $Q(s, \cdot; \theta)$, where θ is a set of weights in the neural network. I will then train the neural network by using some reinforcement learning algorithms.

3.1 Architecture of Neural Network

Before explaining the reinforcement learning algorithms, here I introduce the (local) neural network I will train for this project. The neural network is composed of two fully-connected layers with 128 neurons followed by a fully-connected output layer with 4 neurons. For the activation function, the rectified linear unit is used. The architecture is summarized in Figure 1. We trained this neural network every $t_{learn} = 5$ time steps by using some reinforcement learning algorithms explained below.

3.1.1 Deep Q-learning

In the deep Q-learning [1], another neural network with the same architecture (called the target network, weights are denoted by θ^{target}) is introduced. The weights of the target network is update at the end of each learning process by using the soft-update i.e. $\theta^{target} \leftarrow \tau\theta + (1-\tau)\theta^{target}$ where $\tau(= 0.001)$ is a hyper-parameter. In the learning process, for a given tuple (s_t, a_t, r_t, s_{t+1}) sampled from the replay buffer (which is explained below), the target y_t defined by

$$y_t = \begin{cases} r_t & \text{if episode ends at time step } t+1\,, \\ r_t + \gamma Q(s_{t+1}, \mathrm{argmax}_a Q(s_{t+1}, a; \theta^{target}); \theta^{target}) & \text{otherwise}\,, \end{cases}$$

is computed, and, by using this, the loss is defined as the square of the TD error, $(y_t - Q(s_t, a_t; \theta))^2$ averaged over the samples. The local neural network is trained to minimize this loss. For the optimizer, the Adam optimizer with learning rate 0.001 is used.

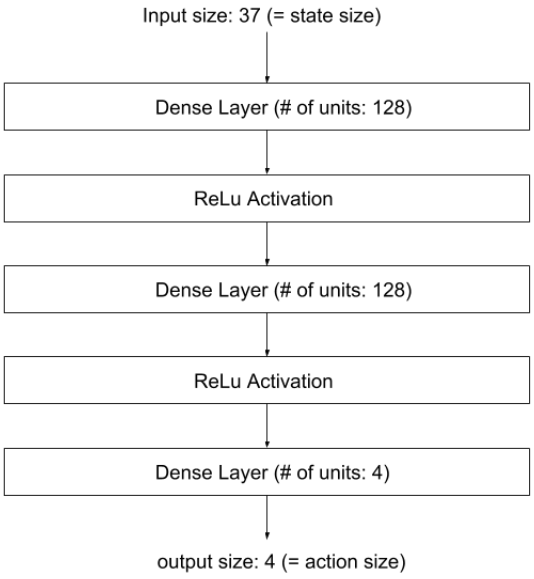

Figure 1: The architecture of the neural network.

3.1.2 Double Deep Q-learning

In the standard deep Q-learning, the same weights ($= \theta^{target}$) is used for selecting and evaluating an action. In the double deep Q-learning [2] (here I follow the implementation in [3]), selection of an action is done by the local network, while the evaluation of the action is by the target network:

$$y_t = \begin{cases} r_t & \text{if episode ends at time step } t+1\,, \\ r_t + \gamma Q(s_{t+1}, \mathrm{argmax}_a Q(s_{t+1}, a; \theta); \theta^{target}) & \text{otherwise}\,. \end{cases}$$

The rest is the same as the deep Q-learning algorithm explained above.

3.2 Prioritized Replay Buffer

For training the model by using Q-learning/double Q-learning algorithm, experience tuples made of state, action, reward and next action (and if the episode is done or not) are stored in the replay buffer. The buffer size is set to $N = 10^5$, and, in the each learning process, 128 experience tuples are samples from the standard replay buffer randomly.

To train more efficiently, in this project, I also implemented the prioritized replay buffer [3] which samples the experience tuple by following a probability distribution defined based on priorities. More concretely, the priority p_i for an experience tuple i is defined based on the TD error δ_i as $p_i = |\delta_i| + \epsilon_{priority}$ where a small non-zero constant $\epsilon_{priority} (= 1.0 \times 10^{-8})$ is added so that the algorithm works even when the TD error is zero. Then the experience tuple i is chosen with probability

$$P(i) = \frac{(p_i)^\alpha}{\sum_{j \in \text{replay buffer}} (p_j)^\alpha}\,,$$

where the exponent $\alpha (= 0.9)$ is a hyper-parameter.

As mentioned in [3], to correct the bias introduced due to the prioritized replay, the importance-sampling weights

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{\beta}$$

(β is a hyper-parameter) must be introduced. The hyper-parameter β is updated such that $\beta = 1.0 + (0.4 - 1.0)/n$ at the n-th learning process. For the prioritized replay buffer, this importance-sampling weights are multiplied when the gradient of the loss with respect to the network weights is evaluated. I note that the choice of the hyper-parameters for the prioritized replay buffer in this project is motivated by the original paper [3] with some tuning for the current project.

4 Performance

To see the performance, here I provide the plot for the time evolution of the average score. In Figure 2, the average score over the last 100 episodes (which is used for how well the agent has trained in this project) is plotted. I note that I trained the agents for 1000 episodes.

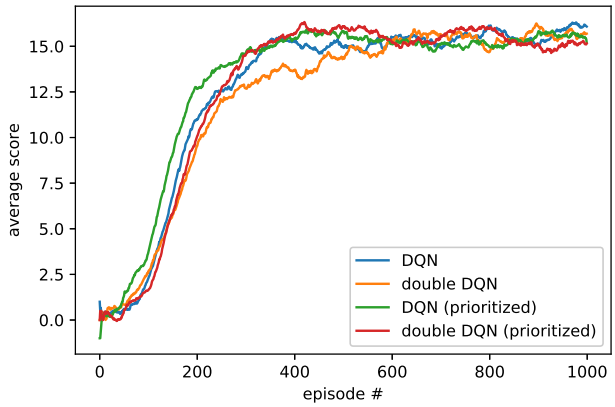

Figure 2: Evolution of the average score over the last 100 episodes.

The number of episodes required for solving the environment is summarized as follows for each model:

	DQN	double DQN	DQN (prioritized buffer)	double DQN (prioritized buffer)
# of episodes	278	318	214	262

This result shows that with the prioritized replay buffer, the agent solves the environment more quickly than with the standard replay buffer. For the highest average score, there is no crucial difference in the four cases I have analyzed but the double DQN with the prioritized buffer achieved the highest average score, 16.33.

5 Summary and Comments

In this project, I have implement several deep reinforcement learning algorithms with additional improvements to train an agent for solving the navigation problem. The agents trained with the four different algorithms have been all trained efficiently, solving the environment in less than 320 episodes.

As shown in the above figure, I could not see remarkable improvement of the performance by using double deep Q-learning algorithm. In this project, I used the same set of hyper-parameters for both the double deep Q-learning and standard deep Q-learning. It will be interesting to use different set of hyper-parameters for the double deep Q-learning case. Also in this project, I considered a simple neural network with two hidden layers. It is also worthwhile to see how the performance can get improved by considering more layers (and more units in each layers). Another important point is that, training of the deep reinforcement learning agent seems to be highly dependent on the initial value of weights of the neural network (and choice of the random seeds). It is thus important to systematically analyze how the initialization of the weights can affect the training speed and the performance of the trained model.

References

1. "Human-level control through deep reinforcement learning," V. Mnih et. al., Nature, **518** (2015) 529

2. "Deep Reinforcement Learning with Double Q-learning," H. van Hasselt et. al., arXiv:1509.06461[cs.LG]

3. "Prioritized Experience Replay," T Schaul et. al., arXiv:1511.05952[cs.LG]