Udacity Deep Reinforcement Learning
Project Navigation

Rohil Pal (rohilpal9763@gmail.com)

Abstract

In this project, we have to train an agent to navigate (and collect bananas!) in a large, square world. This environment is provided by Unity Machine Learning agents (ML-agents). We are free to use any value-based methods that were taught in the course.

1 Environment

The **state space** has 37 dimensions each of which is a continuous variable. It includes the agent's velocity, along with ray-based perception of objects around the agent's forward direction.
The **action space** contains the following 4 legal actions:

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

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

2 Method

Two variants of DQNs are used to solve this problem.

1. **Vanilla DQN with Experience Replay and Fixed Q-targets:**
   Vanilla DQN employs a simple Deep neural network to approximate the Q-value function. For each state, it estimates the Q-values for each action and performs gradient descent on the MSE loss between the expected Q-value and the current Q-value. To achieve this efficiently, there are two things employed by a Vanilla DQN:

(a) **Experience replay:** In a Vanilla DQN, we feed the experiences tuples (s, a, r, s') sequentially. So there exists a possible correlation between two consecutive tuples which can be learnt by the network. To avoid this, we store the tuples in a *replay buffer* or a *experience replay buffer*. Then, we randomly sample a batch of tuples and use them to calculate the expected value function.

(b) **Fixed Q-targets:** We use two networks: **online/local** and **target**. The online network is updated at every gradient step while the target network is updated with the current weights of the online network at regular intervals.

2. **Double DQN:** Double DQNs help in handling the problem of the over-estimation of Q-values. In a Vanilla DQN, we estimate the TD target by the following formula:

$$Q(s, a) = r(s, a) + \gamma max_a Q(s', a) \tag{1}$$

Now, the problem with the above approach is how are we sure that the best action for the next state is the action with the highest Q-value?

We know that the accuracy of Q values depends on what action we tried and what neighboring states we explored.

As a consequence, at the beginning of training we don't have enough information about the best action to take. Therefore, taking the maximum Q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

The solution to the above problem is **Double DQN**. We use 2 networks to decouple the action selection from the target Q value generation. We:

- use our DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- use our target network to calculate the target Q value of taking that action at the next state. The formula for the TD target is given below.

$$Q(s, a) = r(s, a) + \gamma Q(s', argmax_a Q(s', a)) \tag{2}$$

3 DQN Architecture

It has 3 fully connected layers each 150 neurons. Each of them is followed by a $ReLU$ activation function. To avoid overfitting, *dropout* with a probability of 0.5 is used after every fully connected layer except the last layer.

The network accepts a tensor of dimension 37 which is the dimension of each state and outputs a tensor of 4 dimension which is the number of actions an agent can perform at each state.

4 Hyperparameters

4.1 Vanilla DQN

- Replay buffer size : e^5
- Batch Size : 64
- Discount factor ($gamma$) : 0.99
- Soft Update parameter (TAU) : e^{-3}
- Learning rate ($alpha$) : e^{-3}
- Frequency of network update : 50

4.2 Double DQN

- Replay buffer size : e^5
- Batch Size : 32
- Discount factor ($gamma$) : 0.99
- Soft Update parameter (TAU) : e^{-3}
- Learning rate ($alpha$) : e^{-4}
- Frequency of network update : 4
- Frequency of target network update : 100

5 Results

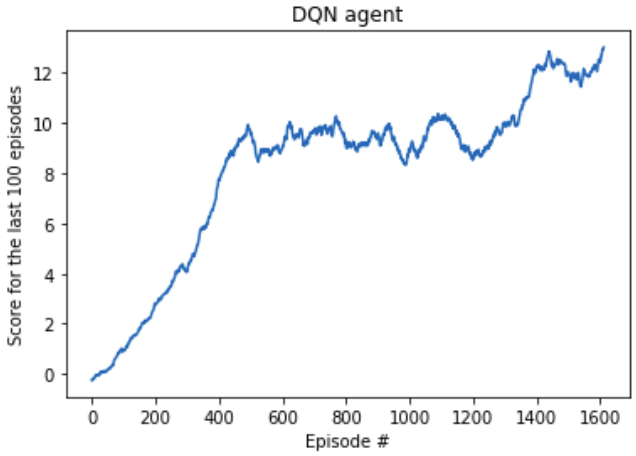

Figure 1: Vanilla DQN with replay buffer and fixed Q-targets

Figure 2: Double DQN

Vanilla DQN was able to solve the environment in approximately 1600 episodes. As is evident, after 500 episodes, the score did not improve much (it kept oscillating between 8 and 10). After 1200 episodes, the score finally starts improving.

On the other hand, Double DQN was able to solve the environment in about 700 episodes, which is a great improvement over Vanilla DQN. Also in this case, the score consistently improves for all episodes of training.

6 Future work

- Instead of uniformly sampling experience tuples from the replay buffer, we can sample them based on a priority score. For instance, priority score can be defined in terms of the TD-error. If error is high, then we have a lot to learn from it, while we don't really need samples that we are actually good at estimating their values. This approach also helps in avoiding the case when some rare but important experiences may not be used much when sampling is done uniformly.
- Using Dueling DQN, which basically decouples Qvalue estimation into state value estimation $V(s)$ and advantage of each action $A(s, a)$. By doing so, intuitively our DDQN can learn which states are (or are not) valuable **without** having to learn the effect of each action at each state. This is particularly **useful for states where their actions do not affect environment in a relevant way**.

References

[1] `https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf`

[2] `https://arxiv.org/abs/1509.06461`