Navigation Project

Udacity Nanodegree on Deep Reinforcement Learning.

By Peerajak Witoonchart

Project Rubric

In this document, the project rubric are answered on the following topic. Problem Statement Topic answers the following rubrics

- The report clearly describes the learning algorithm, along with the chosen hyperparameters. It also describes the model architectures for any neural networks.
- A plot of rewards per episode is included to illustrate that the agent is able to receive an average reward (over 100 episodes) of at least +13. The submission reports the number of episodes needed to solve the environment.

Result Topic answers the following rubric

- state and action spaces, and when the environment is solved.

Ideas for Future Work answers the Future work rubic.

Problem Statement

In this project, we will show how to write a program, called an agent, to build an automatic banana eating monkey robot for a virtual continuous environment containing both yellow and dark blue bananas.

The goal of the robot is to collect as much as possible yellow bananas while avoid collecting dark blue bananas. The game ends episodically.

Suppose you are to build an automatic banana eating monkey robot, how could you do?. Suppose there are two kinds of banana, the yellow bananas which taste good, and the dark blue bananas which taste bad. How could you program the monkey robot in such a way that it selects only yellow bananas? To solve this problem, let us start with a human controllable handjoy and control the monkey robot manually. We have

- **`0`** - move forward.

- **`1`** - move backward.

- **`2`** - turn left.

- **`3`** - turn right.

, where the **'n'** is the nth botton of handjoy.

Solution

State Space Definition

Action Space Definition

Let us define this problem into the domain of reinforcement learning. This problem is episodic problem, with continuous space. That is, we have a continuous state-space. According to lessons, if we have discrete state-space, we could represent the Action-Value function in the form of Q table where actions and states are enumerated out. How ever, since we have continuous state-space, this method cannot be used because there will be too large the number of states. Instead of using table representation of a function, we use function approximation method. In this method, the idea is that There exists an Action-Value function whose values give the optimal Policy.

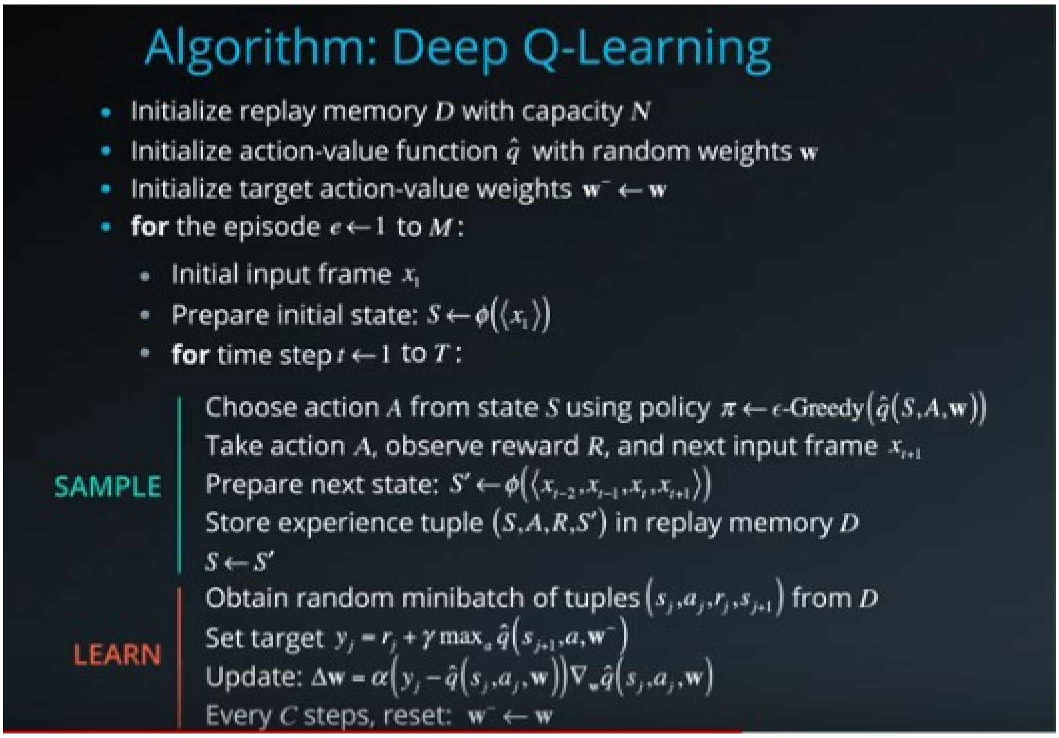
To find such a function with function approximation. We could think of a Q table as a prediction problem where state vector is an input, and the action vector is an output. The values in the state vectors correspond to the continuous value of space and other states, like velocity on each direction, position on each direction, etc. The predicted values in the output action vector is the Q table values.

Suppose such a function can be approximated by a composite function, the aforementioned prediction problem matches the definition of neural networks. We therefore, use Neural network as a mean to do this Q table prediction.

To avoid agent from believe in the early success of its training, we use *experience replay,* which is a buffer to sample at random. 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.

Now, let us looking at the training algorithm.

Algorithm: Deep Q learning



Algorithm: Deep Q-Learning

```
• Initialize replay memory $D$ with capacity $N$
• Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
• Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
• for the episode $e \leftarrow 1$ to $M$:
    • Initial input frame $x_1$
    • Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
    • for time step $t \leftarrow 1$ to $T$:
              | Choose action $A$ from state $S$ using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S,A,\mathbf{w}))$
              | Take action $A$, observe reward $R$, and next input frame $x_{t+1}$
      SAMPLE  | Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
              | Store experience tuple $(S,A,R,S')$ in replay memory $D$
              | $S \leftarrow S'$
              | Obtain random minibatch of tuples $(s_j, a_j, r_j, s_{j+1})$ from $D$
      LEARN   | Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
              | Update: $\Delta\mathbf{w} = \alpha(y_j - \hat{q}(s_j, a_j, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s_j, a_j, \mathbf{w})$
              | Every $C$ steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$
```

Figure 1. Deep Q learning algorithm

Currently, we have quite enough raw idea how to write such an agent. Let us start.

Writing an agent

The agent must have the following methods: act, learn, constructor, and the following objects: predictor models. The predictor model comprise of w minus called target network, and w, called local network. The w- (w minus) is a fixed from previous and only update from local network once a while. Another object we need is the Replay Buffer. Together, we conclude that the Agent has the following objects:

- local_qnetwork. A standard neural network, whose architecture is a hyper parameter.

- target_qnetwork A standard neural network, whose architecture must be the same as local network.

- optimizer

- Replay buffer

Agent must have the following methods

- constructor()

- get consequence(). This method input the consequence of Agent action from the environment. Hence the name.

- act()

- learn()

- targetnetwork_update().

Details of these important objects, and methods are described below. Methods will be shown with () suffix.

QNetwork

Q network is an approximation function mapping observed state to the action value. The state vector of size state_size is directly mapped to action value vector of size action_size. We use Neural network to do function approximation. Two main operations of the neural network is weight-update operation, and the forward operation. The forward operation, in Deep Q learning context, is to find the action, usually with epsilon greedy selection. The weight update operation has two steps: gradient calculation, and weights update. This can be done by sharing the access to its weights to an optimizer. The optimizer calculate the gradient, and do the weight update for Qnetwork. Therefore, the Qnetwork only has the following objects, and methods: Constructor(), Forward(), and layer objects. These layer objects are constructed to a neural network through constructor().

Act()

To find an action, we have a lookup table where we look at the current state of the agent and find the best action according to action whose value maximize the action value function. With Q network implemented as a Neural Network, the Action Values of a state can be done by forward operation. Thus forward operation is the look up operation.

selected_action = arg max local_Qnetwork.forward(S).

This is called the greedy policy. However, since we use epsilon greedy instead of greedy to avoid early confidence, instead, we have

- If the coin lands tails (so, with probability $1-\epsilon$), the agent selects the greedy action.
- If the coin lands heads (so, with probability ϵ), the agent selects an action *uniformly* at random from the set of available (non-greedy **AND** greedy) actions.

This is called Epsilon Greedy algorithm.

Once the agent acts, it get the consequence.

getConsequence()

Get from the environment: the consequence of your agent’s action. Since we use the ReplayBuffer memory, we just keep all the environmental output to the ReplayBuffer memory waiting to be learned.

ReplayBuffer

This object memorizes the past experience in the form S,A,R,S. It has BUFFERSIZE as hyperparameter. It’s main operations are sample() where BUFFERSIZE samples of the memory is output for the learning algorithm to learn, and add() operation where a tuple S,A,R,S is added to the buffer. The “done” boolean provides a flag for system to know that the episode has ended. This flag is required for our learning algorithm.

Learn()

The learn method calculates the learning part of the algorithm shown in Figure 1. This start from

- get Samples from ReplayBuffer

- Calculate target y_j, which should have a size 1x1 for j th sample. For a batch of BUFFERSIZE, this target y is a vector of the size BUFFERSIZE.

- Calculate Loss function, and gradients using local_qnetwork.

- Update the weights with optimizer object.

- Update the weight of target_qnetwork.

Training the Agent

A dqn() function provide overall training procedure described by Figure 1. It starts with looping over episodes. There is an inner loop which loop over time step, take action and get the consequence. Keep the score statistic for every episodes. It conclude the mean score for further comparision.

Result and Discussion

The following are hyperparameters for this Deep Q learning algorithm. We wish to know a set of hyperparameter values which maximize the performance of Deep Q learning algorithm. A successful hyperparameter is shown here.

```
BUFFER_SIZE = int(1e5)  # replay buffer size

BATCH_SIZE = 32         # minibatch size

GAMMA = 0.999           # discount factor

TAU =  1e-3             # for soft update of target parameters

LR = 5e-5               # learning rate
```

UPDATE_EVERY = 4   # how often to update the network

Epsilon Decay policy

eps_start=1.0, eps_end=0.01, eps_decay=0.995

number of timesteps per episode = 1000

number of episodes = 1800

Qnetwork Architecture

input → Fully Connect 1 → ReLu → Fully Connect 2 → ReLu → Fully Connect 3 → output
37 37x64 64x64 64 x 4

Loss Function

||y_j – q^(s_j, a_j, w)||^2

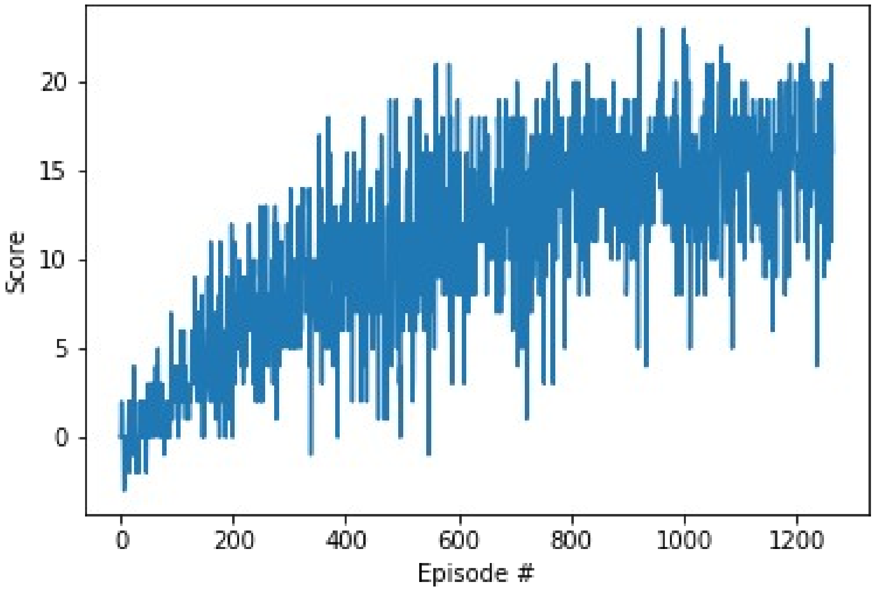


Figure 2. Training Score Vs Episode #.

Result

I train the above network with above hyperparameters. The average score shows how good the network is doing. I get 16.00 score within 1800 episodes. This is shown in Figure 2. I test the trained network with a new episode and get score of 20. I test the random action policy and get the score of 0.0. I also show how to save and download the trained network.

Ideas for Future Work

- I would like to try different hyperparameters to see which of the parameters give better result
- I would like to test with different type of Neural Network parameters.
- Can we try to make the robot learn to map the map?