SAYON PALIT

17th June 2020

Deep Reinforcement Learning

Project 1 : Navigation

OVERVIEW

This project aims to use deep reinforcement learning ,more specifically Deepmind's Deep Q Network Algorithm to solve a Unity based square world environment where the main objective is to collect the maximum possible number of yellow bananas while avoiding the blue ones.

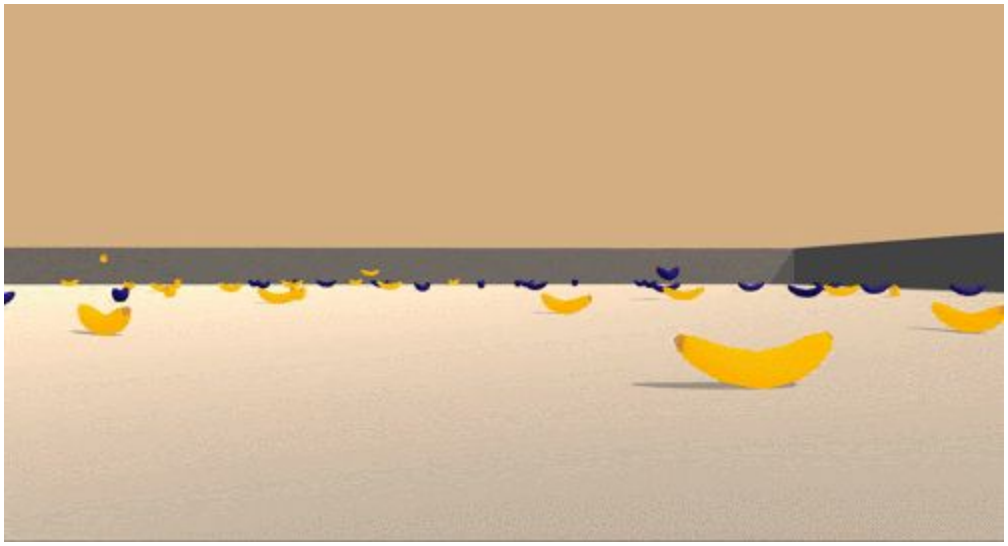


Figure 1 depicts the environment.

- 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 state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the 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 in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

LEARNING ALGORITHM

1. In this algorithm we use a deep neural network to estimate the Q-values for each state-action pair in a given environment, and in turn, the network will approximate the optimal Q-function. The act of combining Q-learning with a deep neural network is called deep Q-learning, and a deep neural network that approximates a Q-function is called a deep Q-Network, or DQN.

2. We take an arbitrary deep neural network that accepts the states from our environment as input.For any state given as input, the network returns the estimated Q-values for each action that can be taken from that state. The main objective of this network is to approximate the optimal Q-function.

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

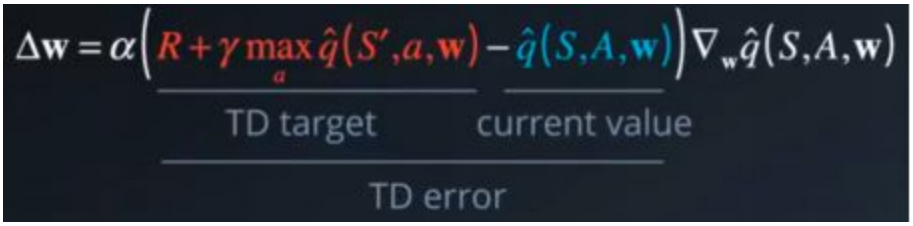
3. The loss from the network is calculated by comparing the output Q-values to the target Q-values from the right hand side of the Bellman equation, and as with any network, the objective here is to minimize this loss.

4. After the loss is calculated, the weights within the network are updated via SGD and backpropagation, again, just like with any other typical network. This process is done over and over again for each state in the environment until we sufficiently minimize the loss and get an approximate optimal Q-function.

5. We also use a technique called **Experience Replay** which is a buffer that stores experience tuples in the form (S,A,R,S′).We use replay memory to break the correlation between consecutive samples.If the network learned only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and would therefore lead to inefficient learning. Taking random samples from replay memory breaks this correlation.
6. **Fixed Q-Targets**

   In Q-Learning, when we want to compute the error, we compute the difference between the target optimal Q-value function and the current predicted Q-value (estimation of Q) . In DQN that is done by:-

   - Firstly doing a forward pass for the given input state , calculating the expected return.
   - Then using the observed next state and the current Q value policy we do another forward pass to find the actual return for the given state.



   - After the second forward we use the formula mentioned below to calculate

     The error is followed by a backpropagation step to update the network weights using Stochastic gradient descent.

     But this leads to a situation similar to a dog chasing its tail wherein as the current Q-value function moves closer to the Target optimal Q-value function ,the target function moves further away which is the root cause of instability in DQN instability.

     Rather than doing a second pass to the policy network to calculate the target Q-values, we instead obtain the target Q-values from a completely separate network, appropriately called the target network.

The target network is a clone of the policy network. Its weights are frozen with the original policy network's weights, and we update the weights in the target network to the policy network's new weights every certain amount of time steps.

7.**MODEL ARCHITECTURE**

The model consists of an input layer , 3 fully connected layers and the output layer which gives out 4 outputs corresponding to the four different actions values.

```
self.fc1 = nn.Linear(state_size, 256)
self.fc2 = nn.Linear(256,256)
self.fc3 = nn.Linear(256, action_size)
x = F.relu(self.fc1(state))
x = F.relu(self.fc2(x))

x = self.fc3(x)
```

Where state size = 37 and action size is 4.

We use ReLU activation function after each fully connected layer except for the output layer

ALGORITHM

1. Initialize replay memory capacity.
2. Initialize the policy network with random weights.
3. Clone the policy network, and call it the target network.
4. For each episode:
    1. Initialize the starting state.
    2. For each time step:
        1. Select an action.
            - Via exploration or exploitation(GLIE)
        2. Execute selected action in an emulator.
        3. Observe reward and next state.
        4. Store experience in replay memory.
        5. Sample random batch from replay memory.
        6. Preprocess states from batch.
        7. Pass a batch of preprocessed states to the policy network.

8. Calculate loss between output Q-values and target Q-values.
    - Requires a pass to the target network for the next state
9. Gradient descent updates weights in the policy network to minimize loss.
    - After x time steps, weights in the target network are updated to the weights in the policy network.

HYPERPARAMETERS

BUFFER_SIZE = int(1e4) # replay buffer size

BATCH_SIZE = 64 # minibatch size

GAMMA = 0.995 # discount factor

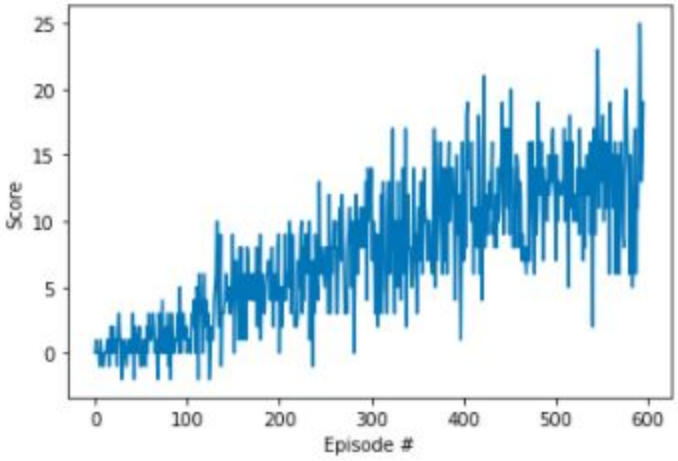
TAU = 2e-3 # for soft update of target parameters

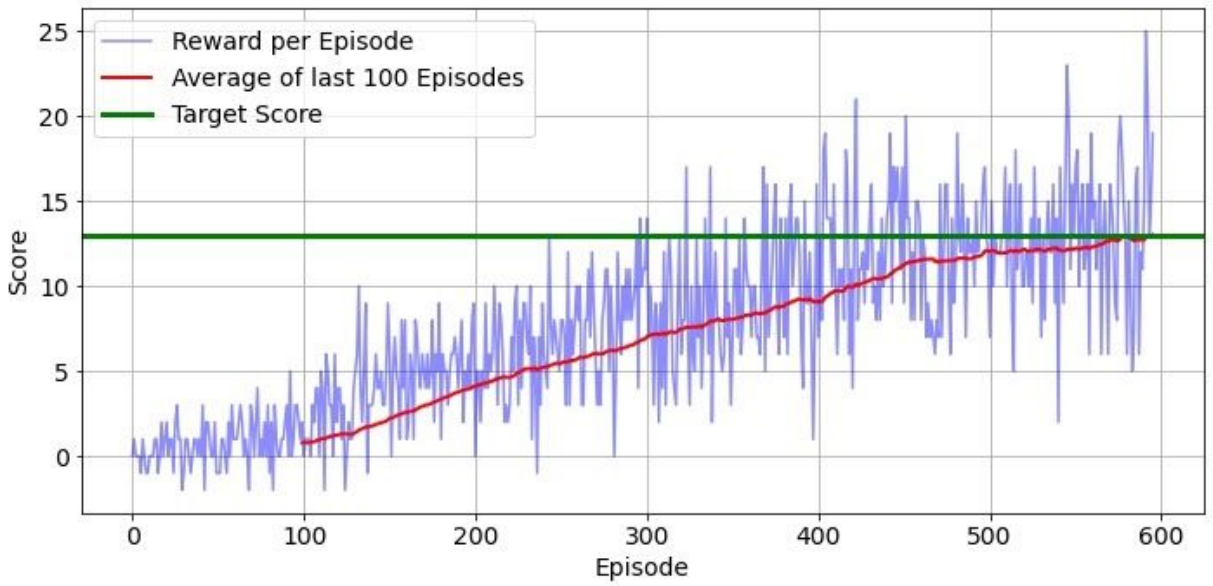
LR = 5e-4 # learning rate

UPDATE_EVERY = 5 # how often to update the target network with policy network

RESULTS

```
Episode 100     Average Score: 0.78
Episode 200     Average Score: 4.11
Episode 300     Average Score: 6.89
Episode 400     Average Score: 9.12
Episode 500     Average Score: 12.06
Episode 596     Average Score: 13.02
Environment solved in 496 episodes!     Average Score: 13.02
```





As can be seen from the plot above the given DQN is able to solve the Banana environment in 496 episodes with an average score of 13.02.

FUTURE WORK

- **Prioritized experience replay** is an improvement to the experience replay procedure, and the main idea behind it is to accord more interest on experiences that seems to be more important for the learning procedure.
- The network can be modified further to implement **pixel based learning** by adding convolutional layers that will be able to grasp spatial patterns in the environment and use that for the DQN process.
- Also the hyperparameters as well as network layer structure can be further tweaked to reduce the solving time from the current 700-800 episodes to even lower.

REFERENCES

1.Training a Deep Q-Network with Fixed Q-targets - Reinforcement Learning

2.Human-level control through deep reinforcement learning

3.Deep Reinforcement Learning Online Course - Udacity