Navigation: Banana Collection Agent

Vishal Anand S

1. Introduction

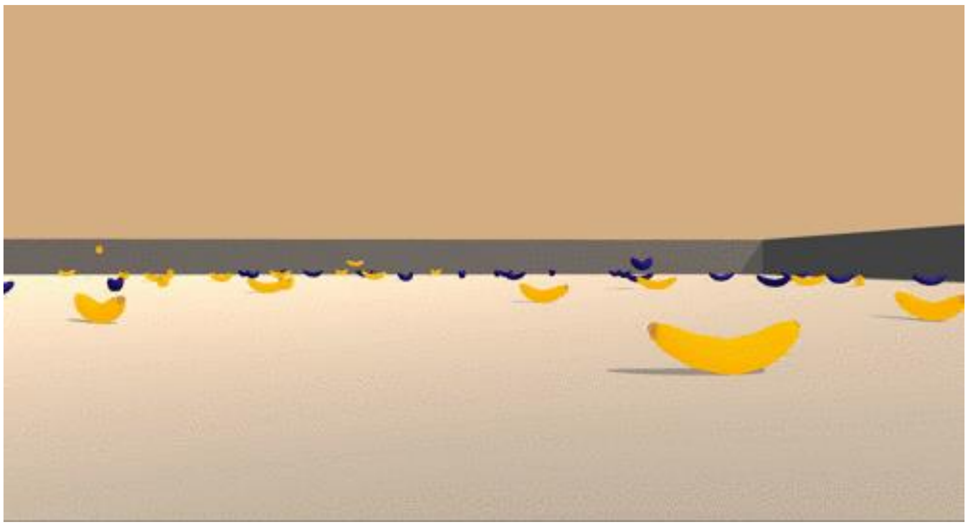
Reinforcement learning is branch in Artificial Intelligence which is thought to be a key instrument in achieving human level intelligence. This is mainly due to its learning strategy which varies from the traditional machine learning algorithms such as supervised and unsupervised learning which relies on finding patterns from dataset containing information on both inputs and their corresponding targets. Reinforcement learning works on a reward-based system where an agent explores an environment and receives rewards based on the actions taken. This report presents my implementation of Navigation project in the Udacity Deep Reinforcement Learning Nanodegree program.

The basic objective of the project is to make the agent collect yellow bananas and avoid blue bananas. Detailed explanation of the environment is presented in the below section. The agent was trained using both DQN (Deep Q-network) and DDQN (Double Deep Q-network) and the results are posted in **Results section**.

2. Environment

The environment is a large square world surrounded by gray walls. Agent is placed randomly on the environment surrounded by the objects (bananas) and is expected to collect them by navigating through the world. 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 the 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. This is the information given to the agent regarding its current state in the environment.



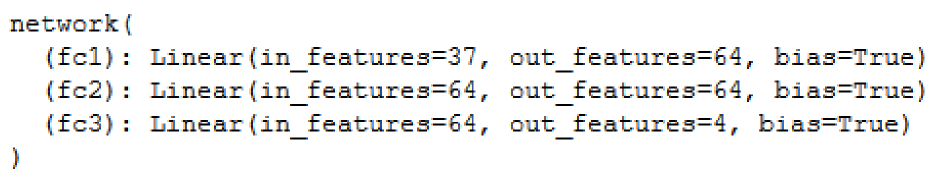
The agent can move in four discrete directions, corresponding to:

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

Given the above actions, the agent must learn to take the best action in its current state to achieve maximum future reward (collecting maximum yellow bananas avoiding blue).

3. Network Architecture

DQN varies from traditional Q-learning since it uses a deep neural network as its function approximator. The network's architecture is presented below:



```
network(
  (fc1): Linear(in_features=37, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
```

The network takes a vector of size 37 (state size) as its input and produces a vector of size 4 (number of actions). This is a basic network containing two hidden layers each containing 64 neurons.

4. Agent

The agent is responsible for the navigation process. For the agent to learn to navigate, the state space variables are given as input to the neural network and the network outputs the action which is considered as the best one in order to maximize its future reward. Improvements such Prioritized Experience Replay and Double DQ Networks were used which are explained in detail below.

4.1. Prioritized Experience Replay

Experience replay is a concept where the current state, agent's action, reward and the next state is stored in queue considered as memory. Normally when a sequential data is provided to the agent, it can get stuck in a local action set (fitting to a specific set of actions) due to the correlation between previous, current and next data patterns, so the replay buffer is made which stores all the information and a batch is sampled randomly from it. This breaks the correlation, thus giving the agent the chance to learn from almost all the possibilities.

Experience replay takes random batch of experiences from the memory which can be any state stored. This is great technique to break correlation, but the agent won't be able to reduce its error on state where the error is high. For example: If a state produces high error and the state is chosen randomly from the memory, the network won't be able to reduce the error efficiently. This introduces us to Prioritized Experience Replay where the experiences from the memory are selected based on its priority. The process is as follows:

- An additional queue is initialized which holds the priorities of the corresponding experience.
- At every step, the experience is added to the memory with the priority of 1 (or maximum priority) making the most recent experience to be selected.
- The priority for sampling the memory is calculated using:

$$\mathbf{p(i)^{\circ} / \Sigma(p)^{\circ}},$$

where p(i) is the individual priority and p is the queue containing all the priorities. Constant α (°) is the priority scale where 0 represents total randomness and 1 represents total priority.

- The use of prioritized experiences will cause a bias in the network. So, to tackle the bias

Weight = [(1 / N) * (1 / P(i))]°,

where N is the length of memory and P(i) is the set of chosen probabilities and β (°) which starts of as low value and increases to 1 over time. The weight is multiplied to the loss function,

Loss = squared loss * weights,

This reduces the step size for the biased experiences which has higher probabilities.

- Finally, the probabilities of the sampled experiences are updated using,

p(i) = |Loss| + offset,

the offset is used to avoid the zero loss occurrences.

The code for the process is present in *PrioritizedReplay.py* . At every step, both the memory and priority queues are updated. The queue can hold maximum of buffer size (100000 for this implementation).

4.1.2 Double DQN

The basis of Q learning relies on:

- The value of agent's current state (S)
- Action taken by the agent from current to next state (A)
- Reward received for reaching the next state (R)
- The value of the next state (S')

The q-value (state action value) of the current state is denoted as,

Q (S, A) = reward + (γ * Q (S', A')),

The neural network's job is to predict the best action for the given state which is **Q (S, A)** given the value of the state. In order to achieve this, two separate networks of same architecture are initialized, local and target network.

The process for DQN is as follows:

- *local_network* predicts **Q (S, A)** and *target_network* predicts **Q (S’, A’)**.
- The output from the *local_network* is considered as **Q_expected** and **Q_target** is computed using the above formula.
- Loss is computed using,

Loss = mean ((Q_expected – Q_target)2 * weights)**,

where the weights are received from priority relay.

- The *local_network* parameters are trained using optimizer (Adam in this implementation).
- The *target_network* parameters are soft updated with a value of TAU.

The above process leads to overestimation of Q values since the target_network and local_network work separately to estimate values. This is solved by using Double DQN where both the networks are used to estimate **Q (S’, A’)**. This can lead to a mutual agreement between the networks on the estimation of the Q value of the next state.

This implementation uses Double DQN to solve the environment. Code for this method is present in *agent.py* .

5. Training

The above sections provide some basics on techniques used in my implementation. This section combines everything and provides you how it is implemented in step by step manner.

The maximum episodes for the agent to train was set as 3000 which means the agent must learn the environment within the number of episodes. A single episode terminates when it reaches 100 timesteps in which the agent must score a minimum of 13 (collect 13 yellow bananas avoiding blue and wall).

At every time step the environment provides the agent its state and the rewards for reaching the state. The experience (S, A, R, S’) [as discussed in the previous section] is stored in the memory along with its priority.

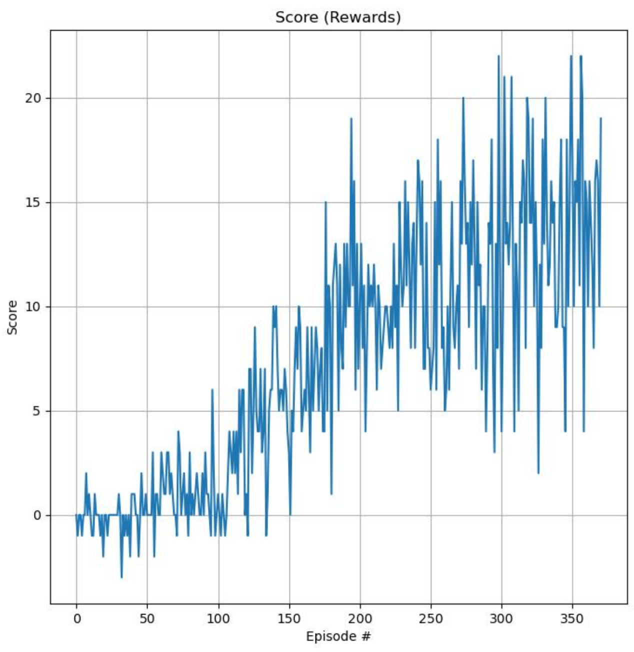
At every 4 time step the agent learns the prioritized data from the memory using DDQN method and updated the network parameters and priorities.

The agent follows varying ε-greedy policy where a random action is taken with the probability of ε and the greedy policy (from the network) is taken with probability of ε. The ε is set to 1.0 at the beginning and is decayed gradually with the rate of 0.99 as the episode furthers. This gives the agent to explore at the beginning of training and exploit as it proceeds.

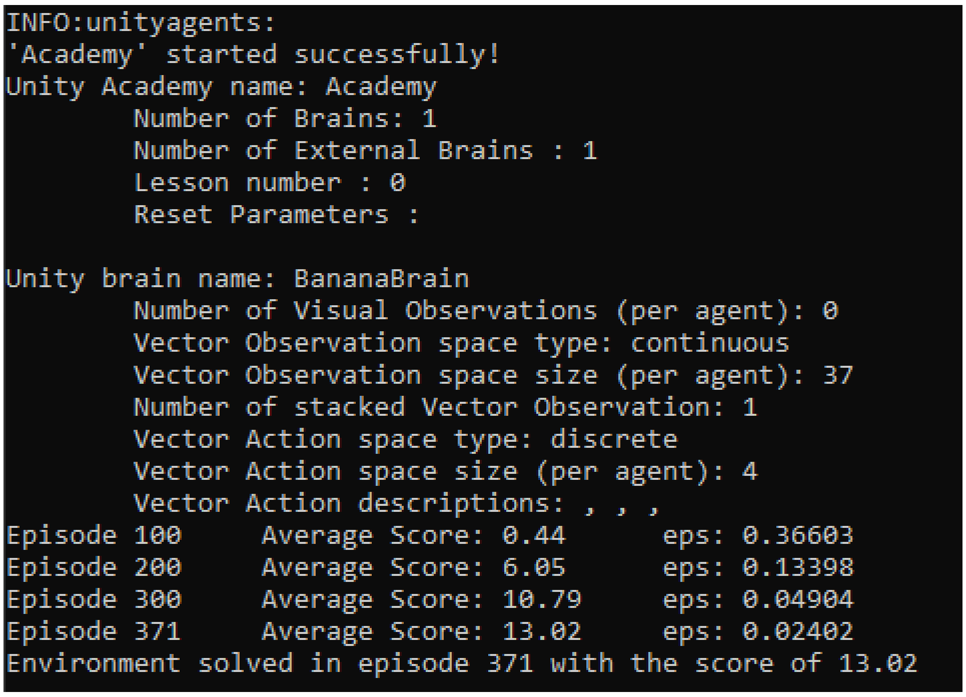
The model parameters are saved when the average score of an episode reaches 13.0 . The saved parameters stored in the file *BananaBrain.pth* and the scores are plotted and saved. The results are shown in the below section.

6. Results

I initially trained the model using *Random Replay Buffer* and *DQN* which was able to solve the environment in around 1000-1100 episodes which was the baseline of the project. I lost the score graph that training but I didn't the current one which used *DDQN* and *Prioritized Replay* which is shown below,



The agent was able to solve the environment within 371 episodes. The breakdown of score on every 100 episodes:



```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :

Unity brain name: BananaBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 37
        Number of stacked Vector Observation: 1
        Vector Action space type: discrete
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
Episode 100     Average Score: 0.44      eps: 0.36603
Episode 200     Average Score: 6.05      eps: 0.13398
Episode 300     Average Score: 10.79     eps: 0.04904
Episode 371     Average Score: 13.02     eps: 0.02402
Environment solved in episode 371 with the score of 13.02
```

Compared to DQN, DDQN trained with agent in 1/3rd of the former's time. Replay buffer is sufficient for this project as the chances of getting negative reward (collecting blue bananas) happens only once. Thus, there won't be any significance in using a Prioritized buffer.

7. References

Deep Mind DQN Paper: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Deep Mind Double DQN Paper: https://arxiv.org/pdf/1509.06461.pdf

Prioritized Experience Replay Paper: https://arxiv.org/pdf/1511.05952