Udacity Deep Reinforcement Learning

Project: Navigation

Author – Anish Amul Vaidya (avaidya172@gmail.com)

Abstract - Project Navigation: 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] (https://github.com/Unity-Technologies/ml-agents).

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 our 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 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.

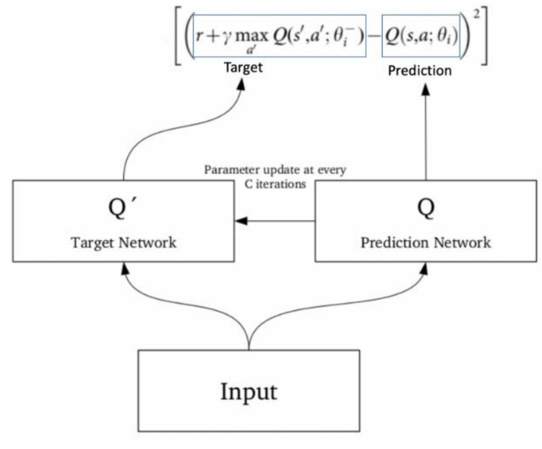
The task is episodic, and in order to solve the environment, our agent must get an average score of +13 over 100 consecutive episodes.

Methods –

There are 2 methods to solve this problem:

1. **Vanilla DQN with experience replay and Fixed – Q targets :**
   The agent was trained on a Deep Q network. We use two different models with the same architecture. We use one model for training and another for doing prediction. We update the target network from the prediction network after some time. We use the MSE error function to calculate the error in our network.

   We store the agent's experiences at each time step in a data set called the replay memory. All of the agent's experiences at each time step over all episodes played by the agent are stored in the replay memory. A key reason for using replay memory is to break the correlation between consecutive


$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

samples.

Architecture

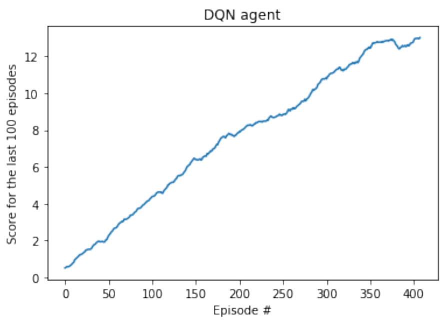
```
state_size = 37
action_size = 4
    self.fc1 = nn.Linear(state_size, 128)
    self.fc2 = nn.Linear(128, 64)
    self.fc3 = nn.Linear(64, 32)
    self.fc4 = nn.Linear(32, action_size)
```

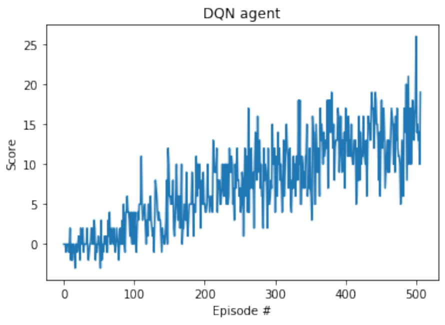
Hyperparameters

Learning Rate (LR): = 5e-4
Discount Factor (GAMMA): = 0.99
Soft update of target parameters (TAU) = 1e-3
Batch size (BATCH_SIZE) = 64
Replay memory buffer (BUFFER_SIZE) = int(1e5)

Training the model

The model finishes training in 400 - 500 episodes where it achieves an average score of +13.
The average score is calculated over a period of 200 episodes.
Following are some plots of the training info:





2. **DQN with convolutional neural networks to learn from pixel data –**
We can modify our vanilla DQN network to input visual data (pixels) from the environment and use convolutional neural network on top of the linear model to train the agent with respect to the image data.

Architecture

```
    input_image_size = (3, 84, 83)          #(3 channels RGB, 84 * 84 image)
    self.conv1 = nn.Conv2d(3, 32, 3, 2)     # 41 * 41 * 32
```

```
        self.conv2 = nn.Conv2d(32, 64, 5)     # 37 * 37 * 64
        self.fc1 = nn.Linear(87616, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 32)
        self.fc4 = nn.Linear(32, action_size)
```

Hyperparameters
Learning Rate (LR): = 5e-4
Discount Factor (GAMMA): = 0.99
Soft update of target parameters (TAU) = 1e-3
Batch size (BATCH_SIZE) = 64
Replay memory buffer (BUFFER_SIZE) = int(1e5)

Future improvements:

Prioritized Experience Replay (aka PER): It is an improvement over the default DQN. Prioritized sampling, as the name implies, will weigh the samples so that "important" ones are drawn more frequently for training. The magnitude of the TD error (squared) is what we want to minimize in the Bellman equation. Hence, pick the samples with the largest error so that our neural network can minimize it.

Double DQNs: we use two 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 and use our target network to calculate the target Q value of taking that action at the next state. Double DQN helps us reduce the overestimation of q values and, as a consequence, helps us train faster and have more stable learning.

Dueling DQNs: We decompose Q(s, a) into:
V(s): the value of being at that state
A(s, a): the advantage of taking that action at that state
separate the estimator of these two elements, using two new streams:
One to estimate V(s) and another to estimate advantage for each action A(s,a)
We combine them using an aggregation layer to get an estimate of Q(s, a).