Navigation

This document provides description of the implementation for the Navigation project.

The Environment

For this project, an agent is required to be trained to navigate (and collect bananas!) in a large, square 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 your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

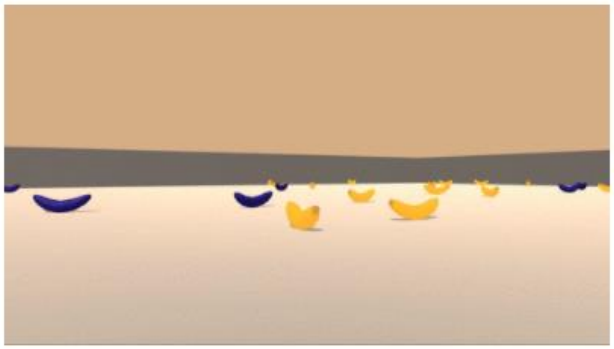

Figure 1. Unity ML 'Banana' Environment

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.

The Implementation

The implementation uses DQN (Deep Q-Network) to solve the task. A DQN agent interacts with and learns from the Unity ML environment 'Banana' for the task.

The agent implementation is written in PyTorch and the Python API was used to control the agent.

The solution has following files:

- ***navigation.ipynb***: Jupyter notebook, code for training the agent.

- ***dqn_agent.py***: The Agent class and ReplayBuffer class. The sampled batch of experience tuples is provided thru ReplayBuffer class. ReplayBuffer class implements a fixed-size buffer to store experience tuples. The local and target Q-networks are used to compute the loss, before taking a step towards minimizing the loss.

- ***model.py***: The QNetwork class. Defines a neural network architecture that maps states to action values.

- ***checkpoint.pth***: Saved model weights of the trained agent.

DQN network architecture consists of three fully connected layers. Input layer has 37 inputs (corresponding to 37 dimensions of the environment state) and 64 outputs, hidden layer has 64 inputs and outputs, output layer has 64 inputs and 4 outpus (corresponding to 4 possible actions of the agent). Figure 2 shows the illustration of DQN architecture for this implementation.

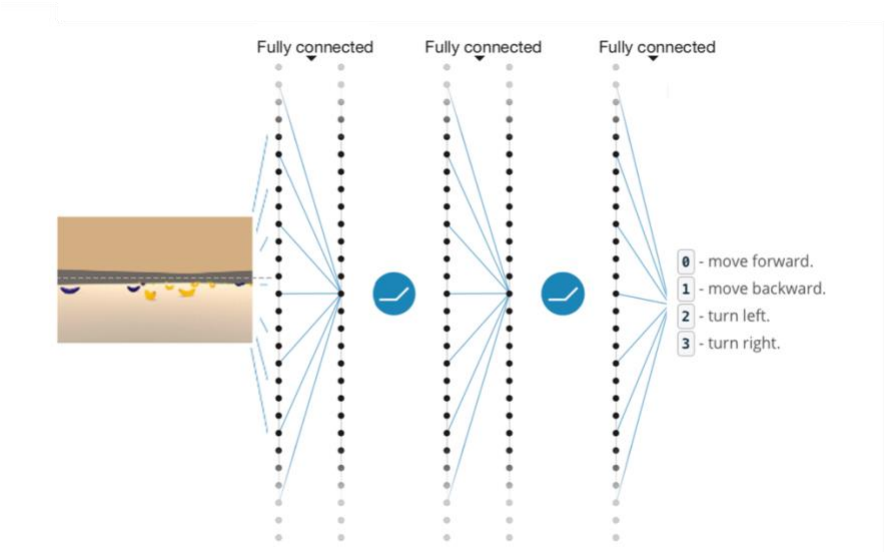

Figure 2. Illustration of DQN Architecture

The final architecture of the DQN model looks like the following:

```
QNetwork(
  (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)
)
```

Adam optimizer is used as the optimizer.

The **QNetwork** class has following methods:

- *__init__*: Initializes parameters and build model
- *forward*: Builds a network that maps state -> action values

The **ReplayBuffer** class has following methods:

- *__init__*: Initializes a ReplayBuffer object.
- *add*: Adds a new experience to memory.
- *sample*: Randomly samples a batch of experiences from memory.
- *__len__*: Returns the current size of internal memory.

The **Agent** class has following methods:

- *__init__*: Initializes an Agent object.
- *step*: Saves experience in replay memory.
- *act*: Returns actions for given state as per current policy.
- *learn*: Updates value parameters using given batch of experience tuples.
- *soft_update*: Soft updates model parameters.

The agent was trained in a Jupyter notebook.

Environment was solved in **523** episodes with an average score of **13.02**. Figure 3 shows how agent's score evolves throughout the training. It can be observed that the agent is able to receive an average reward (over 100 episodes) of at least +13.

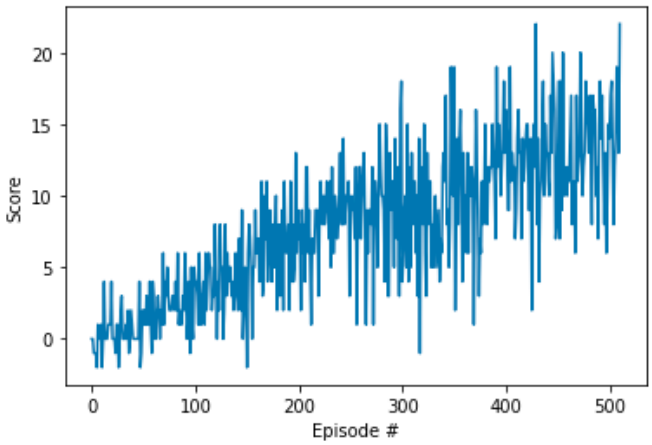

Figure 3. Plotted Rewards

It is observed that the agent was trained quickly enough on CPU, GPU support was not needed.

Hyperparameters and Settings

The various hyperparameters and settings used througout the implementation are given below:

```
BUFFER_SIZE = int(1e5)      # replay buffer size
BATCH_SIZE = 64             # minibatch size
GAMMA = 0.99                # discount factor
TAU = 1e-3                  # for soft update of target parameters
LR = 5e-4                   # learning rate
UPDATE_EVERY = 4            # how often to update the network

n_episodes =  2000          # maximum number of training episodes
max_t = 1000                # maximum number of timesteps per episode
eps_start =  1.0            # starting value of epsilon, for epsilon-greedy action selection
eps_end =  0.01             # minimum value of epsilon
eps_decay =  0.995          # multiplicative factor (per episode) for decreasing epsilon

state_size = 37             # dimension of each state
action_size = 4             # dimension of each action

fc1_units = 64              # number of nodes in first hidden layer
fc2_units = 64              # number of nodes in second hidden layer
```

DQN Improvements

Although they are not tried in the implementation, following three Deep Q-Learning improvements can be made to the implementation to improve the agent's performance:

- Double DQN
- Prioritized Experience Replay
- Dueling DQN