Report

October 5, 2019

1 Introduction

The aim of this project was to train an agent to navigate and collect bananas in a large, square world. A reward of +1 was given for collecting a yellow banana and -1 for purple banana. Thus, the goal of our agent was to collect more yellow bananas while avoiding the blue bananas.

The state space had 37 dimensions and contains agent's velocity along with ray-based perception of objects around agent's forward direction. Given this information, the agent's task was to select an action from the following 4 actions:

- `0`: move forward
- `1`: move backward
- `2`: turn left
- `3`: turn right

The task was episodic and the task was considered solved if the agent achieved an average score of +13.0 over 100 consecutive episodes

To get a simulation of the environment, we used the API of **unityagents**. This api provided us with the simulation of the environment which helped to save a lot of time.

2 Model

We used a Deep Neural Network as a function approximator for the Q-function. This is called Deep Q-learning. The architecture we chose for the DNN was of 3 linear layers. The first layer was having input dimension 37 and output dimension of 128. The second layer was having input dimension of 128 and output dimension of 64 and the last layer had input dimension 64 and output dimension of 4 where 4 specifies the number of actions. The activation function for each layer is a RELU function. Here's the summary of the model used in the agent:

```
In [15]: summary(agent.qnetwork_local, input_size=(1, 37))

----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 128]           4,864
            Linear-2                [-1, 1, 64]           8,256
            Linear-3                 [-1, 1, 4]             260
================================================================
```

```
Total params: 13,380
Trainable params: 13,380
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.05
Estimated Total Size (MB): 0.05
----------------------------------------------------------------
```

The model was trained using Gradient Descent with the **Adam optimizer** to update the weights.

3 Agent

As stated above, the agent was a Q-learning agent with a 3-layered neural network as a function approximator. Q-learning is a famous algorithm of reinforcement learning which learns the action-value function for a given policy.

The main feature of q-learning as compared to SARSA algorithm is that it directly learns optimal q-value instead of switching between evaluation and improvement

Non-linear function approximators suffer from the problem of instability. To imporve convergence, the modifications made are:

- **Experience Replay** To avoid learning experiences in sequence (i.e correlated experiences), we use a buffer memory called replay buffer to store the experience tuple (consisting of state, action, reward and next state). We allow agent to randomly sample experiences from this buffer. This will allow us to learn from same experience multiple times. This is very helpful if we encounter certain rare experiences
- **Fixed Q-values** The TD target is dependent on the network parameter w that we're trying to learn. This can lead to instability. To remove this, we use a separate network with identical architecture. The target network gets updated slowly with hyperparameter TAU and local network updates aggressively with each update called **soft update**.

The learning algo uses something called an **Epsilon-Greedy** policy to select actions while being trained. Epsilon specifies the probability of selecting a random action instead of following the "best action" in the given state (exploration-exploitation tradeoff)

The agent class defines how the agent will act provided an action, learn from a time step, update the network every UPDATE_EVERY time steps and store the experiences in the memory buffer.

To use experience replay, we define a memory buffer class. An object of ReplayBuffer initializes a deque of maximum length BUFFER_SIZE to store experience tuples. It has methods to store tuples and sample a set of tuples given a BATCH_SIZE

4 DQN Algorithm

Below is the dqn method which contains the DQN algorithm. It returns list of scores for all episodes and terminates when the reward value >= 15.0 is achieved. We used epsilon decays

of 0.995 starting from an epsilon of 1.0 and ending at an epsilon of 0.01.

The relevant hyperparameters found were:

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

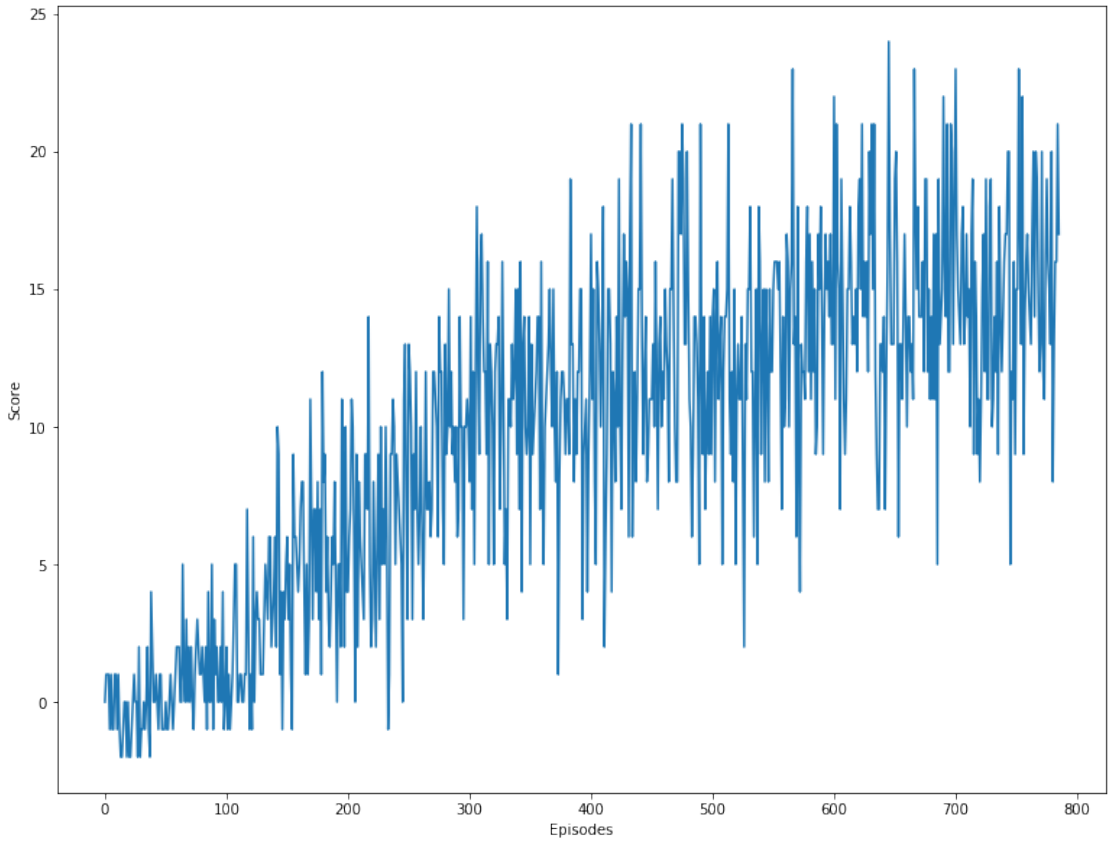
These parameters were chosen as in the Lunar Lander example in lesson 2. These worked very well for this project

```
In [7]: def dqn(n_episodes=2000, max_t=100000, eps_start=1., eps_end=.01, eps_decay=.995):
            scores = []
            scores_window = deque(maxlen=100)
            eps = eps_start

            for i_episode in range(n_episodes):
                env_info = env.reset(train_mode=True)[brain_name]
                state = env_info.vector_observations[0]
                score = 0
                for t in range(max_t):
                    action = (int)(agent.act(state, eps))
                    env_info = env.step(action)[brain_name]
                    next_state = env_info.vector_observations[0]
                    reward = env_info.rewards[0]
                    done = env_info.local_done[0]
                    agent.step(state, action, reward, next_state, done)
                    state = next_state
                    score+=reward
                    if done: break
                scores_window.append(score)
                scores.append(score)
                eps = max(eps_end, eps_decay * eps)
                print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_win
                if i_episode % 100 == 0:
                    print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores
                    save_model(agent.qnetwork_local, i_episode)
                if np.mean(scores_window) >= 15.:
                    print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format

                    torch.save(agent.qnetwork_local.state_dict(), 'model.pth')
                    break
            return scores
```

5 Result

An average score of 15.0 over the last 100 episodes was achieved at 685 episodes

```
In [10]: fig = plt.figure(figsize=(13, 10))
         ax = fig.add_subplot(111)
         plt.plot(np.arange(len(scores)), scores)
         plt.ylabel('Score')
         plt.xlabel('Episodes')
         plt.show()
```



5.1 Future work to consider:

I'm planning to add following features in this:

- Duelling DQN
- Double DQN
- Prioritized Experienced Replay

Apart from this, i'm also planning to use DQN to train using the pixels of the environment, similar thing was done in the DQN paper

```
In [ ]:
```