Report June 23, 2021

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 a purple banana. Thus, the goal of our agent was to collect more yellow bananas while avoiding the blue bananas.

The state space has 37 dimensions and contains agent's velocity along with ray-based percep- tion 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

We used the prebuilt unity environment to train an agent .

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 had input dimension 37 and output dimension of 128. The second layer had 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.

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

Below is code snippet for model

```
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=128,
fc2_units=64):
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

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 opti- mal q-value instead of switching between evaluation and improvement.

Non-linear function approximators suffer from the problem of instability. To improve conver- gence, 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 the agent to randomly sample experiences from this buffer. This will allow us to learn from the same experience multiple times.

- **FixedQ-values:** The TD target is dependent on the network parameter 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 hyperparameters and local network updates aggressively with each update called **soft update**.

The learning algorithm uses 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, provide an action, learn from a time step, update the network every **UPDATE_EVERY**(defined in hyperparameters) time step 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(**defined in hyperparameters**)** to store experience tuples. It has methods to store tuples and sample a set of tuples given a **BATCH_SIZE(**defined in hyperparameters**)**

Below is agent class   implementation

```
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

class Agent():
    def __init__(self, state_size, action_size, seed, fc1_units,
fc2_units):
        self.state_size = state_size
        self.action_size = action_size
        self.seed = seed

        self.qnetwork_local = QNetwork(self.state_size, self.action_size,
```

```
seed, fc1_units=fc1_units, fc2_units=fc2_units).to(device)
        self.qnetwork_target = QNetwork(self.state_size, self.action_size,
seed, fc1_units=fc1_units, fc2_units=fc2_units).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(),
lr=LR)

        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE,
seed)

        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)
        self.t_step = (self.t_step + 1) % UPDATE_EVERY

        if self.t_step == 0:
            if len(self.memory)>BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        if random.random()>eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        states, actions, rewards, next_states, dones = experiences

        Q_targets_next =
self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        Q_targets = rewards + (gamma * Q_targets_next * (1-dones))

        Q_expected = self.qnetwork_local(states).gather(1, actions)

        loss = F.mse_loss(Q_expected, Q_targets)
        self.optimizer.zero_grad()
```

```python
        loss.backward()
        self.optimizer.step()

        return self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_network, target_network, tau):
        for target_param, local_param in zip(target_network.parameters(), local_network.parameters()):
            target_param.data.copy_(tau * local_param.data + (1. - tau)*target_param.data)

class ReplayBuffer:
    def __init__(self, action_size, buffer_size, batch_size, seed):
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple('Experience', field_names=['state', 'action', 'reward', 'next_state', 'done'])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        return len(self.memory)
```

DQN Algorithm

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

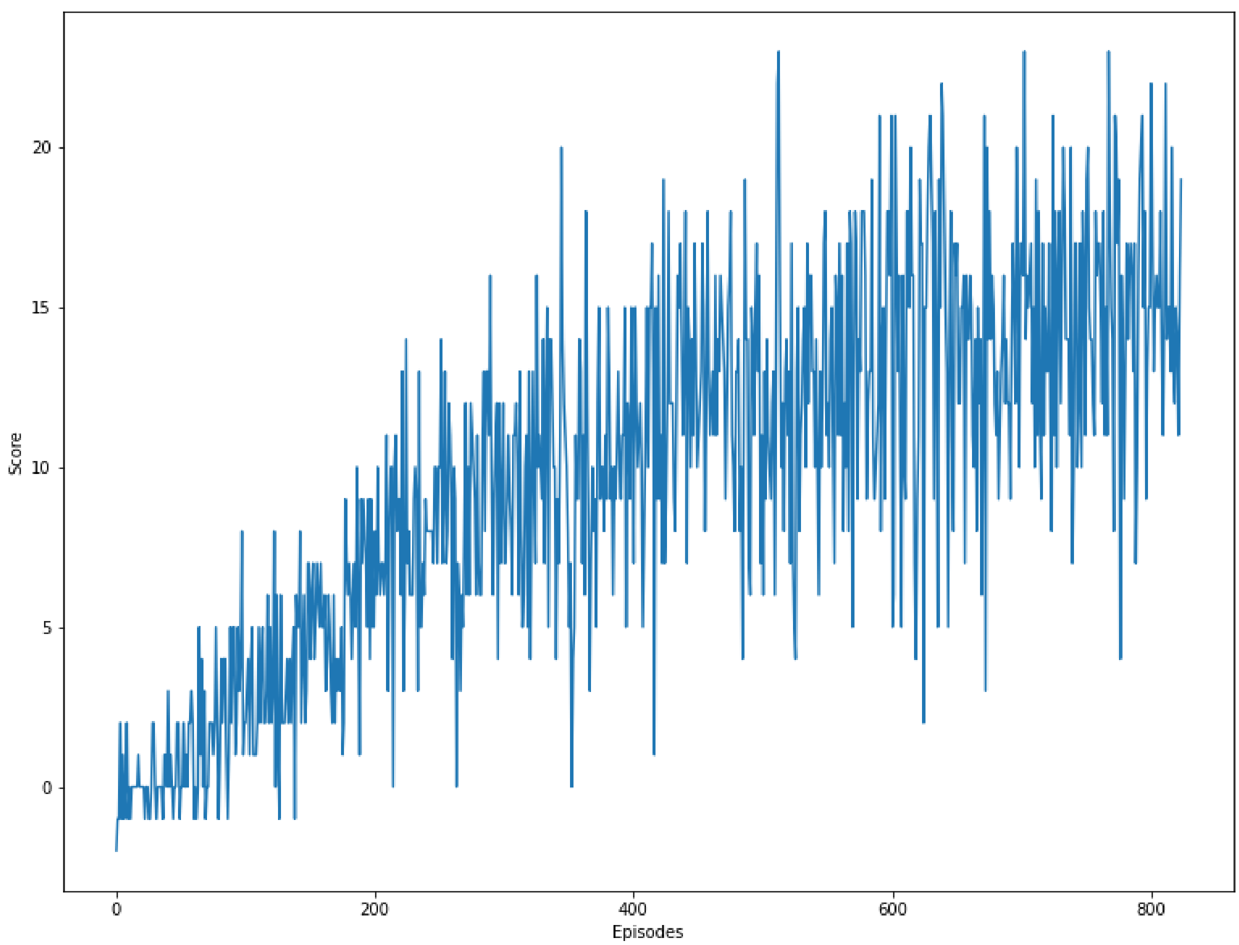
```
def dqn(n_episodes=2000, max_t=100000, eps_start=1., eps_end=.01,
eps_decay=.995):
    scores = []
    scores_window = deque(maxlen=100)
    eps = eps_start

    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_window)), end='')
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
np.mean(scores_window)), end='')
            save_model(agent.qnetwork_local, i_episode)
        if np.mean(scores_window) >= 15.:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score:
{:.2f}'.format(i_episode - 100,

np.mean(scores_window)))
```

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

Result

The agent was able to achieve an average score of 15.0 over the last 100 episodes at 723 episodes.

Plot showing the score per episode over all the episodes.



Future work to consider:

- Duelling DQN
- Double DQN
- Prioritized Experience Replay