Report

18 February 2023

Introduction

This project involves training an agent to navigate a virtual world and collect yellow bananas while avoiding blue bananas. The agent interacts with the world using a Unity-based environment provided by Unity Technologies. The environment is called Banana Collector and contains 37 state features that the agent can observe, such as the agent's velocity and the ray-based perception of objects in the agent's forward direction. The agent takes actions based on the observed state and receives a reward of +1 for collecting a yellow banana and -1 for collecting a blue banana. The goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

State Space

The state space has 37 dimensions and includes the agent's velocity and ray-based perception of objects in the agent's forward direction.

Action Space

The action space consists of four discrete actions:

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

Solved Criteria

This project is considered solved when the agent achieves an average score of at least 13 over 100 consecutive episodes. The score for each episode is the sum of rewards obtained during the episode.

Learning Algorithm

A Deep Q-Network (DQN) using Experience Replay and Fixed Q-Targets was implemented. The network has 3 linear layers and the activation function for each layer is a Rectified Linear Unit (RELU) function.

The dimensions of the network are as follows:

	Input Dimensions	Output Dimensions
1st Layer	37	128
2nd Layer	128	64
3rd Layer	64	4

The output of the last layer corresponds to the number actions in the action space. The network was trained using Gradient Descent with the Adam optimizer.

Using non-linear function approximator such as the 3-layered neural network in the Q-learning agent causes the algorithm to diverge. To ensure stability, two methods were used i.e., **Experience Replay** and **Fixed Q-Targets.** Actions are selected based on **Epsilon-Greedy** policy.

The hyperparameter are as follows:

```
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
```

```
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        state = env_info.vector_observations[0]
        score = 0
        for t in range(max_t):
            action = 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)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
        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)))
        if np.mean(scores_window)>=13.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode-100,
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
```

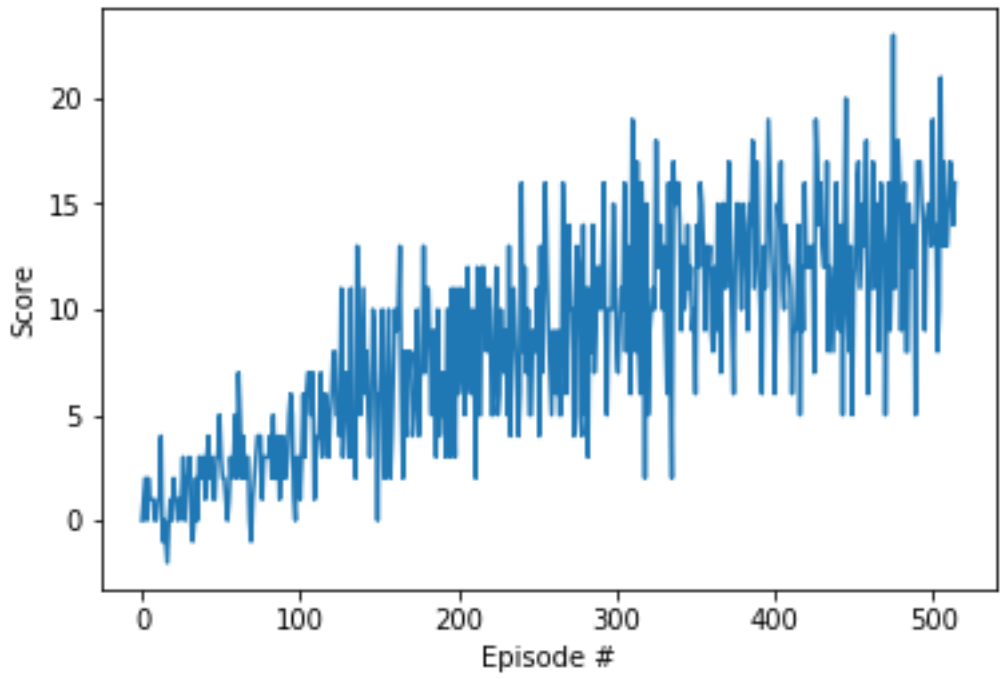
Results

An average score of 13 was achieved in **415 episodes**.

```
scores = dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

```
Episode 100     Average Score: 2.02
Episode 200     Average Score: 6.28
Episode 300     Average Score: 8.85
Episode 400     Average Score: 11.75
Episode 500     Average Score: 12.47
Episode 515     Average Score: 13.00
Environment solved in 415 episodes!     Average Score: 13.00
```



The score obtained when testing the trained agent was 14.

```
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))

env_info = env.reset(train_mode=False)[brain_name]

state = env_info.vector_observations[0]
score = 0

while True:
    action = agent.act(state)
    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]
    state = next_state
    score += reward
    if done:
        break

print('Score: ' + str(score))
```

```
Score: 14.0
```

Future Works

To improve the performance of the agent, below are some of the possible future works.

- Double Deep Q-Network
  This technique can reduce the overestimated values achieved in DQN.
- Prioritized Experience Replay
  This technique learns more efficiently by replaying important transitions more frequently unlike in the current implementation where all transitions are sampled uniformly.
- Dueling DQN
  This network represents two estimators i.e., one for the state value function and one for the state-dependent action advantage function. So, it can determine the value of each state independent of the effect of the action.