Project 1: Navigation Report

1. Introduction

The aim of this project was to train an agent to navigate a large square world full of bananas, where the agent would receive a reward of +1 for collecting a yellow banana and -1 or a blue banana. Thus, the goal of the agent was to collect as many yellow bananas as possible while avoiding the blue bananas.

The environment used contained a state space of 37 dimensions and the agent's velocity that utilised a ray-based perception of objects around its forward direction. Given this information, the agent was tasked with selecting one of the following actions:

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

The approach taken consisted of an episodic solution to the environment, where the task was considered solved when the agent achieved an average score of +13.0 over 100 consecutive episodes. The environment is based on an API from Unity's Machine Learning Agents Toolkit (ML-Agents), which enables simulations to serve as environments for training Reinforcement Learning agents.

2. Model Implementation

The model implemented focuses on a Deep Q-Learning neural network, where the architecture used consisted of four layers: an input layer with 37 inputs representing the environment state space, two hidden layers both with 64 hidden nodes and an output layer with 4 output nodes, one output for each corresponding action. Both the hidden layers are passed through a ReLU activation function to provide non-linearity to the models training and are trained using Gradient Descent with the Adam optimizer to update the models weights.

The agent uses Q-learning with an action-value function, while following an epsilon-greedy policy to select the appropriate action to achieve the best reward for a given state. Epsilon is a parameter that specifies the probability of selecting a random action, instead of 'greedily' choosing the best action within a given state. When neural networks are used with an action-value function, they can become very unstable due to the non-linearity provided by the function approximator (neural network). To improve the models convergence, two modifications are applied: an experience replay buffer and fixed Q-targets.

The experience replay buffer is used to store agent experiences (state, action, reward and next state) when the agent interacts with the environment, allowing the agent to learn additional information from each experience. Using this replay buffer, the agent can learn from individual experiences multiple times and recall rare occurrences after randomly sampling experiences from the buffer.

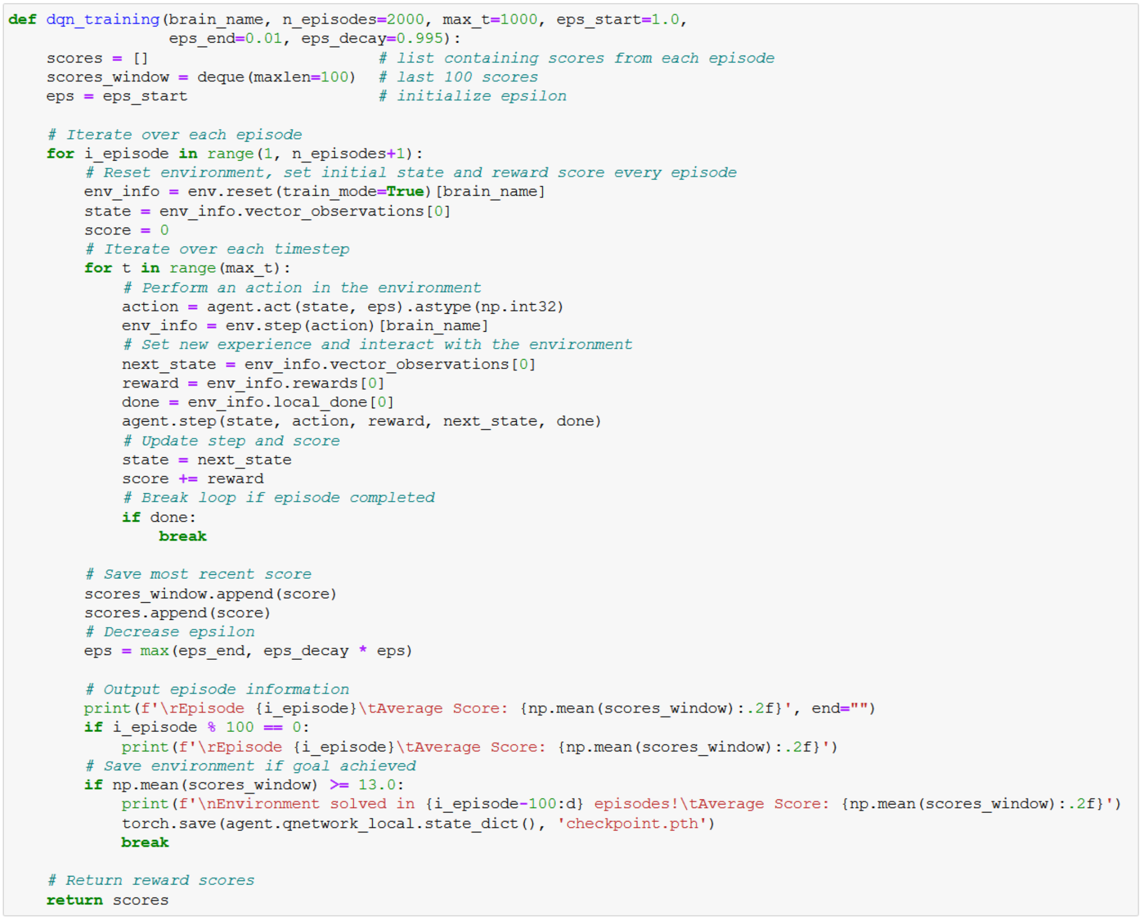
Fixed Q-targets are used to generate the TD target within the TD error and prevent correlation between the TD target and the parameters that are being changed. Without this, it would be like chasing a moving target, which can cause instability within the network. The fixed Q-targets (w^-) are a copy of the parameters (w) that are slowly updated after a set number of learning steps, called a soft update. The targets are a separate network, using an identical architecture as the original, that are compared against the parameters that are updated at every learning step.

3. Deep Q-Network Algorithm

The DQN algorithm focuses on one core function: dqn_training. This function returns a list of scores for all episodes, once the model has achieved an average reward score of over 13.0. The hyperparameters used are as follows:

- BUFFER_SIZE = int(1e5): the experience replay buffer size stores a maximum of 100,000 experiences.
- BATCH_SIZE = 64: a total of 64 samples were created and stored within the replay buffer at a time.
- GAMMA = 0.99: a discount factor of 0.99 is used during the fixed Q-targets calculation.
- TAU = 1e-3: a tau factor of 0.001 is used within the soft update of the target parameters.
- LR = 5e-4: a learning rate of 0.0005 is used to create minor adjustments to the weight parameters within the Adam optimizer during training.
- UPDATE_EVERY = 4: at every 4 learning steps the Q-targets are updated to the latest parameters.

The hyperparameters are utilised within the dqn_agent.py file, which focuses on the Agent and ReplayBuffer classes. The Agent class is a key component of the DQN algorithm, which can be seen in Figure 1, while the ReplayBuffer class is a component of the Agent class.



```python
def dqn_training(brain_name, n_episodes=2000, max_t=1000, eps_start=1.0,
                 eps_end=0.01, eps_decay=0.995):
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon

    # Iterate over each episode
    for i_episode in range(1, n_episodes+1):
        # Reset environment, set initial state and reward score every episode
        env_info = env.reset(train_mode=True)[brain_name]
        state = env_info.vector_observations[0]
        score = 0
        # Iterate over each timestep
        for t in range(max_t):
            # Perform an action in the environment
            action = agent.act(state, eps).astype(np.int32)
            env_info = env.step(action)[brain_name]
            # Set new experience and interact with the environment
            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)
            # Update step and score
            state = next_state
            score += reward
            # Break loop if episode completed
            if done:
                break

        # Save most recent score
        scores_window.append(score)
        scores.append(score)
        # Decrease epsilon
        eps = max(eps_end, eps_decay * eps)

        # Output episode information
        print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}', end="")
        if i_episode % 100 == 0:
            print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}')
        # Save environment if goal achieved
        if np.mean(scores_window) >= 13.0:
            print(f'\nEnvironment solved in {i_episode-100:d} episodes!\tAverage Score: {np.mean(scores_window):.2f}')
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break

    # Return reward scores
    return scores
```

Fig. 1. DQN algorithm code

The algorithm accepts six parameters: the agent brain to use to solve the environment, the number of episodes, the number of timesteps per episode, epsilons starting value, epsilons ending value and epsilons decay rate after each episode. The algorithm starts by iterating over each episode, where it resets the environment, initializes the first state and sets the score to 0.

Next, the model training begins by creating the agents action and uses it to interact with the environment, where a new experience is generated, stored and the Q-targets are updated after every 4 timesteps. Once the experience is generated and stored, the reward produced is added to the total and a new state is set. This process is repeated for the maximum number of timesteps specified, where the final timestep concludes the episode. Once the episode has finished, the scores are stored within a list, epsilon is reduced by the decay rate and the average score is returned and checked for the average score goal of 13.0. If the score has been achieved, the models best parameters are stored, and the algorithm finishes with returning a full list of the scores over each episode.

4. Results

The results achieved by the trained agent show an average score of over 13.0 over the last 100 episodes within 421 episodes, this is highlighted in Figure 2.

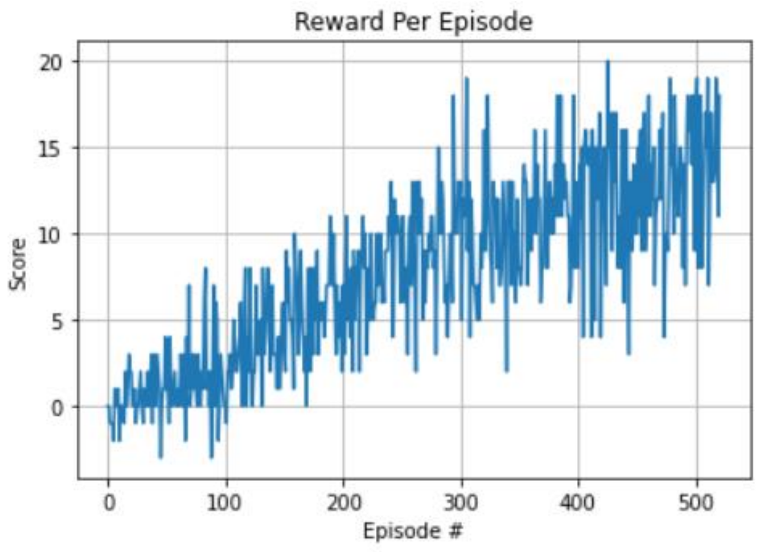


Fig. 2. Plot showing the reward per episode

Looking at Figure 2, the models training is very noisy, constantly performing poorly over a large quantity of episodes, while gradually improving overtime.

5. Ideas for Future Work

To improve the results agents results, DQN extensions can be implemented. Some of these includes: Double DQN, helping to prevent an overestimation of action-values; Prioritised Experience Replay, helping agents to learn more effectively through prioritising important and unique experiences; and Dueling DQN, helping to assess the value of each state without having to learn the effect of each action.

Alternatively, another implementation to solve the environment can consist of a combination of both convolutional layers and fully-connected layers, over the basic approach of strictly fully-connected layers. This approach focuses on training a DQN by learning the pixels of the environment.