Project 3: Collaboration and Competition Report

1. Introduction

The aim of this project was to train two agents to keep to a ball within the air while playing table tennis. The environment involves two agents that control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1 If an agent lets a ball hit the ground, or hits the ball out of bounds, it receives a reward of -0.01. Thus the goal of each agent is to keep the ball in play for as long as possible.

The state space consists of 8 variables corresponding to the position and velocity of the ball and racket, where each agent receives its own, local observation. They both have two continuous actions that correspond to movement toward (or away) from the net and a jump.

The approach taken uses an MADDPG model with the Ornstein-Uhlenbeck process and an Experience Replay, solving the environment in **X** episodes with an average score of +0.5 over 100 consecutive episodes, across both agents.

2. Model Implementation

The model implemented focuses on a Multi-Agent Deep Deterministic Policy Gradient (MADDPG), where the architecture used consists of two neural networks, an actor and a critic. Both contained four layers: an input layer with 24 inputs representing the environment state space, two hidden layers, one with 256 hidden nodes and the other with 128 hidden nodes, and an output layer with 2 output nodes, one output for each corresponding action. There is one addition to the critic's network within the first hidden layer, the actions are added as inputs to create the effect of a Deep Q-Network (DQN). Both the hidden layers are passed through a ReLU activation function to provide non-linearity to the models training and are trained using Gradient Ascent with the Adam optimizer to update the models weights. Additionally, the batches of data passed into each network are normalized to help accelerate the agents training.

THE MADDPG extends the functionality of a normal DDPG and provides an element of centralized training with decentralized execution, allowing the agents to use extra information to ease training. This focuses on the critic being augmented with extra information about the policies of other agents. Centralized planning provides each agent a direct access to local observations (states), where the agents are guided by a centralized critic. This provides feedback to the agents, like a teacher, advising them on how to improve their policies.

During testing, the centralized critic is removed, leaving only the agents, their policies and the environment states. This helps to reduce the detriments of increasing state and action space due to joint policies, which are never explicitly learned.

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, three modifications are applied: an experience replay buffer, fixed Q-targets and the Ornstein-Uhlenbeck process.

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. Both the actor and critic use their own separate sets of weights to account for this implementation by using a soft updates strategy. At every step, a minimum of 0.01% of the regular network weights blended with the target network weights.

The Ornstein-Uhlenbeck process (OUP) is used to add noise to the action itself, providing the agent an opportunity to explore the environment more. As the actions used are continuous, an epsilon-greedy policy isn't sufficient enough on it's own for the agents to successfully explore the environment, hence the need for noisy actions.

3. MADDPG Algorithm

The MADDPG algorithm focuses on one core function: maddpg_training (seen in Figure 1). This function returns a list of average scores for all episodes across both agents, once the model has achieved an average reward score of over 0.5. The hyperparameters used are as follows:

- BUFFER_SIZE = int(1e6): the experience replay buffer size stores a maximum of 1,000,000 experiences.
- BATCH_SIZE = 128: a total of 128 samples are created and stored within the replay buffer at a time.
- GAMMA = 0.99: a discount factor of 0.99 is used to encourage agents to consider future rewards.
- TAU = 1e-3: a tau factor of 0.001 is used within the soft update of the target parameters.
- LR_ACTOR = 1e-3: a learning rate of 0.001 is used to create minor adjustments to the weight parameters within the Adam optimizer, during training, for the actor network.
- LR_CRITIC = 1e-3: a learning rate of 0.001 is used to create minor adjustments to the weight parameters within the Adam optimizer, during training, for the critic network.
- UPDATE_EVERY = 2: at every 2 learning steps the Q-targets are updated to the latest parameters.
- NUM_UPDATE = 10: at every 2 learning steps the Q-targets are updated 10 times.
- NOISE_DECAY = 0.9995: a noise decay rate of 0.9995 is used to slowly reduce both agents exploration rate overtime.

The hyperparameters are utilised within the agent.py file and maddpg.py file. The agent file focuses on the DDPG agent implementation, while the maddpg file extends its functionality.

```
def maddpg_training(brain_name, n_agents, n_episodes=2000,
                    max_t=1000, print_every=100):
    scores_list = []  # list containing scores from each episode
    scores_window = deque(maxlen=print_every) # last set of scores
    scores_avg = []

    # Iterate over each episode
    for i_episode in range(1, n_episodes+1):
        # Reset environment and agents, set initial states
        # and reward scores every episode
        env_info = env.reset(train_mode=True)[brain_name]
        states = env_info.vector_observations
        scores = np.zeros(n_agents)
        maddpg.reset_agents()

        # Iterate over each timestep
        for t in range(max_t):
            # Perform an action for each agent
            actions = maddpg.act(states)

            # Step through the environment using the actions
            env_info = env.step(actions)[brain_name]

            # Set new experiences and interact with the environment
            next_states = env_info.vector_observations
            rewards = env_info.rewards
            dones = env_info.local_done
            maddpg.step(states, actions, rewards, next_states, dones, t)

            # Update states and scores
            states = next_states
            scores += rewards

            # Break loop if an agent finishes the episode
            if any(dones):
                break

        # Save most recent scores
        scores_window.append(np.max(scores))
        scores_list.append(np.max(scores))
        scores_avg.append(np.mean(scores_window))

        # Output episode information
        print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}', end="")
        if i_episode % print_every == 0:
            print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}')

        # Save environment if goal achieved
        if np.mean(scores_window) >= 0.5:
            print(f'\nEnvironment solved in {i_episode} episodes!\tAverage Score: {np.mean(scores_window):.2f}')
            maddpg.save_model()
            break

    # Return reward scores
    return scores_list, scores_avg
```

Fig. 1. MADDPG algorithm code

The algorithm accepts five parameters: the agent brain to use to solve the environment, the number of agents used, the number of episodes, the number of timesteps per episode and the quantity of episodes to print to the console, for algorithm progression updates. The algorithm starts by iterating over each episode, where it resets the environment, initializes the first state, sets all the scores to 0 and resets both agents.

Next, the model training begins by creating each 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 2 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 and the average score is returned and checked for the average agent score goal of 0.5. If the score has been achieved, the models best parameters are saved, and the algorithm finishes with returning a full list of the average scores across both agents over each episode.

4. Results

The results achieved by the trained agents show an average score of +0.5 was achieved in 442 episodes over 100 consecutive episodes, combining both agents scores, which is highlighted in Figure 2.

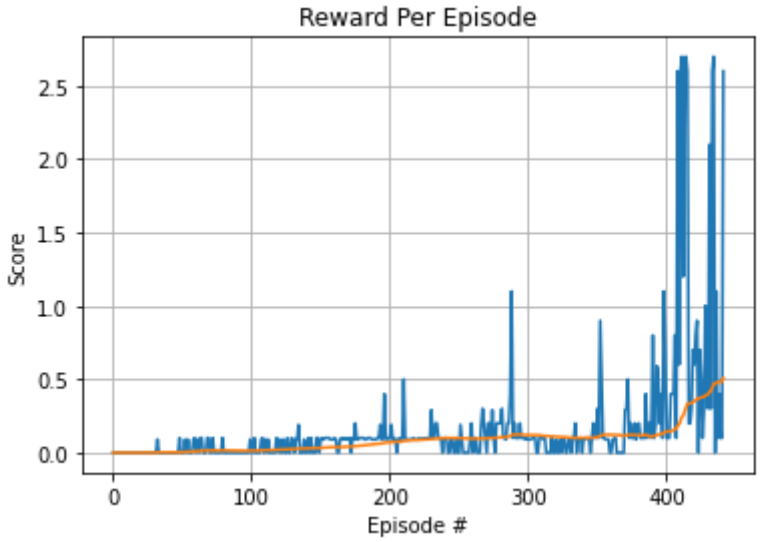

Fig. 2. Plot showing the average reward per episode, across both agents

Looking at Figure 2, the models training steadily increases at a slow pace. This is because the agents are not learning an ordinary task, they are not competing with each other, instead are trying to cooperate with each other to keep the ball in the air for as long as possible. Once the agents are familiar with each other's movements the score achieves peak performance by episode 400. Note that one episode the agents achieved the highest score of ~2.75.

5. Ideas for Future Work

To further improve the performance of the model and learning efficiency of the agent, implementation of a Prioritized Experience Replay Buffer can be utilised. Additionally, a review of other algorithms is advised to solve the challenge and comparing them against the MADDPGs results to determine the best algorithm for the challenge.