1 Environment details

1.1 State and Action spaces

The state space space, for each agent, has 24 dimensions corresponding to the position and velocity of the ball and racket (3 stacked frames of 8). Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping

2 Learning algorithm

Since the action space is continuous and we have two competing agents, we use the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) method for this project.

2.1 Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

We use the DDPG algorithm introduced by Lillicrap et al. (`https://arxiv.org/pdf/1509.02971.pdf`) as the base to implement the MADDPG algorithm. This is a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. This is an extension of Deep Q-learning specialized for continuous tasks. We started with the based DDPG implementation given in the Udacity repository and improved upon that.

2.2 Actor-Critic Methods

Actor-critic methods leverage the strengths of both policy-based and value-based methods. Using a policy-based approach, the **Actor** learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. Meanwhile, employing a value-based approach, the **Critic** learns how to estimate the value (i.e., the future cumulative reward) of different state-action pairs. Actor-critic methods combine these two approaches in order to accelerate the learning process. Actor-critic agents are also more stable than value-based agents, while requiring fewer training samples than policy-based agents. We use two networks, for the Actor and Critic (local and target networks) to improve stability.

This implementation utilizes the decentralized actor with centralized critic approach. The general approach is to employ a single critic that receives as input the actions and state observations from all agents. This makes the training easier and helps centralized training have decentralized execution. Each agent will take the actions based on their own observations of the environment.

2.3 Exploration vs Exploitation

In DQNs we use a ϵ greedy approach to address the exploration vs exploitation dilemma. However, this approach will not work for controlling a tennis racket as the actions are no longer a discrete set of simple directions (e.g., up, down, left, right). If we base our exploration mechanism on random uniform sampling, the direction actions would have a mean of zero, thereby canceling each other out. This can cause the system to oscillate without making much progress.

Therefore, we use the Ornstein-Uhlenbeck process, introduced by Lillicrap et al. The Ornstein-Uhlenbeck process adds a certain amount of noise to the action values at each time step. This noise is correlated to previous noise, and therefore tends to stay in the same direction for a longer duration without canceling itself out. This allows the arm to maintain velocity and explore the action space with more continuity.

The Ornstein-Uhlenbeck process has three hyperparameters that determine the noise characteristics and magnitude:

- mu: the long-running mean
- theta: the speed of mean reversion
- sigma: the volatility parameter

There is an epsilon (ϵ) parameter used to decay the noise level over time. This decay mechanism ensures that more noise is introduced earlier in the training process (i.e., higher exploration), and the noise decreases over time as the agent gains more experience (i.e., higher exploitation). The starting value for epsilon and its decay rate are two hyperparameters that were tuned during experimentation.

The final noise parameters were set as follows:

```
OU_SIGMA = 0.2  # Ornstein-Uhlenbeck noise parameter
OU_THETA = 0.14  # Ornstein-Uhlenbeck noise parameter
EPSILON = 1.0  # explore->exploit noise process added to act step
EPSILON_DECAY = 1e-6  # decay rate for noise process
```

2.4 Learning Interval

Learning at every time step can lead to long training times and instability. Therefore, we set the learning interval to 10 (i.e., learn every 10 time steps). At every learning step, the algorithm randomly samples experiences from the replay buffer and updates the weights 20 times.

2.5 Gradient Clipping

We implement gradient clipping using the torch *clip_grad_norm* function. The function was set to "clip" the norm of the gradients at 1, therefore placing an upper limit on the size of the parameter updates, and preventing them from growing exponentially. This implementation, along with batch normalization, made the learning process more stable.

2.6 Experience Replay

The experience gained by each agent while acting in the environment is saved in a memory buffer, and a small batch of observations from this list is randomly selected and used as the input to train the weights of the DNN (i.e., Experience Replay). In this project, we use one central replay buffer utilized by both agents, therefore allowing agents to learn from each others' experiences.

The replay buffer contains a collection of experience tuples with the state, action, reward, and next state "(s, a, r, s')". Each agent samples from this buffer as part of the learning step. Experiences are sampled randomly, so that the data is uncorrelated. This prevents action values from oscillating or diverging catastrophically, since a naive algorithm could otherwise become biased by correlations between sequential experience tuples.

2.7 Model Architecture

The following model architectures were used for the Actor and the Critic respectively.

2.7.1 Actor

- Input layer: 48 (state size)
- Fully connected hidden layer 1: 256 units, ReLu activation
- Fully connected hidden layer 2: 128 units, ReLu activation
- Output layer: 2 (action size), tanh activation

2.7.2 Critic

- Input layer: 48 (state size)
- Fully connected hidden layer 1: 256 units, ReLu activation
- Fully connected hidden layer 2: 128 units, ReLu activation
- Output layer: 1 (action size), linear activation

2.8 Other hyper-parameters

Following is a list of other parameters used by the DDPG algorithm.

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128  # minibatch size
LR_ACTOR = 1e-03  # learning rate of the actor
LR_CRITIC = 1e-03  # learning rate of the critic
WEIGHT_DECAY = 0.0000  # L2 Weight decay
LEARN_EVERY = 10  # learning timestep interval
LEARN_NUM = 20  # number of learning passes
GAMMA = 0.99  # discount factor
```

3 Results

Figure 1 shows the max score per episode, as well as the rolling average reward for over the last 100 steps for the best MADDPG model. The agent can solve the environment (i.e., get an average score of +0.5 over 100 consecutive episodes) in 1420 episodes.

4 Future Work

A more exhaustive search for hyper-parameters could result in faster convergence of the MADDPG model. It would also be interesting to see how different network architectures affect the agent's behavior. The use of other algorithms other than DDPG may increase the accuracy of the agent. Batch normalization and prioritized experience replay may increase the overall performance. Although there is room for improvement, the current implementation solved the environment well above the required threshold.

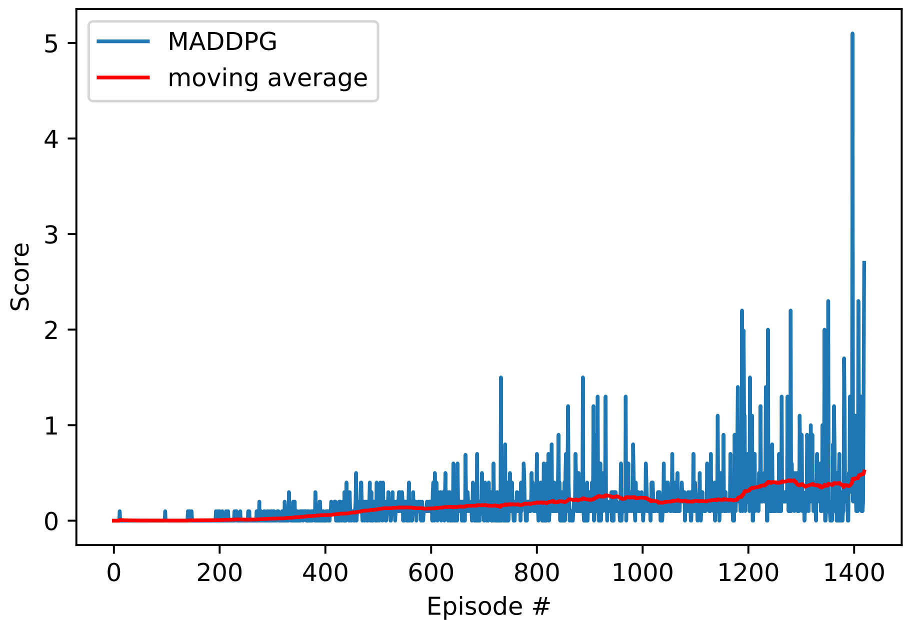

Figure 1: Average score per episode and the moving average.