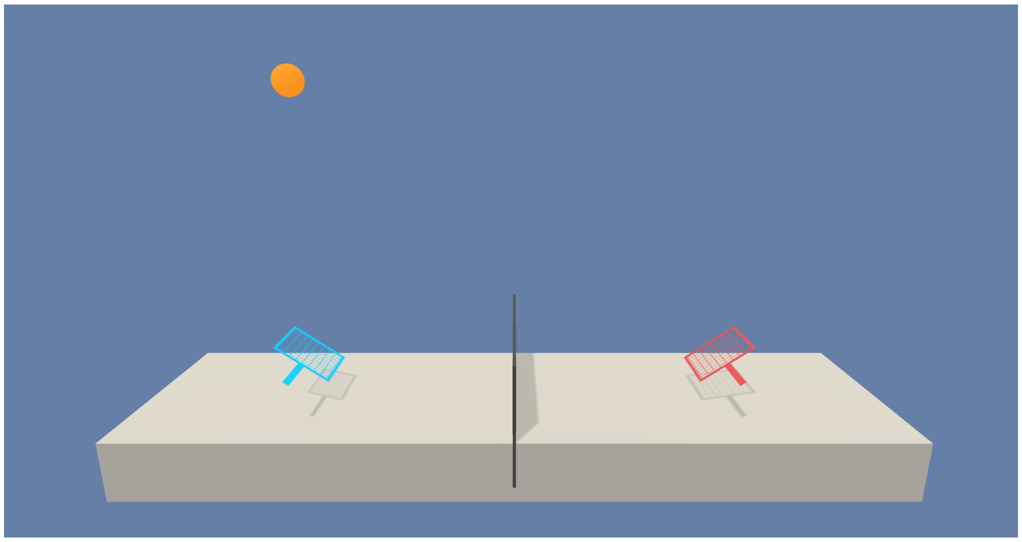


Project 3: Collaboration and Competition

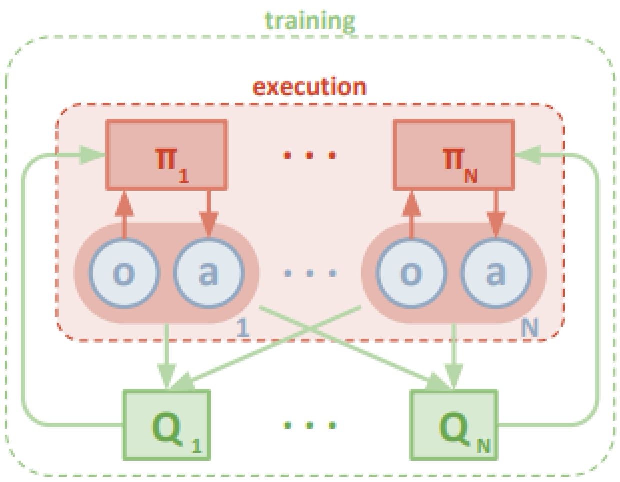
04.27.2021

Project Goals:

In this project, we train two agents to play tennis with one another. Not exactly, but close. We train the two agents to hit a ball with a racket over a net towards each other. We use the Unity ML-Agents Tennis Environment to interact our agents with. This environment provides observations for each agent from that agent's perspective. Our objective is to use these observations for each episode, including their scores, to train the agents and make tennis players out of them. Solving this environment requires training the agents until they can reach an average maximum score of +0.5 after 100 consecutive episodes. The average maximum score means the average over the last 100 episodes where "maximum score" means the maximum score between the two agents in the environment after one episode.

Description of the implementation:

To solve this problem, I use the Multi-agent Deep Deterministic Policy Gradient (MADDPG) algorithm since it operates on two collaborative agents in the continuous action spaces found in this environment.   As the name suggests, MADDPG uses DDPG with a modification for collaborative or competitive agents. Like DDPG, it has an actor model to map the environment states to actions for the current policy, and a critic Q model. The difference is in the critic model where the environment observations and actions from both agents are used to produce the Q targets. The MADDPG algorithm was first introduced by Ryan Lowe, et. Al., as in “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”, is illustrated below.



Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

```
for episode = 1 to $M$ do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length do
    for each agent $i$, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to $N$ do
      Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

          $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$

    end for
    Update target network parameters for each agent $i$:

          $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$

  end for
end for
```

The implementation consists for four parts:

1. A function called maddpg defined in the tennis_t1.ipynb notebook that controls the agent training process and interacts with the environment. The suffix “t1” indicates “trial 1”. It essentially remained unchanged throughout the project since most of the tuning involved the hyperparameters and python module code.
2. The maddpg_agent.py module that contains the multi_Agent class, ddpg Agent class, replay buffer class and OUNoise class definitions and methods.
3. The model_maddpg_1BN.py module that contains the Actor class and Critic class. The Actor class defines a network that is a fully connected multilayer perceptron model used to select an action based on the state of the environment. The Critic

class defines the Critic network and is also a fully connected multilayer perceptron model, but it is used to generate the Q targets based on the states and actions of both agents, as well as Q expected values.

4. The code in the tennis_t1.ipynb notebook calls the maddpg function that runs training until an average "maximum" score has been achieved over the last 100 episodes. In the next cell block, these results are printed out in a chart showing the maximum score, episode by episode, as well as the average maximum score over the previous 100 episodes.

Approach for Solving the Unity Tennis Environment:

I started with the code used for Project 2, Continuous Control that I then modified to get to the final code for the Collaboration and Competition project. My Jupyter notebook from Continuous Control is essentially the same as for this project with a few changes in function name and adaptation for the new goal of average "maximum" agent scores. The major differences with Project 3 using MADDPG are:

1) Store both agent's environment states, actions, rewards, next states and dones for the same episode instead of a single agent's data.

2) Applying both agent's actions and states to the Critic models.

All the hyperparameters used for DDPG and Continuous Control were carried over to MADDPG. However, several hyperparameters were not used such as noise decay, update frequency and update number.

Hyperparameters:

The following hyperparameters are provided, but several are set such that they have no effect:

Hyperparameters used in the Jupyter Notebook: tennis_t1.ipynb

Epsilon (eps): Not the epsilon of the epsilon greedy policy in DQN, but used in a similar manner in the algorithm to reduce the probability of exploration as the networks train (noise decay). Use of noise decay did not improve training so it was not used in the final successful run.

eps_start: starting value of epsilon

eps_end: ending value of epsilon

eps_decay: factor to multiply epsilon by after each episode

The maddpg() function tests if a random number between 0 and 1 is greater than eps. This result is passed to the act() method as a flag where if it is True, then noise is not added to the action and if False, then noise is added to the action. Since starting eps at 1 guarantees that the result is False, the first action of the Actor will have added Ornstein–Uhlenbeck process noise, where each successive episode will reduce epsilon by a factor of eps_decay increasing the likelihood of not applying noise. Thus exploration declines as the networks train, but does not go to zero because a lower bound of eps_end maintains a small probability of exploration. In the final run, eps_decay was set to 1.0 to prevent any noise decay from occurring.

PRINT_EVERY: Sets the length of the moving average used to determine whether the environment has been solved. It is also used to indicate how often to print the moving average: prints every 1 in PRINT_EVERY episodes.

max_t: During an episode, if no Done has been received from the environment after max_t timesteps, then the episode is terminated.

n_episodes: The number of episodes run by maddpg() function.

Hyperparameters for the multi_Agent Class in the module: maddpg_agent.py

BUFFER_SIZE: the depth of the replay buffer

The replay buffer is defined as a python deque (double ended queue)

BATCH_SIZE: The number of experiences in a sample - A.K.A the mini-batch size

GAMMA: The discount factor as shown in DDPG algorithm

TAU: Defines the proportions of the local and target QNetwork parameters to apply to the target QNetwork's parameter update. This is performed by the soft_update() method of the ddpg Agent class.

LR_ACTOR: Learning Rate used by the Optimizer for the Actor

LR_CRITIC: Learning Rate used by the Optimizer for the Critic

WEIGHT_DECAY: Adams optimizer weight decay for the Critic network - not used, set to 0

UPDATE_FREQ: Update the models every "one in UPDATE_FREQ" calls of the multi_Agent step() method (one in every UPDATE_FREQ trial of every episode).

UPDATE_NUM: Number of updates to perform to the models for every call of the multi_Agent step() method.

Hyperparameters for the Actor and Critic Classes in model_ddpg_1BN.py

fc1_units: The size of the first hidden layer in the Actor network model.

fcs1_units: The size of the first hidden layer in the Critic network model

fc2_units: The size of the second hidden layer in both the Actor and Critic network models

Description of functions in the Jupyter Notebook:

maddpg() function

maddpg() is defined in tennis.ipynb Jupyter notebook. It takes additional arguments: n_episodes, max_t, eps_start, eps_end and eps_decay.

maddpg() has the arguments:

multi_agent: Is the instantiated multi_Agent object.

env: Is the Unity tennis environment.

The remaining arguments: n_episodes, max_t and print_every are defined above.

maddpg() steps through each of n_episodes for a maximum of max_t timesteps. It keeps a 100 episode moving average of reward totals that includes the average score across all the agents, prints those totals to the screen, and breaks when the moving average reaches +0.5. It returns a list of the maximum score of the two agents. It also returns a list of the moving average computed for each episode. These two lists are used to plot the results.

Description of the multi_Agent Class:

maddpg_agent.py

This python module contains the class definitions for multi_Agent, Agent, ReplayBuffer and OUNoise. The multi_Agent Class has the step(), act() and reset() methods. It also instantiates the two Agent Class objects, one ReplayBuffer Class object and one OUNoise object. The Agent Class is the same Agent Class used for DDPG except the learn() method was modified to perform multi-agent updates. The ReplayBuffer Class was also borrowed from the DDPG code from Project 2, but modified to add experiences from both agents for each call of the add() method. The OUNoise Class is identical to the one used in Project 2.

The multi_Agent step() method does two actions:

- Adds an experience to the replay buffer for both agents.
- Calls the learn() method for both DDPG agents of the Agent Class **UPDATE_NUM** times every one in **UPDATE_FREQ** time steps.

The ReplayBuffer Class from the ddpg used for Project 2 to accommodate two agents. The Unity Tennis environment provides the states, next_states, actions and dones for both agents. However, the ReplayBuffer Class from the ddpg project is designed to handle experiences for one agent at a time. To adapt the ReplayBuffer Class for two agents, either the data from both agents must be combined as though it was one agent, or additional fields must be added to the ReplayBuffer Class to accommodate both agents. In the former case, the data from both agents must be separated for training after sampled from memory. In the latter case, no separation is required; however, data from both agents must be concatenated to be applied to the critic model. I chose to use the latter method, so the step() method must separate out the data for each agent before adding it to memory. The multi_Agent step() method then calls the ddpg Agent learn() method for each agent using a different random sample from the ReplayBuffer Class' memory.

The ddpg Agent learn() method performs a training step for the actor and critic networks. It also calls ddpg Agent soft_update() method to copy a portion of the local network parameters to the target networks parameters.

The multi_Agent act() method gets an action from the local actor network of each agent with the current environment state for each agent as the input. The environment is expecting a list containing a list from each agent that includes it's actions. If the add_noise argument is True, then a noise sample from the OUNoise class object noise is added to the actions returned from the local actor network. The noise shifts the actions to explore the environment and potentially actions yielding more consistent rewards.

The OUNoise class provides samples of Ornstein-Uhlenbeck noise. These noise samples differ from random noise samples in that they are a running sum of noise with mean 0.

model_maddpg_1BN.py

Contains the network class, reset() and forward() methods for the Actor and Critic networks. The Actor class takes the state size, action size and hidden size (**fc1** and **fc2**) to create a fully connected network. Input consists of the state and output is the action the agent should take. The Critic class also takes the state size and action size, and hidden size (**fcs1** and **fc2**), but the Critic network takes in the action as input, along with the state, from both agents to generate the q-values. For the Actor class, the forward() method takes the state as input and moves it through the network, returning the action tensor. The Critic class concatenates in the actions with the states to generate q-values. Both networks use one batch normalization layer.

<u>Chosen hyperparameters</u>:

eps_start: 1.0

eps_end: 0.01

eps_decay: 1.0 (noise decay disabled)

PRINT_EVERY: 100

max_t: 5000

n_episodes: 2500

Hyperparameters for the multi_Agent Class in the module maddpg_agent.py

BUFFER_SIZE: 100000

BATCH_SIZE: 256

GAMMA: 0.99

WEIGHT_DECAY: 0

TAU: 0.001

LR_ACTOR: 0.0002

LR_CRITIC: 0.001

UPDATE_FREQ: 1 (update every time step)

UPDATE_NUM: 1 (update both agents every time step)

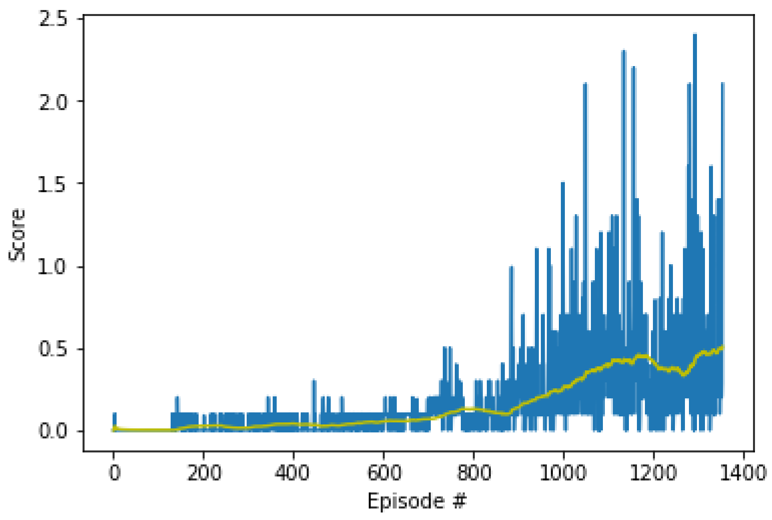
Hyperparameters for the Actor and Critic Classes in model_ddpg_.py

I used the same methodology to pick hidden units that I did for Project 2: For the first hidden units, I chose 8x the input size (8 x 24 = 192). For the second hidden units, I chose 6x the input size (6 x 24 = 144).

fc1_units: 192, **fcs1_units**: 192. **fc2_units**: 144.

Plot of the Maximum Rewards attained by the two agents each episode:

The plot of the maximum rewards between the two agents each episode (blue) and the average over 100 episodes (yellow) versus episode number using the chosen hyperparameters and the notebook tennis_t1.ipynb is shown below:



Episode 100	Average Score: 0.0010
Episode 200	Average Score: 0.0255
Episode 300	Average Score: 0.0149
Episode 400	Average Score: 0.0412
Episode 500	Average Score: 0.0276

```
Episode 600      Average Score: 0.0538
Episode 700      Average Score: 0.0676
Episode 800      Average Score: 0.1290
Episode 900      Average Score: 0.1375
Episode 1000     Average Score: 0.2650
Episode 1100     Average Score: 0.3852
Episode 1200     Average Score: 0.4175
Episode 1300     Average Score: 0.4552
Episode 1353     Average Score: 0.5060
Environment solved in 1353 episodes!     Average Score: 0.5060
```

Discoveries from trying different sets of hyperparameters:

Noise decay: Although the printed output appears to show a monotonically increasing average maximum reward, it was quite erratic on an episode to episode basis. The average maximum score for the last 100 episodes was printed out each episode so that I could observe the progress. The closer the score got to +0.5, the more erratic the scores got. For Project 2, adding noise decay improved training. I expected adding noise decay would take out some of the fluctuations like in Project 2, but it actually increased it resulting in the agents never reaching the goal score. I did not find anything in the papers or other literature that addressed this behaviour. I can only conclude that for this environment, the highest possible exploration continues to be essential in order to continue learning and achieve higher scores with a model trained capable of scores above +0.4.

Number of hidden layers: Using much smaller size hidden layers than the chosen number of layers (fc1 = fcs1= 48, fc2 = 32) resulted in very slow learning. I suspect the agents would train eventually to achieve the goal, but I cut it off at 2500 episodes with an average maximum score of 0.14.

Batch Normalization layers: I tried 0, 1 and 2 batch normalization layers. For DDPG Continuous Control in Project 2, I discovered that one batch normalization layer worked best for one agent and 2 batch normalization layers worked best when training 20 agents simultaneously. I found similar results for tennis where we are training two agents to work collaboratively rather than independently worked best with one batch normalization layer. Agents with models of 0 or 2 batch normalization layers failed to reach the goal +0.5 average maximum score after 2500 episodes.

Actor Learning Rate: The successful Actor Learning Rate I ended with was 0.0002. The goal was actually achieved with an Actor Learning Rate of 0.0001, but it took nearly 2000 episodes, so I wanted to see if it could train faster. Here is the output of that run:

```
Episode 100      Average Score: 0.0029
Episode 200      Average Score: 0.0000
Episode 300      Average Score: 0.0110
Episode 400       Average Score: 0.0059
Episode 500       Average Score: 0.0170
Episode 600       Average Score: 0.0390
Episode 700       Average Score: 0.0317
Episode 800       Average Score: 0.0231
Episode 900       Average Score: 0.0429
Episode 1000      Average Score: 0.0660
Episode 1100      Average Score: 0.1026
Episode 1200      Average Score: 0.1212
Episode 1300      Average Score: 0.1289
Episode 1400      Average Score: 0.2066
Episode 1500      Average Score: 0.2188
Episode 1600      Average Score: 0.3412
Episode 1700      Average Score: 0.3611
Episode 1800      Average Score: 0.4598
Episode 1900      Average Score: 0.4579
Episode 1947      Average Score: 0.5045
Environment solved in 1947 episodes!        Average Score: 0.5045
```

I tried an Actor Learning rate of 0.001, but that resulted in the scores becoming very unstable and dropping to zero after episode 1400. Instability is usually a sign of too high a learning rate, so instead I tried a much smaller increase to 0.0002 and that yielded the desired result.

Ideas for Future Work:

1) *Redesign the multi_Agent and Agent python classes to accept an arbitrary number of agents*. Since there were only two agents in the environment, I did not try to write the code to accept an arbitrary number of agents. This enabled me to explicitly create additional fields in the ReplayBuffer memory for the second agent. By having both agents defined in memory without packing their data, I could simply concatenate their states and actions to create the inputs to the critic models. For future work, the code could be written to accept an arbitrary number of agents by packing their environment data in the ReplayBuffer memory. However, that would require more complex unpacking and reshaping of the batch samples for training.
2) *Different hyperparameter settings*: In the interest of time, I did not try a lot of different hyperparameter settings. The paper "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments", had different values for GAMMA and TAU. I did not try adjusting these to see their effects. They also tried different random seeds as well.
3) *Different settings for the frequency and number of updates*: I have this feature in my code, but I did not use it. The paper "Multi-Agent Actor-Critic for Mixed

Cooperative-Competitive Environments", in the appendix it mentions updating only once after every 100 samples added to the replay buffer. I had only used updating after every sample added to the replay buffer.