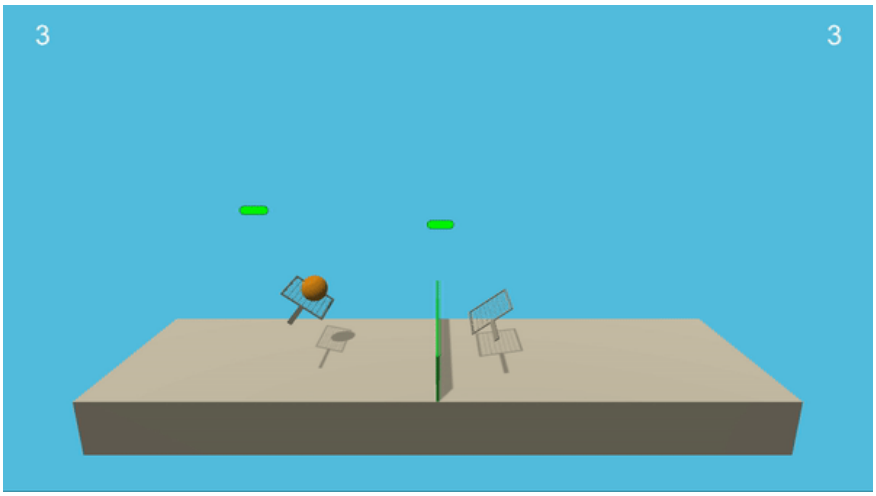
Project Report
DRLND Project 3:
Collaboration and Competition

Problem Description

For this project, we work with the Tennis environment.



The problem is framed as a rich game environment on the Unity platform. The environment has been designed by the Udacity Deep RL Nanodegree (DRLND) program team, and is provided as part of the starting materials. Interacting with the environment is through the use of a Python API exposed by the Unity Machine Learning Agents (ML-Agents) Toolkit.

In this environment, two agents 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.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. The observation is stacked in a vector of size 3, representing 3 consecutive frames of state information. Further, each agent receives its own, local observation, so the whole state given by the environment is [2, 24]. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, our agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically:

- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
- This yields a single **score** for each episode.

The environment is considered solved, when the average (over 100 episodes) of those **scores** is at least +0.5.

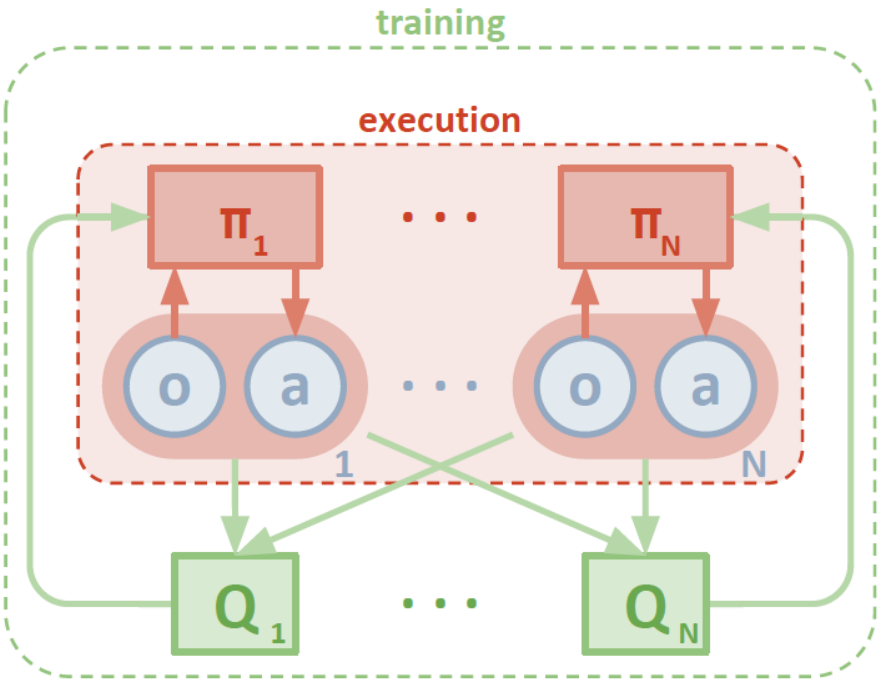
Learning Algorithm

To train our Tennis agents to use their rackets to bounce a ball over the net and achieve high rewards, we resort to Multi-Agent Reinforcement Learning (MARL) using an Actor-Critic Method. We follow closely the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) solution used by Lowe et al. [1], given the need to train multiple agents in parallel, and the Continuous nature of the action space used by the environment.

Multi-Agent Actor Critic

From the paper [1]:

“We allow the policies to use extra information to ease training, so long as this information is not used at test time. ... We use a simple extension of actor-critic policy gradient methods where the critic is augmented with extra information about the policies of other agents, while the actor only has access to local information.”



Overview of multi-agent decentralized actor, centralized critic approach

Twin Delayed Deep Deterministic Policy

Instead of the Deep Deterministic Policy Gradient (DDPG) method for learning, we use the Twin Delayed Deep Deterministic Policy (TD3) [2] method in our solution.

TD3 is widely considered a replacement successor to DDPG, and utilizes the following techniques to improve upon the success of DDPG:

- **Clipped Double Q-Learning for Actor-Critic**: Use Double Q-Learning to alleviate the overestimation bias problem in Actor-Critic. Specifically, use two pairs of critic networks (2 local and 2 target). While calculating target Q values, use both critic networks to compute and choose the smaller outcome as target Q value to update the network. This reduces the impact of overestimation.
- **Target Networks and Delayed Policy Updates**: Just like DDPG, use target networks and a soft update method to stabilize the training procedure. Aside from these methods, also use delayed updates rather than updating target and actor networks on every timestep. This results in higher quality policy updates, thereby improving the performance of the agent.
- **Target Policy Smoothing Regularization**: Intuitively, similar actions should have similar values. To implement this idea, add a small amount of random noise to the target policy. That is to say, fitting the value of a small area around the target action would have the benefit of smoothing the value estimate by similar state-action value estimates.

The full algorithm for training the TD3-network is as follows:

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Model Architecture

- The neural network uses the rectified non-linearity for all layers.
- The final output layer of the actor is a *tanh* layer, to bound the actions.
- The networks have 2 hidden layers with 256 and 128 units respectively.
- Actions are not included until the 2nd hidden layer of *Q*.
- The final layer weights and biases of both the actor and critic are initialized from a uniform distribution [-0.003, 0.003]. This is to ensure the initial outputs for the policy and value estimates are near zero.
- We train with minibatch sizes of 256.
- We use a replay buffer size of 10^5.
- Each agent uses its own actor network
- Agents share the critic networks
- Agents share the replay buffer.

```
Actor(
    (fc1): Linear(in_features=24, out_features=256, bias=True)
    (fc2): Linear(in_features=256, out_features=128, bias=True)
    (fc3): Linear(in_features=128, out_features=2, bias=True)
)

Critic(
    (fc1): Linear(in_features=48, out_features=256, bias=True)
    (fc2): Linear(in_features=260, out_features=128, bias=True)  ⇐ actions added
    (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```

Training Results

The implementation for solving this project used Python 3 and PyTorch. The code is shared on GitHub: https://github.com/aquanta-aagyaa/drlnd-p3-collab-compet

The important files are:

1. Tennis.ipynb - Jupyter notebook to walk through the training process
2. matd3.py - The MATD3 agent implementation
3. td3.py - The TD3agent implementation
4. model.py - implementation of the Actor and Critic (DNN) classes using PyTorch
5. buffer.py - The replay buffer implementation
6. OUNoise.py - The OUNoise implementation

Hyperparameters

The following hyperparameters allow the model to achieve the desired goal, of a 2-agent average score of +0.5 over the last 100 episodes.

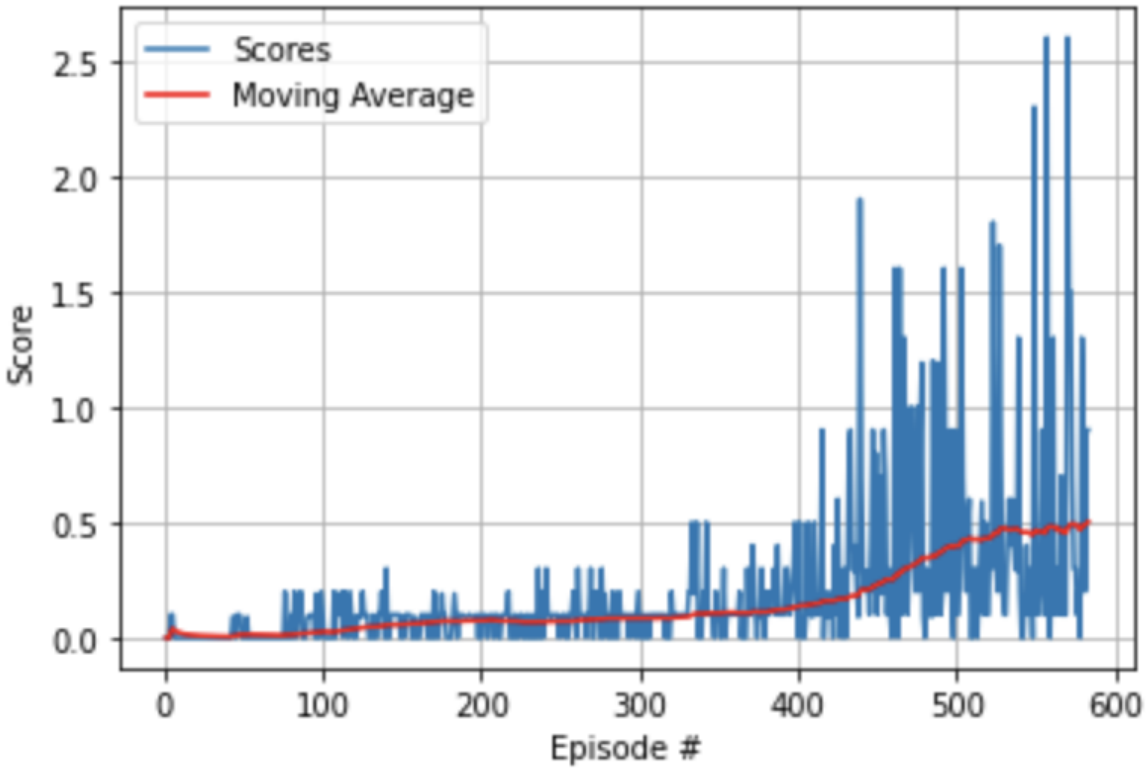
- BUFFER_SIZE: int(1e5) # replay buffer size
- BATCH_SIZE: 256 # minibatch size
- GAMMA: 0.99 # discount factor
- TAU: 1e-2 # for soft update of target parameters
- LR_ACTOR: 1e-3 # learning rate of the actor
- LR_CRITIC: 1e-3 # learning rate of the critic

TD3-specific:

- td3_noise: 0.2 # noise to add to actions
- td3_noise_clip: 0.5 # extent of noise
- td3_delay: 2 # how often to update policy

Successful training session:

```
Episode 100     Score: 0.1000   Average Score: 0.0261
Episode 200     Score: 0.0000   Average Score: 0.0723
Episode 300     Score: 0.0000   Average Score: 0.0843
Episode 400     Score: 0.0000   Average Score: 0.1347
Episode 500     Score: 0.1000   Average Score: 0.4005
Episode 583     Score: 0.9000   Average Score: 0.5015
Environment solved in 583 episodes!     Average Score: 0.5015
```



The model weights of the successful agent are saved to the ***models*** folder.

Ideas for Future Work

Some ideas we can pursue to improve the performance of our agent:

Prioritized Experience Replay
The current implementation samples uniformly from the replay buffer. We might want to sample more frequently those transitions from which there is much to learn.

Parallel Processing
We might want to try solving the learning problem using algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

References

1. Lowe, Ryan, et al. “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”. https://papers.nips.cc/paper/2017/file/68a9750337a418a86fe06c1991a1d64c-Paper.pdf
2. Fujimoto, Scott, et al. “Addressing Function Approximation Error in Actor-Critic Methods”. https://arxiv.org/abs/1802.09477