Multi-Agent MADDPG RL Agents

Project Tennis

As part of Udacity Deep Reinforcement Learning Nanodegree

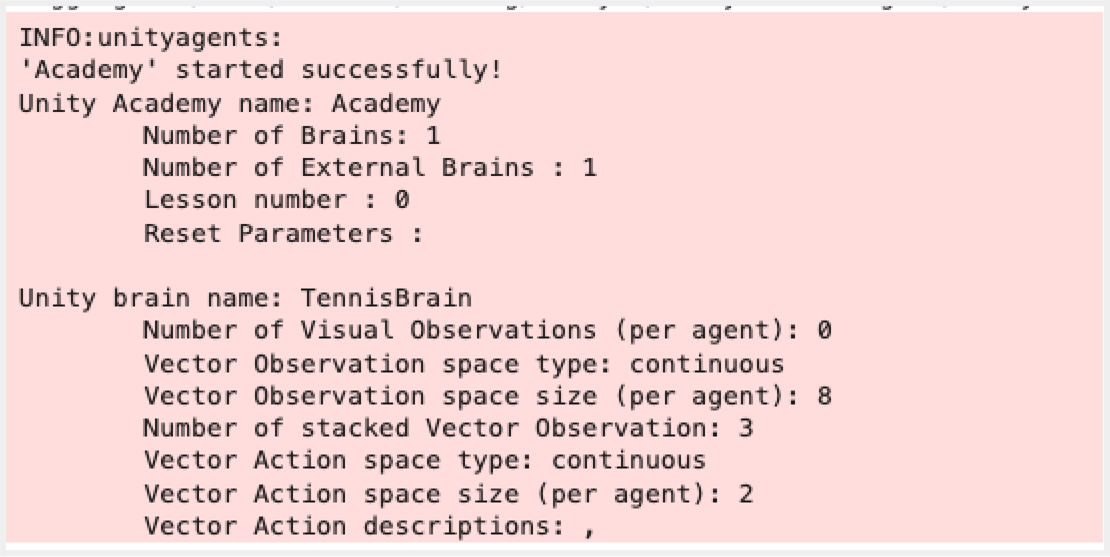
Sourav Babi Seal (sourav.seal@gmail.com)
July 2025

Summary: Train two agents to control rackets to bounce a tennis ball over a net and keep the ball in play using Multi-Agent Deep Deterministic Policy Gradient Reinforcement Learning. The environment provided by Unity is provided by Unity Machine Learning agents.

Problem Statement

- Traditional Q-learning methods for multiple agents are challenged by inherent non-stationarity of the environment from the perspective of any individual agent (in a way that is not explainable by changes in the agent's own policy). This causes stability challenges and prevents use of past experience replay.
- Policy gradient methods exhibit high variance when coordination of multiplet agents is required.

Environment



```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :

Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        Vector Action descriptions: ,
```

State Space

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. The observation vector for each agent corresponds to a vector of size 24 potentially corresponding to 3 stacked observations to express temporal and spatial directional aspects.

```
states.shape = env_info.vector_observations
states.shape = (2, 24)
```

Action Space

Two continuous actions for each agent, corresponding to movement toward (or away from) the net, and jumping.

Rewards

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

To solve this environment we must get an average score of +0.5 over 100 consecutive episodes.

Approach

The paper Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments implements a general-purpose algorithm that implements centralized training with decentralized execution. They use a simple extension of the Actor-Critic policy gradient methods where each agent's critic is augmented with extra information about the policies of other agents, but the actor only has access to local information. The centralized critic function explicitly uses the decision-making policies of the other agents, so that they can learn approximate models of other agents and use them in their own policy learning procedure.

Actor and Critic Model Architecture

Employs Deep Neural Network to approximate the Actor Policy function and the Critic Q-value function for each of the local and target networks for each of the agents.

Actor

It has 3 fully connected (FC) layers

- FC 1 dimensions: input state size 24, and the first hidden dim 256
- FC 2 dimensions: input first hidden dim 256, and second hidden dim 256
- FC 3 dimensions: input second hidden dim 256, and the output action dimension 2

FC 1 and FC2 have ReLU activation functions. Apply BatchNorm1d after activation function (leaky ReLU) on FC1. Since the outputs are continuous values a tanh function is used to keep the values between -1 and 1.

BatchNorm normalizes the inputs to have a zero mean and unit variance which reduces internal covariate shift, makes learning smoother and faster, stabilizes training dynamics and prevents divergence in actor outputs. We use Leaky ReLU to address the dying ReLU problem where neurons can become inactive and stop learning. We still allow a small non-zero gradient for negative inputs so that neurons can still contribute to learning.

As the dimensions indicate, the actors only store the action and state size for each agent.

Critic

Each agent's critic on the other hand has access to all the state observations and the actions of all agents. The output is a single Q-value approximation

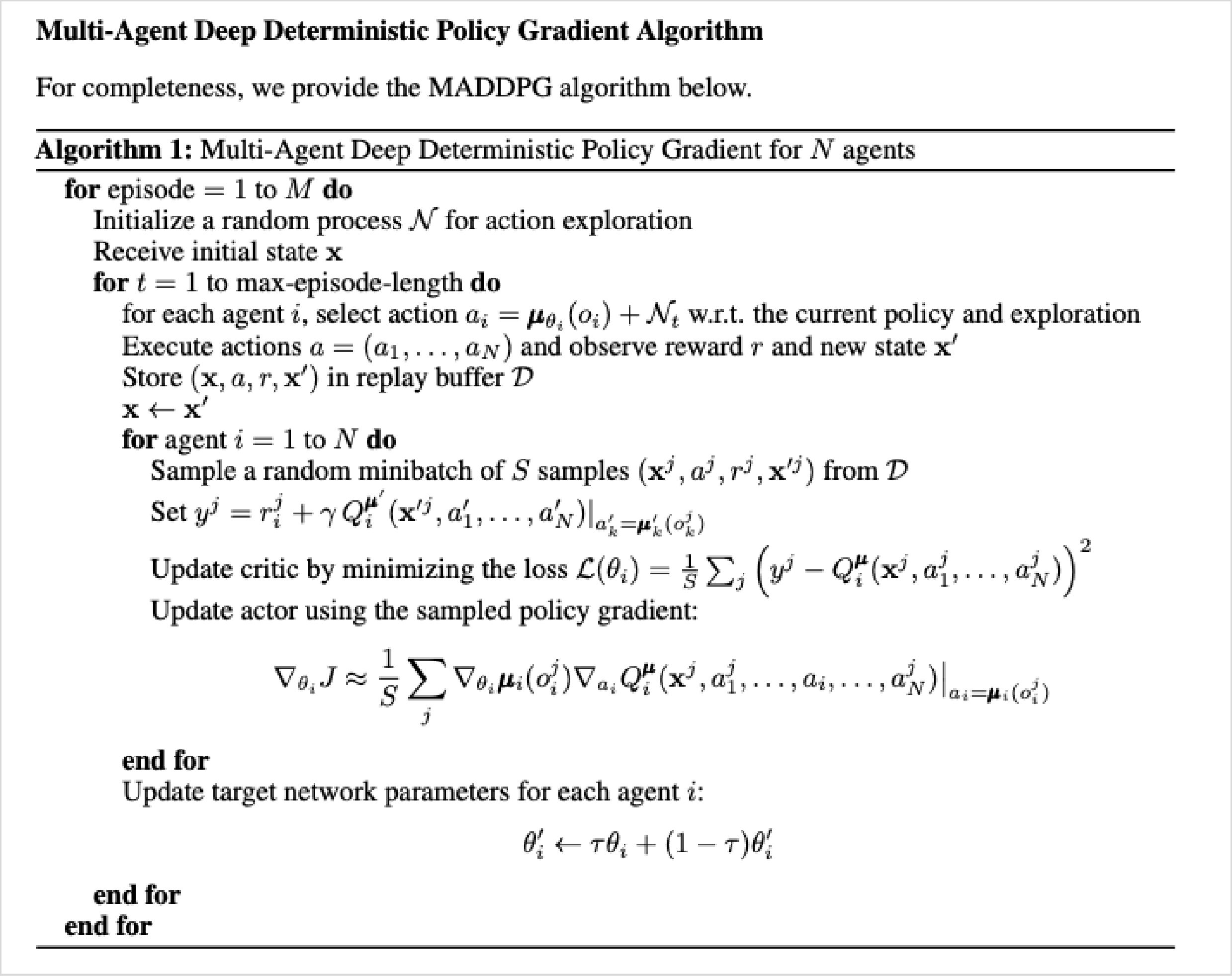
It has 3 fully connected (FC) layers

- FC 1 dimensions: input (total state size (state_size*num_agents) + total action size (action_size * num_agents), and the first hidden dim 256
- FC 2 dimensions: input first hidden dim 256, and the output second hidden dim 256
- FC 3 dimensions: input second hidden dim 256, and the output 1

FC 1 and FC2 have Leaky ReLU activation functions. Apply BatchNorm1d after activation function (leaky ReLU) on FC1.

Training algorithm for MADDPG

We implement the following training algorithm as outlined in the paper:



Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm 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
```

Changes to ReplayBuffer Implementation from DPDG implementation

Note now the Experience (s, a, r, s’, done) stores the information across all agents. This means that after `ReplayBuffer.sample()` each mini batch the following information:

Variable	**Shape**
`states`	`(B, N, S)`
`actions`	`(B, N, A)`
`rewards`	`(B, N) - one per agent`
`next_states`	`(B, N, S)`
`dones`	`(B, N)`

where

Variable	Shape
`B`	Batch size
`N`	Number of Agents
`S`	State Dimension per agent (24)
`A`	Action Dimension per agent (2)

Our approach is to keep the data in the "native" MADDPG shape `(batch, agent, -)` all the way through and avoid reshaping every update. This is why we use `np.stack` which adds a new leading dimension instead of flattening.

Changes to DDPGAgent implementation

`DDPGAgent.learn()`

For the agent actor we focus on extracting the slice for agent i , only that agent's observations.

Update Critic

Setting up the Critic loss: (MSE (Q_i(s,a), y_i)
For the agent critic, we concatenate along the state dimension so that (B, N, S) flattens to (B, NxS) using `.view(B, -1)`. Similar actions are used for flattening the `next_actions`.

Y_i (from the paper, _targets) for agent i

- `rewards[:,i].unsqueeze(1)` makes it (B, 1) so that it can broadcast across the batch.
- Similarly for done_i including casting it to a float.

Update Actor

Setting up the Actor loss: -E[Q_i(s, μ_i(o_i), a_{-i})]

We build the joint action where only this agent's action is differentiable. Others are `.detach()` to avoid back-prop into their policies.

`DDPGAgent.act()`

Added `.unsqueeze(0)` so that the every single-state forward pass looks like a batch of size of 1 so as to keep BatchNorm happy.

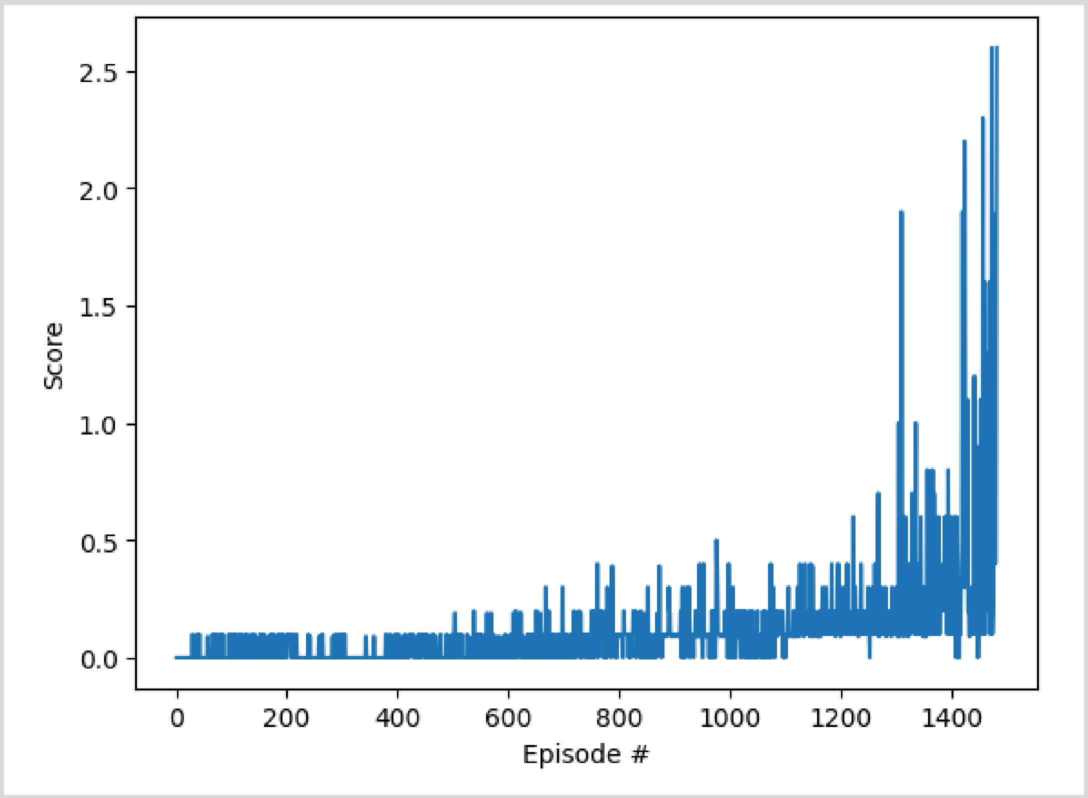
Hyperparameters

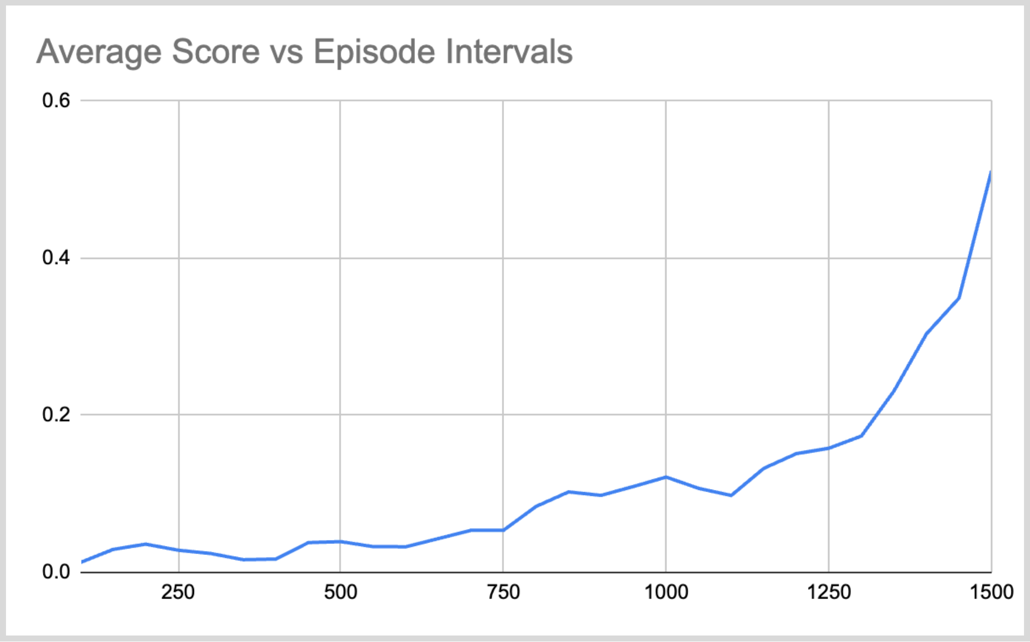
Hyperparameter	Value
Buffer Sizer	1e6 (from the paper)
Batch Size	256
Discount Factor (GAMMA)	0.99
Learning Rate Actor (LR_ACTOR)	1e-4
Learning Rate Critic (LR_CRITIC)	1e-3
Noise Weight Start	0.5
Learn Every	2 steps (after memory batch size)
Weight Decay	0

Training Parameters	Value
Max Episodes	6000
Max time step of an episode	1000

Results

We solved the environment in 1382 episodes, sustained an average of 0.5+ score over the past 100 episodes.


2.5
2.0
1.5
1.0
0.5
0.0
Score
0
200
400
600
800
1000
1200
1400
Episode #



Future Directions

There are multiple model options and hyperparameters that are possible with this MADDPG implementation. While training we experienced multiple times when training collapsed. We would like to do a hyper-parameter sweep to see if we can get quicker convergence in training than currently achieved.
This exercise covered just the Tennis environment - extending this to a Soccer environment

References

Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments, by Ryan Lowe, Yi Wu, Amir Tamar, Jean Harb, Peter Abbeel, Igor Mordatch