TENNIS ENVIRONMENT

In the Tennis 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. Each agent receives its own, local observation. 2 continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

- 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

The task is episodic, and in order to solve the environment, 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.

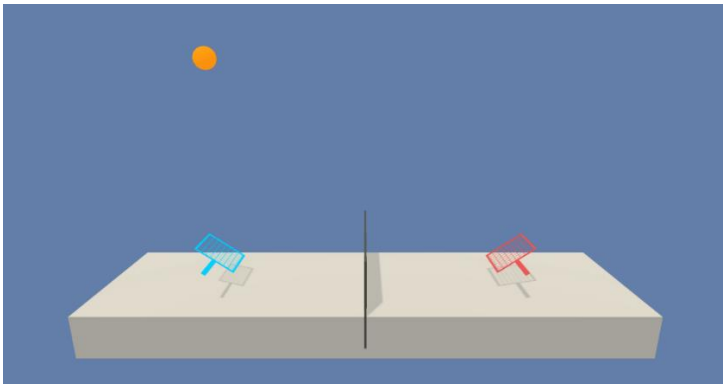


Figure 1: Unity ML-Agents Tennis Environment

DEEP DETERMINISTIC POLICY GRADIENTS FOR MULTI-AGENT ENVIRONMENTS

Deep deterministic policy gradients (DDPG) is a combination of deterministic policy gradients with ideas from deep Q-learning. It is under the umbrella of actor-critic algorithms. DDPG uses a critic network that learns a Q-function and an actor network that learns a policy. The critic can be thought of generalization of the DQN to continuous action spaces. DQN estimates the Q function and computes the max over actions, $max_a = Q(s,a)$, to select a policy. Computing max over actions for a continuous action space, requires iterative optimization and is expensive. Therefore, critic approximates it by $max_a Q(s,a) \approx Q(s,\mu(s))$. The loss function for the critic is

mean-squared Bellman error (MSBE), same as DQN. The actor network then performs policy gradient using $Q(s, \mu(s))$ as a loss function (step 14 in Figure 2).

DDPG Algorithm:

DDPG implementation is described here in detail. The pseudo code of the algorithm is below and follows the **OpenAI baseline**.

Deep Deterministic Policy Gradient

1: Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: **repeat**
4: Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** however many updates **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

13: Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

14: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

15: Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

16: **end for**
17: **end if**
18: **until** convergence

Figure 2: Pseudocode of DDPG

To summarize the key components:

- ***Replay buffer:*** State transitions and corresponding rewards are saved in a buffer and sampled randomly in batches during training (step 11). Therefore, DDPG is an off-policy method. Since MSBE is used as a loss function and the off-policy samples can be used.
- ***Soft parameter update***: Just like DQN, DDPG uses target networks for both the actor and the critic to decorrelate target and predicted Q values, which in turn improves the stability of learning algorithm and convergence. The online network weights are slowly blended into the target network (step 15)
- ***Exploration****:* DDPG uses a deterministic policy which can hinder exploration. In discrete action spaces, exploration is encouraged by using an epsilon-greedy policy. In DDPG, Ornstein-Uhlenbeck noise is added during training to encourage exploration over a continuous action space (step 4). The state value is also clipped to the min and max allowable range. In the OpenAI implementation, the agent also takes uniformly sampled random actions for a fixed number of steps at the beginning before starting training.

Extension to Multi-Agent Systems:

There are several challenges in applying Q-learning and policy gradient methods to multi-agent settings. As the paper below summarizes, in a multi-agent setting, since all agents take actions, the environment is non-stationary from the perspective of an individual agent. This prevents the use of experience replay which is crucial to use for stability of Q-learning. Policy gradient methods experience high variance when multiple agents are used. The paper by Lowe *et. al* presents a framework that uses centralized training and decentralized execution.

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, O. P., & Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems* (pp. 6379-6390).

The critic is augmented with extra information about the policies of other agents, while the actor only has access to local information. After training is completed, only the local actors are used at execution phase, acting in a decentralized manner.

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 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 x'
    Store $(\mathrm{x}, a, r, \mathrm{x}')$ in replay buffer $\mathcal{D}$
    $\mathrm{x} \leftarrow \mathrm{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathrm{x}^j, a^j, r^j, \mathrm{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathrm{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathrm{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

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

    end for
    Update target network parameters for each agent i:

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

  end for
end for

Figure 3: Pseudocode of MDDPG

IMPLEMENTATION

Figure 4 show the architecture for multi-agent DDPG (MADDPG). The critic of each agent is a centralized action-value function and uses the actions and observations (states) of all agents. Since the critic has access to actions taken by all agents, the environment is stationary even as the policies change. The experience of all agents are added in the replay buffer such that the input to the critic is : (states*number of agents, actions*number of agents)=52 and the output is the mean action-value (same as DDPG). **There are two main pitfalls during**

implementation. First, if you use the ReplayBuffer class from DDPG project, you need to keep track of which state and action pair belongs to which agent during training. You can either decode observations from OpenAI environment as they come and put them in different ReplayBuffers or you can use one ReplayBuffer but decode the observations during training. I did the latter. I used encode and decode functions provided in fdalisva59@guthub to add data to and grab data from the replay buffer. Most implementations I came across got this part wrong. They essentially create two instances of DDPG, stack each agent's experience as if they are coming from the same source. They essentially create two copies of the same critic since they treat agent states/actions as if they are coming from identical sources. Second, note that during training (backprop step) of an agent's actor network, the other agent's actions should be detached from the network. Otherwise you will observe crazy oscillations during training and your network may fail rather than converging to optimal score. The schematic in the paper looks trivial, but the devil is in the detail.

Architecture*:*

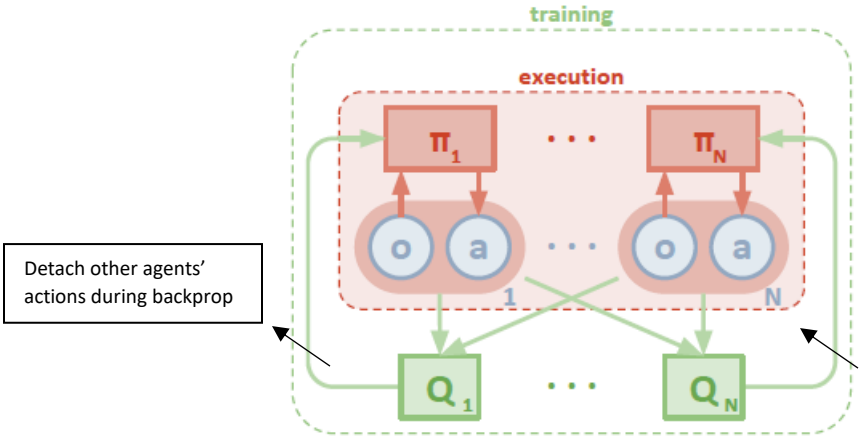


Figure 4: MADDPG architecture

Hyper-parameters:

```
#Parameters from OpenAI Baselines
BUFFER_SIZE = int(1e6)    # replay buffer size
BATCH_SIZE = 256          # minibatch size
UPDATE_EVERY = 3          # Number of episodes that should elapse between gradient descent updates
GAMMA = 0.995             # discount factor
TAU = 1e-3                # for soft update of target parameters
LR_ACTOR = 1e-4           # learning rate of the actor
LR_CRITIC =5e-4           # learning rate of the critic
WEIGHT_DECAY = 0          # L2 weight decay
EPSILON = 1.0             # Exploration noise coefficient
EPSILON_DECAY = 0         # Decay rate for exploration noise
LEARN_TIMES= 2            # Number of times to backprop with the batch
WARM_UP=    0             # Number of steps for uniform-random action selection, before running real policy. Helps exploration.
CLIP_NORM=True
```

RESULTS

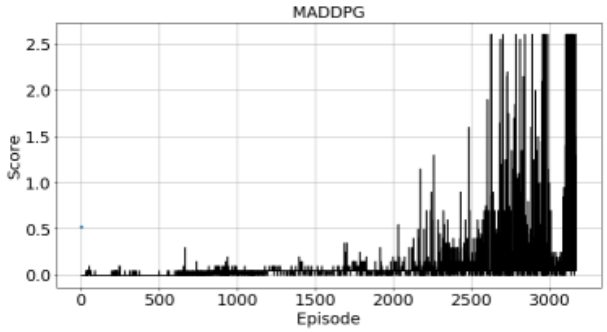
First run on CPU:

Actor:

- (fc1) : Linear (in_features=24, out_features=256, bias)
- (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1)
- (fc2) : Linear(in_features=256, out_features=256)
- (fc3) : Linear(in_features=256, out_features=2)
- tanh activation

Critic:

- (fc1): Linear(in_features=52, out_features=256)
- (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1)
- (fc2) : Linear(in_features=256, out_features=256)
- (fc3) : Linear(in_features=256, out_features=1)
- no activation




```
Episode 100     Average Score: 0.02
Episode 200     Average Score: 0.002
Episode 300     Average Score: 0.020
Episode 400     Average Score: 0.012
Episode 500     Average Score: 0.001
Episode 600     Average Score: 0.010
Episode 700     Average Score: 0.031
Episode 800     Average Score: 0.033
Episode 900     Average Score: 0.063
Episode 1000    Average Score: 0.07
Episode 1100    Average Score: 0.077
Episode 1200    Average Score: 0.077
Episode 1300    Average Score: 0.097
Episode 1400    Average Score: 0.089
Episode 1500    Average Score: 0.098
Episode 1600    Average Score: 0.089
Episode 1700    Average Score: 0.098
Episode 1800    Average Score: 0.109
Episode 1900    Average Score: 0.090
Episode 2000    Average Score: 0.089
Episode 2100    Average Score: 0.108
Episode 2200    Average Score: 0.110
Episode 2300    Average Score: 0.131
Episode 2400    Average Score: 0.143
Episode 2500    Average Score: 0.154
Episode 2600    Average Score: 0.165
Episode 2700    Average Score: 0.296
Episode 2800    Average Score: 0.339
Episode 2900    Average Score: 0.333
Episode 3000    Average Score: 0.403
Episode 3100    Average Score: 0.140
Episode 3174    Average Score: 0.524
 Episode 3174   Average Score: 0.52Environment solved!
Wall time: 2h 11min 46s
```

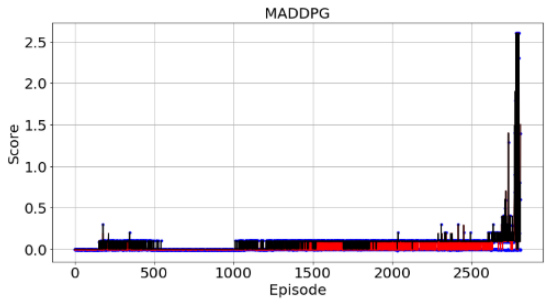
Second Run on Udacity GPU server:

Actor:

- (fc1) : Linear (in_features=24, out_features=512, bias)
- (bn1): BatchNorm1d(512, eps=1e-05, momentum=0.1)
- (fc2) : Linear(in_features=512, out_features=256)
- (fc3) : Linear(in_features=256, out_features=2)
- tanh activation

Critic:

- (fc1): Linear(in_features=52, out_features=512)
- (bn1): BatchNorm1d(512, eps=1e-05, momentum=0.1)
- (fc2) : Linear(in_features=512, out_features=256)
- (fc3) : Linear(in_features=256, out_features=1)
- no activation




```
Episode 100     Average Score: 0.00
Episode 200     Average Score: 0.020
Episode 300     Average Score: 0.042
Episode 400     Average Score: 0.064
Episode 500     Average Score: 0.046
Episode 600     Average Score: 0.014
Episode 700     Average Score: 0.001
Episode 800     Average Score: 0.000
Episode 900     Average Score: 0.000
Episode 1000    Average Score: 0.00
Episode 1100    Average Score: 0.030
Episode 1200    Average Score: 0.043
Episode 1300    Average Score: 0.044
Episode 1400    Average Score: 0.064
Episode 1500    Average Score: 0.096
Episode 1600    Average Score: 0.099
Episode 1700    Average Score: 0.099
Episode 1800    Average Score: 0.069
Episode 1900    Average Score: 0.076
Episode 2000    Average Score: 0.097
Episode 2100    Average Score: 0.109
Episode 2200    Average Score: 0.090
Episode 2300    Average Score: 0.109
Episode 2400    Average Score: 0.100
Episode 2500    Average Score: 0.100
Episode 2600    Average Score: 0.090
Episode 2700    Average Score: 0.129
Episode 2800    Average Score: 0.472
Episode 2810    Average Score: 0.507
 Episode 2810   Average Score: 0.50Environment solved!
CPU times: user 39min 49s, sys: 2min 20s, total: 42min 9s
Wall time: 43min 49s
```

IMPROVEMENTS

- **Prioritized replay**: Sampling of the replay buffer can be prioritized based on each sample's TD error, thereby biasing sampling to samples with higher TD error or rare samples of the state space.
- **Hyperparameter selection**: More systematic way of selecting optimal hyperparameters using Bayesian optimization or starting with grid search. Not being able to connect to a GPU restricted this option for me.
- **Other algorithms:** DDPG requires a lot of tweaking. The next step is to try the algorithms in OpenAI benchmarks such as Proximal Policy Optimization (PPO), Distributed Distributional Deep Deterministic Policy Gradient (D4PG), and twin-delayed DDPG (TD3).