Project 3: Collaboration and Competition

1. Introduction

The Collaboration and Competition project has two agents playing tennis by controlling rackets to bounce a tennis ball over the net. An agent receives a reward of +0.1 if it the tennis ball goes over the net. 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. But we want our agents to collaborate. To achieve this, we “encourage” both agents to work together to achieve a score of +0.5.

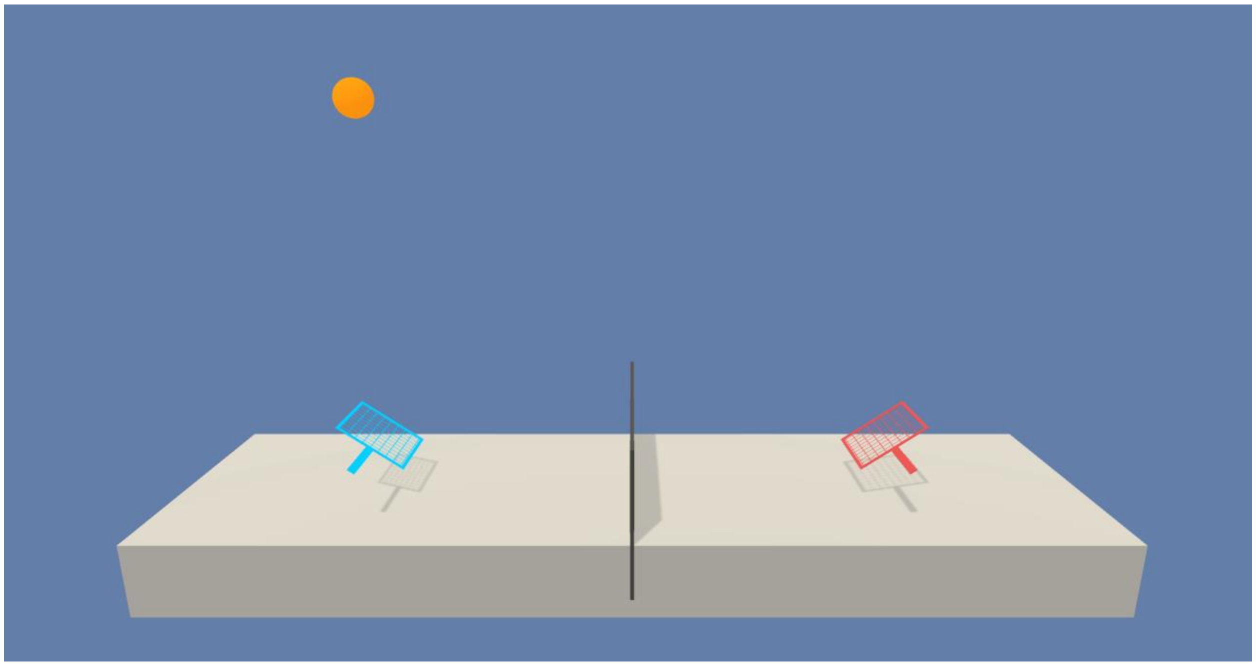


Figure 1 Tennis Environment

We will investigate, implement and assess the Deep Deterministic Policy Gradient (DDPG) deep reinforcement learning algorithm that maximize rewards over training.

We have a reinforcement learning setup with states, actions, rewards between the agent and the environment.

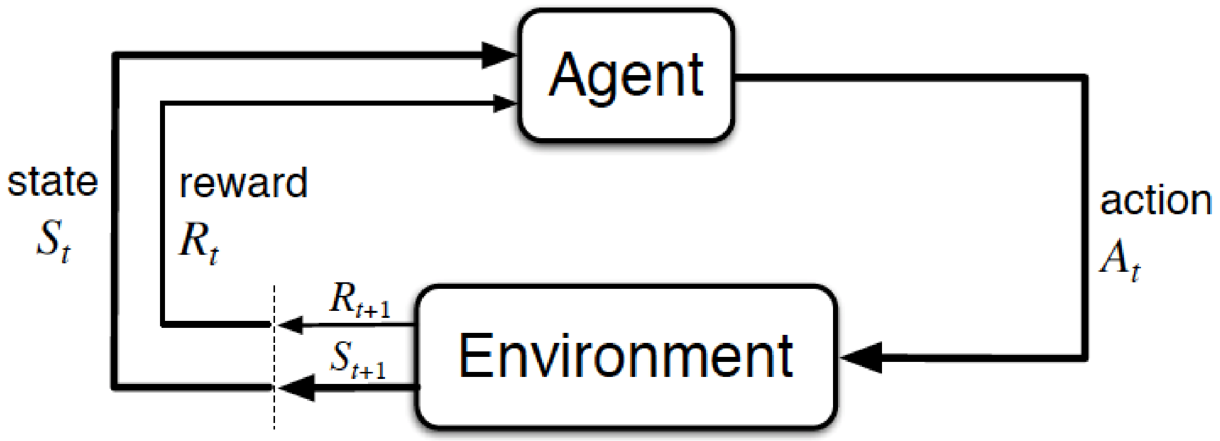


Figure 2 The agent-environment interaction in reinforcement learning. (Source: Sutton and Barton, 2017)

Rewards

A reward of +0.1 is awarded for hitting the ball over the net and reward of -0.01 is given for dropping the ball or letting the ball go out of bounds. The goal of each agent is to keep the ball in play.

To compute the rewards per episode:

- We add up the rewards received by each agent, to get the score per agent. We then take the maximum of the two scores received by each agent;
- This yields the score for the episode and agents are trying to maximize over many episodes;

The environment is considered solved when the average (over 100 episodes) of the rewards is at least +0.5.

States

The observation state space has 8 variables corresponding to position, velocity of ball and racket. Each agent receives its own local observation.

Actions

There are two continuous actions corresponding to movement toward or away from the net and jumping.

We finish this project by looking at potential extensions to the above algorithms.

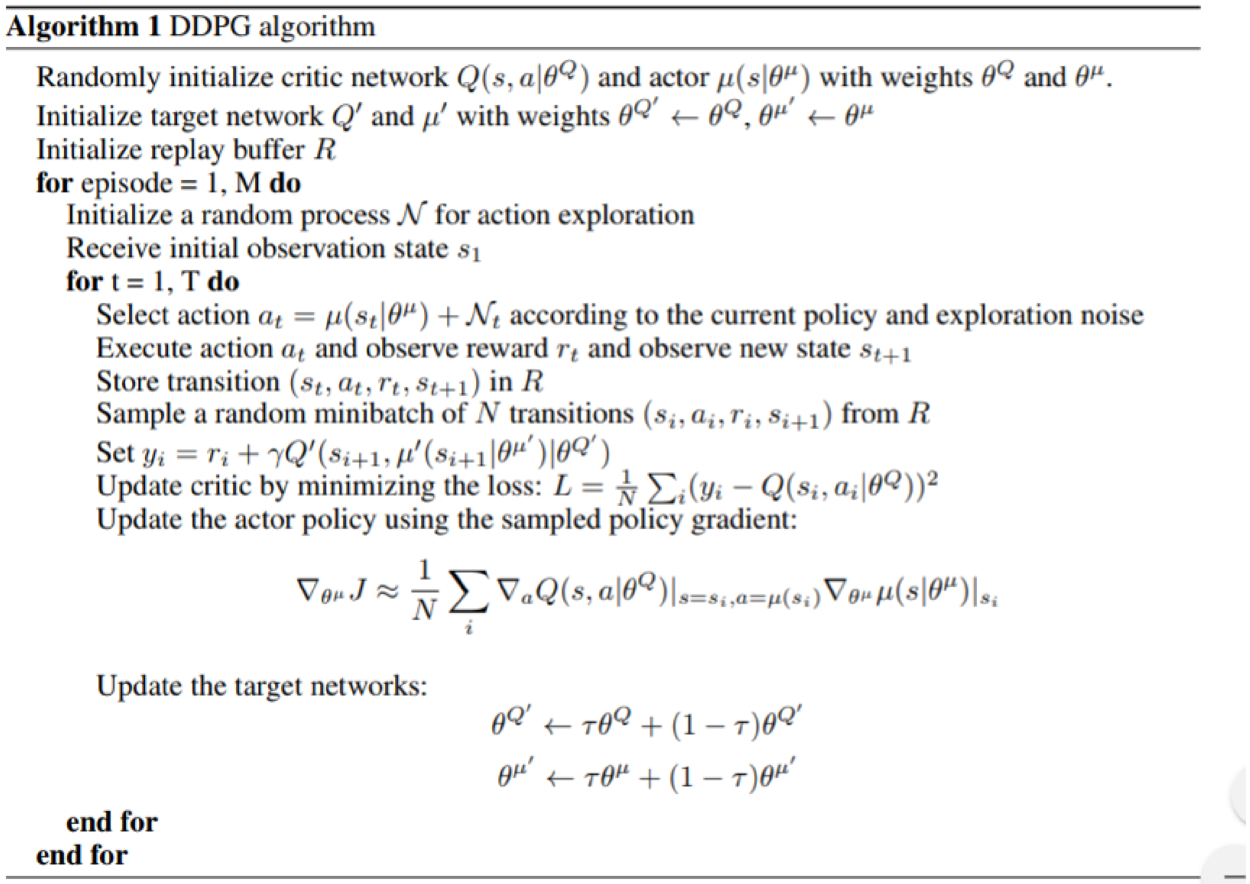
2. Learning Algorithm

The deep reinforcement learning algorithm implemented is Deep Deterministic Policy Gradient with a Replay Buffer and Noise.

Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient algorithm is a model-free approach to reinforcement that can learn action policies directly from low-dimensional observations using an Actor-Critic network as an underlying deep learning architecture.

The DDPG algorithm is an extension of the Deterministic Policy Gradient (DPG) algorithm (Silver et al., 2014). The algorithm itself is as follows:



```
**Algorithm 1** DDPG algorithm
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
end for
```

Figure 3 DDPG Algorithm

The algorithm has a number of features and or attributes:

An Actor-Critic Network

Function approximation is done through a parameterized Actor function which specifies the current policy by deterministically mapping states to a specific action and the Critic function is learnt using the Bellman equation as in Q-Learning. The Actor continuously updates the policy while the Critic evaluates this off-policy. Here I have implemented the Actor-Critic networks as three layer neural nets:

```
Actor(
  (fc1): Linear(in_features=33, out_features=400, bias=True)
  (fc2): Linear(in_features=400, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=4, bias=True)
)
Critic(
  (fc1): Linear(in_features=33, out_features=400, bias=True)
  (fc2): Linear(in_features=404, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=1, bias=True)
)
```

I found that using the exact same network for both Actor and Critic networks improved agent training.

Fixed Q-Targets

An import characteristics of the algorithm is the use of fixed q-targets. We use fixed Q-Targets for both the Actor and Critic in our agent to avoid harmful correlations by decoupling the parameters from the target network – which is fixed and gets updated a lot less frequently that the local network.

Replay Buffer

Just like in Deep Q-Network (DQN) we use a replay buffer to enable the agent to learn from experience. The replay buffer stores transitions sampled from the environment according to our exploration policy, discussed below. The additional power of this replay buffer is that multiple agents can share experiences and benefit from learning across a set of uncorrelated transitions.

Noise

In their paper, the Google Deepmind team, introduced a Noise component to address exploration. Exploration is a major challenge of learning in continuous action spaces. Since DDPG is off-policy the problem of exploration can be addressed independently from the learning problem.

DDPG introduces an exploration μ' by adding noise sampled from a noise process Ņ to the actor policy:

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$$

Ņ can be chosen to suit the environment. Here we implemented an Ornstein-Uhlenbeck process to generate temporally correlated exploration for exploration efficiency. The agent samples this noise during training to enable exploration.

3. Results

The two agents achieve the target score of +0.5 after 1346 episodes:

```
Episode 100     Average Score: 0.01     Score: 0.00
Episode 200     Average Score: 0.01     Score: 0.00
Episode 300     Average Score: 0.00     Score: 0.00
Episode 400     Average Score: 0.00     Score: 0.09
Episode 500     Average Score: 0.01     Score: 0.09
Episode 600     Average Score: 0.01     Score: 0.00
Episode 700     Average Score: 0.01     Score: 0.00
Episode 800     Average Score: 0.07     Score: 0.10
Episode 900     Average Score: 0.08     Score: 0.10
Episode 1000    Average Score: 0.10     Score: 0.10
Episode 1100    Average Score: 0.22     Score: 0.20
Episode 1200    Average Score: 0.43     Score: 0.10
Episode 1300    Average Score: 0.38     Score: 0.50
Episode 1346    Average Score: 0.51     Score: 2.60
Environment solved in 1346 episodes!    Average Score: 0.51
```

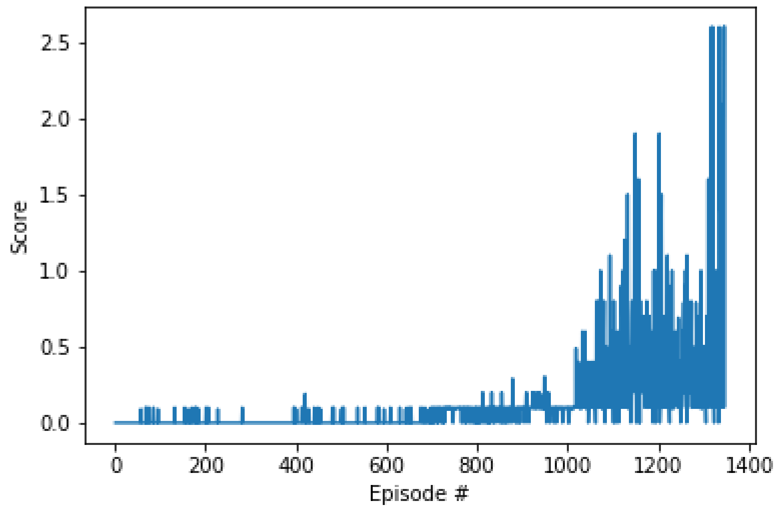


Figure 4 Results

4. Ideas for future work

- Hyper-parameter tuning

  I started tuning the hyper-parameters for this algorithm, adding more layers, drop out and batch normalization didn't improve the speed of training or convergence. I can tune these a bit more using grid search.