Udacity Deep Reinforcement Learning Nanodegree

Collaboration and Competition Project Report

Yasmeen Khaled
June 16, 2021

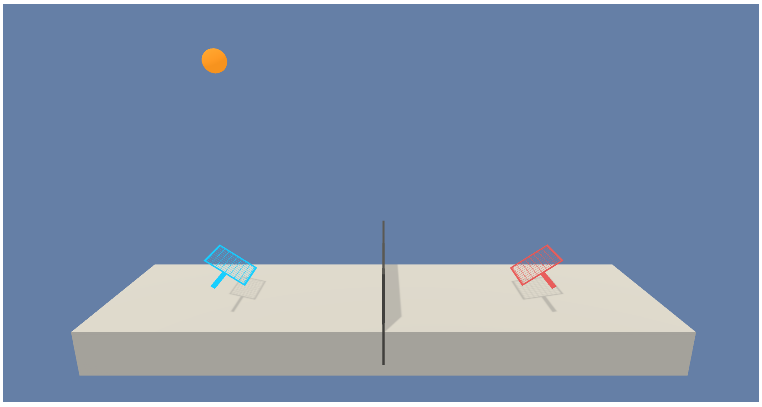

Figure 0.1: Tennis Environment

1 Introduction

During the last couple of years, with DeepMind introducing DQN which managed to beat human experts in Atari games, Deep Reinforcement Learning became a very hot topic that attracted alot researchers. The goal for any RL agent is to maximize its expected reward upon interaction with an environment. The idea behind the usage of neural networks is that approximating policies and/or value functions can actually output satisfying practical results. The environment used for this project he Udacity version of the Tennis environment. In this environment, two agents control rackets to bounce a ball over a net. The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Thus, the goal of each agent is to keep the ball in play.

2 Learning Algorithm

The algorithm used to solve this project is the Multi Agent Deep Deterministic Policy Gradient (MADDPG) following this research paper. The algorithm is an extension to DDPG which is a model free, off-policy actor-critic algorithm that is used to solve environments with continuous action-spaces. The algorithm is explained in the following figure:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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 2.1: DDPG Algorithm

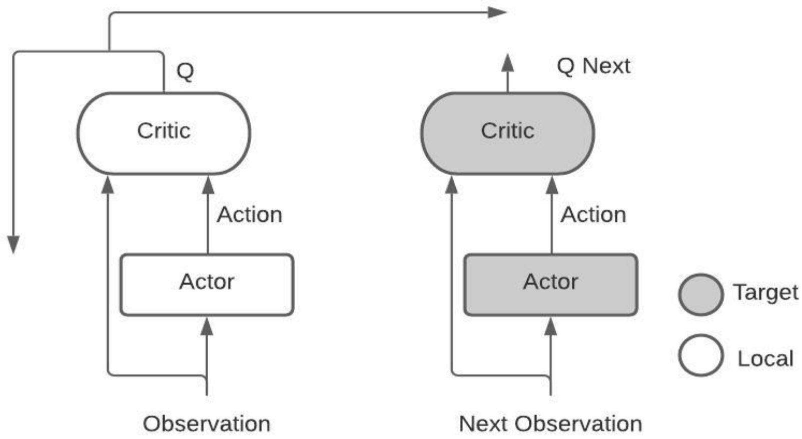

Figure 2.2: DDPG Diagram

Updating the actor is used using the local actor and critic network, we try to maximize the output Q, or in other words, minimize -Q. To train the critic, we use the next observation and pass it to the target actor and critic, then get Qtargetnext. Qtarget = Reward+ discount*Qtargetnext, and now we have Q and Qtargetnext so we want to minimize the difference between them.

The MADDPG can be used to solve cooperative, competitive and mixed environments. The way the rewards are defined defines the type of the environment. The critic is MADDPG uses extra information like states observed and actions taken by all other agents, while the actor can access only its agent observation and actions. MADDPG follows centralized training and decentralized testing. The critic is observing actions taken by everyone and the actor is limited by its own information.
This project was implemented as an extension to the previous one. The replay buffer is fed with samples from both agents, since after all, the goal of both agents is the same.

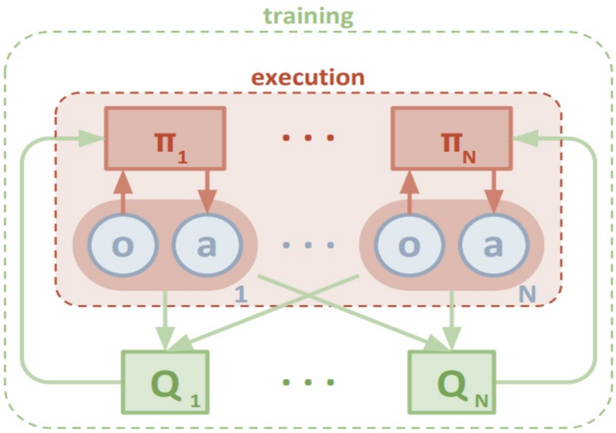

Figure 2.3: MADDPG Diagram

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

Figure 2.4: MADDPG Algorithm

The hyperparamter values used were as follows:

Hyperparameter	Description	Value
BUFFER_SIZE	replay buffer size	int(1e5)
BATCH_SIZE	minibatch size	128
GAMMA	discount factor	0.99
TAU	for soft update of target parameters	1e-3
LR_ACTOR	Learning rate for the actor	1e-4
LR_CRITIC	Learning rate for the critic	1e-3
UPDATE_EVERY	how often to update the network	20
NUM_UPDATES	how many times update the network	200
NOISE_DECAY	a value of decreasing the noise factor every UPDATE_EVERY	0.99999

The maddpg_agent is implemented as follows:

- **init**: the agent state_size and action_size is passed to the constructor and all the values are initialized. Two critic networks are initialized: local and target and same for the actor.
- **step**: it saves the experience in the replay buffer, and if enough samples are existing in the memory, the agents samples from the buffer and learns every update_every times with num_update times.
- **act**: uses random actions for the first 50 episodes then returns the action vector chosen by actor_local network, trains the actor_local network and adds noise to the action to favor exploration.
- **learn**: it optimizes the losses as explained above and uses soft_update to transfer the local network weights to the target network. Gradient clipping was used when updating local_critic network.
- **soft_update**: the learn method updates the target network using this method which updates the target network based on the values of the local network.

The maddpg_Ft.py also contains the ReplayBuffer class which has the following methods:

- **add**: which adds an experience to the replay buffer memory which is initialized as a deque with maxlen of BUFFER_SIZE.
- **sample**: which randomly samples a batch from the replay buffer memory.

2.1 Actor-Critic Networks Architectures

2.1.1 Actor Architecture

The network architecture was as follows:

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

fc1 is passed to leaky relu (as used in maddpg lab) activator then the output is passed to fc2 then another leaky relu activator in the forward pass.

2.1.2 Critic Architechture

The network architecture was as follows:

```
Critic(
  (input_norm): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
  (fc1): Linear(in_features=24, out_features=128, bias=True)
  (fc2): Linear(in_features=132, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```

fc1 is passed to input_norm then leaky relu activator then the output is passed to fc2 then another leaky relu activator in the forward pass.
fc2 has extra four inputs which correspond to the 4 values of the 2 action vectors of both agents (128+2+2).

3 Training and Results

```
Episode 10    Average Score: 0.01
Episode 20    Average Score: 0.01
Episode 30    Average Score: 0.01
Episode 40    Average Score: 0.01
Episode 50    Average Score: 0.01
Episode 60    Average Score: 0.01
Episode 70    Average Score: 0.01
Episode 80    Average Score: 0.01
Episode 90    Average Score: 0.01
Episode 100    Average Score: 0.01
Episode 110    Average Score: 0.01
Episode 120    Average Score: 0.02
Episode 130    Average Score: 0.03
Episode 140    Average Score: 0.03
Episode 150    Average Score: 0.04
Episode 160    Average Score: 0.05
Episode 170    Average Score: 0.06
Episode 180    Average Score: 0.07
Episode 190    Average Score: 0.08
Episode 200    Average Score: 0.08
Episode 210    Average Score: 0.08
Episode 220    Average Score: 0.08
Episode 230    Average Score: 0.09
Episode 240    Average Score: 0.09
Episode 250    Average Score: 0.10
Episode 260    Average Score: 0.13
Episode 270    Average Score: 0.17
Episode 280    Average Score: 0.19
Episode 290    Average Score: 0.22
Episode 300    Average Score: 0.24
Episode 310    Average Score: 0.26
Episode 320    Average Score: 0.29
Episode 330    Average Score: 0.32
Episode 340    Average Score: 0.34
Episode 350    Average Score: 0.35
Episode 360    Average Score: 0.34
Episode 370    Average Score: 0.36
Episode 380    Average Score: 0.36
Episode 390    Average Score: 0.38
Episode 400    Average Score: 0.47
Episode 404      Average Score: 0.52
```

```
Environment solved in 304 episodes!      Average100 Score: 0.52

CPU times: user 1h 7min 3s, sys: 2min 16s, total: 1h 9min 19s
Wall time: 1h 10min 21s
```

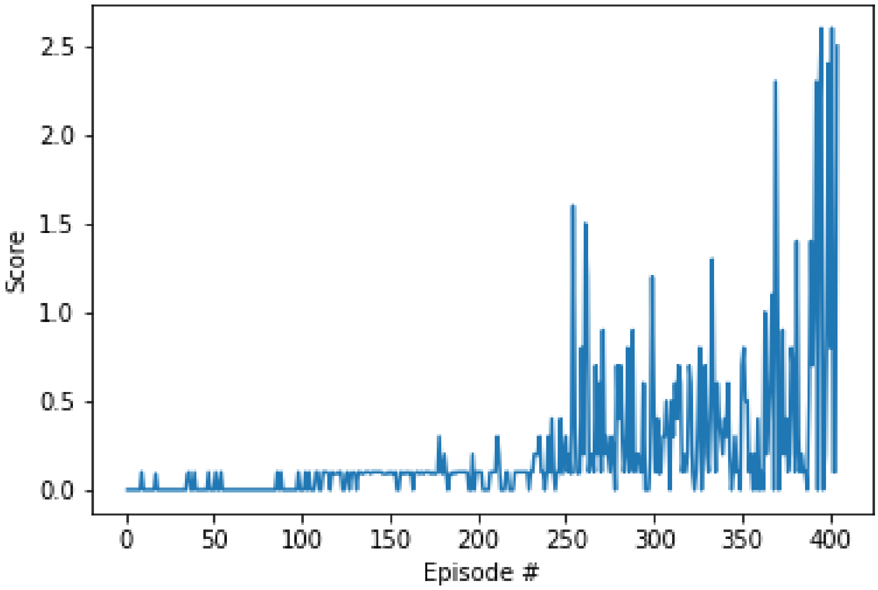

Figure 3.1: MADDPG Training

My MADDPG agent managed to solve the task in 404 episodes to be able to get a reward of at least +0.5 over 100 consecutive episodes.

3.1 Testing

After loading the saved weights and testing the agent for one episode this was the result:

```
Total score of this episode: 0.9950000150129199
```

4 Future Work

To further improve the performance of the model, the current methods are proposed to do in the future:

- **Twin Delayed DDPG (TD3)**: TD3 improves the performance of of DDPG by 3 main modifications which are: Clipped Double Q-learning to reduce positive bias and decouple action selection and evaluation, Delayed Policy updates where the policy is updated less frequently than the Q-function,for example, one policy update for every two Q-function updates ,and lastly, Target poilcy smoothing which adds noise to the target action to make it harder for the policy to exploit Q-function.

- **Hyper-parameter search**: one can try to train multiple agents with different parameters in order to reach the set that significantly improve the performance.
- **Prioritized Experience Replay**: in which experience with more significane can be given high priority to be sampled and hence improve the learning.