Project 3 – Collaboration and Competition

May 20, 2020

1 Introduction and objectives

For this project, two agents is trained in the *Tennis* environment to control rackets to bounce a ball over a net. The primary objective is to **get an average score of** $+0.5$ **over** 100 **consecutive episodes after taking the maximum over both agents**. The *Multi-Agent Deep Deterministic Policy Gradient (MADDPG)* and *Multi-Agent Twin Delayed Deep Deterministic (MATD3)* are utilized for the task. The project is completed in the *Udacity Workspaces*.

2 Environment

Two agents is trained in the *Tennis* environment to control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of $+0.10$. 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. 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, the agents must get an average score of $+0.5$ (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

- After each episode, the rewards that each agent received (without discounting) are added up, to get a score for each agent. This yields 2 (potentially different) scores. By taking the maximum of these 2 scores, yields a single score for each episode.
- The environment is considered solved, when the average (over 100 episodes) of those scores is at least $+0.5$

3 Multi-Agent Deep Deterministic Policy Gradient

The MADDPG algorithm performs centralized training with decentralized execution as follows:

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient

Initialize critic network $Q(s, a|\theta)$with weight θ for each agent.
Initialize actor network $\pi(s|\phi)$ with weight ϕ for each agent.
Initialize corresponding target network $\theta' \leftarrow \theta$, $\phi' \leftarrow \phi$ for each agent.
Initialize replay buffer B.
for $t \leftarrow 1$ **to** *number of episode+1* **do**
    $score \leftarrow 0$
    for $t \leftarrow 0$ **to** *number of step-1* **do**
        For each agent j, execute action $A_t \leftarrow \pi(S_t|\phi) + \nu$ based on current policy and exploration noise, observe reward R_t and state S_{t+1}, store transition (S_t, A_t, R_t, S_{t+1}) in B.
        for *agent* $j \leftarrow 1$ **to** *number of agent-1* **do**
            Sample minibatch of N transition (S_t, A_t, R_t, S_{t+1}) from B
            Evaluate target $\tilde{Q} \leftarrow R_t + \gamma\left[Q'\left(S_{t+1}, \pi'\left(S_{t+1}|\phi'\right)|\theta'\right)\right]$.
            Compute gradient descent by minimizing $N^{-1}\sum\left(\tilde{Q} - Q(S_t, A_t)\right)^2$.
            Update critic network θ.
            Update ϕ by deterministic policy gradient $\nabla_\phi J = N^{-1}\sum \nabla_a Q_1\left(S_t, A_t|\theta_1\right)|_{a=\pi_\phi(S_t)} \nabla_\phi \pi_\phi\left(S_t\right)$.
            Update target network $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$, $\phi' \leftarrow \tau\phi + (1-\tau)\phi$.
        end
    end
    Noise decay $\nu = \max(\nu_{\text{end}}, \nu_{\text{decay}}\epsilon)$
end
return *scores*

The *multiagent.py* served as a wrapper to handle two MADDPG agents in *agent.py*:

Code Listing 1: Multi-agent initialization

```
class Multi_Agent():
  def __init__(self, num_agents=2, state_size=24, action_size=2, random_seed=0, TD3=
                                                    False):

    self.num_agents = num_agents
    self.state_size = state_size
    self.action_size = action_size

    if(TD3):
      # use TD3
      self.agents = [TD3_Agent(state_size, action_size, i+1, random_seed) for i in
                                                    range(num_agents)]
    else:
      # use DDPG
      self.agents = [DDPG_Agent(state_size, action_size, i+1, random_seed) for i in
                                                    range(num_agents)]
```

The local and target of decentralized actor network and centralized critic network of the two agents are constructed using 2 hidden fully connected layer neural network. The loss is minimized by using the Adaptive Moment Estimation (Adam) optimizer:

Code Listing 2: Decentralized actor network and centralized critic network

```
# Actor Network (w/ Target Network)
self.actor_local = Actor(state_size, action_size, random_seed).to(device)
self.actor_target = Actor(state_size, action_size, random_seed).to(device)
self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

# Critic Network (w/ Target Network)
self.critic_local = Critic(state_size, action_size, random_seed).to(device)
self.critic_target = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC,
                                   weight_decay=WEIGHT_DECAY)
```

The model architecture for the decentralized actor networks which consist of 2 hidden fully connected layer neural network with Relu activation function, and hyperbolic tangent at the output layer:

Code Listing 3: Decentralized actor network architecture

```
def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
  super(Actor, self).__init__()
  self.seed = torch.manual_seed(seed)
  self.fc1 = nn.Linear(state_size, fc1_units)
  self.fc2 = nn.Linear(fc1_units, fc2_units)
  self.fc3 = nn.Linear(fc2_units, action_size)

def forward(self, state):
  x = F.relu(self.fc1(state))
  x = F.relu(self.fc2(x))
  return torch.tanh(self.fc3(x))
```

The model architecture for the centralized critic networks of the two agents consist of 2 hidden fully connected layer neural network with Relu activation function, and linear activation function at the output layer. The observations from both agents are used as input of both critic networks, while the actions from both agents are concatenated to the second fully connected layer:

Code Listing 4: Centralized critic network architecture

```
def __init__(self, state_size, action_size, seed, fcs1_units=64, fc2_units=64):
  n_agents = 2
  super(Critic, self).__init__()
  self.seed = torch.manual_seed(seed)
  self.fcs1 = nn.Linear((state_size)*n_agents, fcs1_units)
  self.fc2 = nn.Linear(fcs1_units + (action_size*n_agents), fc2_units)
  self.fc3 = nn.Linear(fc2_units, 1)

def forward(self, state, action):
  xs = F.relu(self.fcs1(state))
  x = torch.cat((xs, action), dim=1)
  x = F.relu(self.fc2(x))
  return self.fc3(x)
```

The action is evaluated based on policy perturbed by exploration noise with decay ν:

Code Listing 5: Action based on policy with exploration noise

```
# multiagent.py
def act(self, states, add_noise=True, nu = 1.0):
  actions = []
  for agent, state in zip(self.agents, states):
    action = agent.act(state, add_noise=add_noise,nu=nu)
    actions.append(action)
  return np.array(actions)

# agent.py
def act(self, state, add_noise=True, nu=1.0):
  state = torch.from_numpy(state).float().to(device)
  self.actor_local.eval()
  with torch.no_grad():
  action = self.actor_local(state).cpu().data.numpy()
  self.actor_local.train()
  if add_noise:
    action += nu*self.noise.sample()
  return np.clip(action, -1, 1)
```

The experience replay buffer contains the experiences of two agents to perform centralized training with decentralized execution:

Code Listing 6: Centralized training with decentralized execution

```
def step(self, states, actions, rewards, next_states, dones, freq):
  states = states.reshape(1, -1)
  actions = actions.reshape(1, -1)
  next_states = next_states.reshape(1, -1)
  self.memory.add(states, actions, rewards, next_states, dones)
  if len(self.memory) > BATCH_SIZE:
    for idx, agent in enumerate(self.agents):
      experiences = self.memory.sample()
      self.learn(experiences, idx, freq)

def learn(self, experiences, idx, freq):
""" observations and actions from all agents. Collect actions from each agent. """
  actions_next = []
  actions_pred = []
  states, _, _, next_states, _ = experiences
  next_states = next_states.reshape(-1, self.num_agents, self.state_size)
  states = states.reshape(-1, self.num_agents, self.state_size)
  for idxx, agent in enumerate(self.agents):
    idt = torch.tensor([idxx]).to(device)
    state = states.index_select(1, idt).squeeze(1)
    next_state = next_states.index_select(1, idt).squeeze(1)
    actions_next.append(agent.actor_target(next_state))
    actions_pred.append(agent.actor_local(state))
  actions_next = torch.cat(actions_next, dim=1).to(device)
  actions_pred = torch.cat(actions_pred, dim=1).to(device)
  agent = self.agents[idx]
  agent.learn(experiences, actions_next, actions_pred, freq)
```

4 Multi-Agent Twin Delayed Deep Deterministic

To alleviate the overestimation of the value function in MADDPG. The MATD3 algorithm is performed to train the multiple agents with the similar framework as follows:

Algorithm 2: Multi-Agent Twin Delayed Deep Deterministic

For each agent, initialize critic network $Q_1(s, a|\theta_1)$, $Q_2(s, a|\theta_2)$ with weight θ_1 and θ_2, respectively.
For each agent, initialize actor network $\pi(s|\phi)$ with weight ϕ.
For each agent, initialize corresponding target network $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$.
Initialize replay buffer B.
for $t \leftarrow 1$ **to** *number of episode+1* **do**
    $score \leftarrow 0$
    for $t \leftarrow 0$ **to** *number of step-1* **do**
        For each agent j, execute action $A_t \leftarrow \pi(S_t|\phi) + \nu$ based on current policy and exploration noise, observe reward R_t and state S_{t+1}, store transition (S_t, A_t, R_t, S_{t+1}) in B.
        for *agent* $j \leftarrow 1$ **to** *number of agent-1* **do**
            Sample minibatch of N transition (S_t, A_t, R_t, S_{t+1}) from B
            Evaluate target $\tilde{Q} \leftarrow R_t + \gamma \min\left[Q_1'\left(S_{t+1}, \pi'\left(S_{t+1}|\phi'\right)|\theta_1'\right), Q_2'\left(S_{t+1}, \pi'\left(S_{t+1}|\phi'\right)|\theta_2'\right)\right]$.
            Compute gradient descent by minimizing $N^{-1}\sum\left(\tilde{Q} - Q(S_t, A_t)\right)^2$.
            Update critic network θ_1 and θ_2.
            if $t \mod freq$ **then**
                Update ϕ by deterministic policy gradient $\nabla_\phi J = N^{-1}\sum \nabla_a Q\left(S_t, A_t|\theta_1\right)|_{a=\pi_\phi(S_t)} \nabla_\phi \pi_\phi\left(S_t\right)$.
                Update target network $\theta_1' \leftarrow \tau\theta_1 + (1-\tau)\theta_1'$, $\theta_2'\tau\theta_2 + (1-\tau)\theta_2'$, $\phi' \leftarrow \tau\phi + (1-\tau)\phi$.
            end
        end
    end
    Noise decay $\nu = \max(\nu_{\text{end}}, \nu_{\text{decay}}\epsilon)$
end
return *scores*

The *multiagent.py* served as a wrapper to handle two MATD3 agents in *agent.py*:

Code Listing 7: Multi-agent initialization

```
class Multi_Agent():
  def __init__(self, num_agents=2, state_size=24, action_size=2, random_seed=0, TD3=
                                                    False):

    self.num_agents = num_agents
    self.state_size = state_size
    self.action_size = action_size

    if(TD3):
      # use TD3
```

```
        self.agents = [TD3_Agent(state_size, action_size, i+1, random_seed) for i in
                                                range(num_agents)]
    else:
        # use DDPG
        self.agents = [DDPG_Agent(state_size, action_size, i+1, random_seed) for i in
                                                range(num_agents)]
```

For each agent, the local and target of decentralized actor network, centralized critic network 1, and centralized critic network 2 are constructed using 2 hidden fully connected layer neural network. The loss is minimized by using the Adaptive Moment Estimation (Adam) optimizer:

Code Listing 8: Decentralized actor network, centralized critic network 1 and 2

```
# Actor Network (w/ Target Network)
self.actor_local = Actor(state_size, action_size, random_seed).to(device)
self.actor_target = Actor(state_size, action_size, random_seed).to(device)
self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

# Critic Network (w/ Target Network)
self.critic_local_1 = Critic(state_size, action_size, random_seed).to(device)
self.critic_target_1 = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer_1 = optim.Adam(self.critic_local_1.parameters(), lr=LR_CRITIC,
                                                weight_decay=WEIGHT_DECAY)

self.critic_local_2 = Critic(state_size, action_size, random_seed).to(device)
self.critic_target_2 = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer_2 = optim.Adam(self.critic_local_2.parameters(), lr=LR_CRITIC,
                                                weight_decay=WEIGHT_DECAY)
```

The model architecture are identical to aforementioned MADDPG. Delayed update of the target and actor networks is introduced to prevent the divergence of value estimation that subsequently jeopardizing the accuracy of policy evaluation:

Code Listing 9: Delayed update of the target and actor networks

```
if freq == 0:
  # ---------------------------- update actor ---------------------------- #
  # Compute actor loss
  #actions_pred = self.actor_local(states)
  actor_loss = -self.critic_local_1(states, actions_pred).mean()
  # Minimize the loss
  self.actor_optimizer.zero_grad()
  actor_loss.backward()
  self.actor_optimizer.step()

  # ----------------------- update target networks ----------------------- #
  self.soft_update(self.critic_local_1, self.critic_target_1, TAU)
  self.soft_update(self.critic_local_2, self.critic_target_2, TAU)
  self.soft_update(self.actor_local, self.actor_target, TAU)
```

5 Training

After some hyperparameter searches, the following parameters are used to train the agent:

64, and 64 nodes of first, and second fully connected layer, respectively;

learning rate of all networks $= 5e-4$, noise decay $= 0.995$;

replay buffer size $= 1000000$, minibatch size $= 512$, discount factor $= 0.99$;

soft update of target parameters $= 0.05$;

Ornstein-Uhlenbeck noise ($\sigma = 0.75$, $\nu_{\text{end}} = 0.001$, $\nu_{\text{decay}} = 0.995$).

Code Listing 10: DDPG

```
agent_maddpg = Multi_Agent(num_agents=num_agents, state_size=state_size, action_size=
                                        action_size, random_seed=23,TD3=False)
ddpg_scores_agents, ddpg_scores_average = maddpg(agent_maddpg,n_episodes=3000,
                                        noise_start=1.0,noise_decay=0.995,noise_end
                                        =0.001)

Episode: 100, MaxScore: 0.00(0.00/-0.01),   Average Score: 0.0048
Episode: 200, MaxScore: 0.00(-0.01/0.00),   Average Score: 0.0020
Episode: 300, MaxScore: 0.10(0.10/-0.01),   Average Score: 0.0291
Episode: 400, MaxScore: 0.00(0.00/-0.01),   Average Score: 0.0713
Episode: 500, MaxScore: 0.10(0.10/-0.01),   Average Score: 0.0685
Episode: 600, MaxScore: 0.10(-0.01/0.10),   Average Score: 0.0816
Episode: 700, MaxScore: 0.00(0.00/-0.01),   Average Score: 0.0688
Episode: 800, MaxScore: 0.00(-0.01/0.00),   Average Score: 0.0772
Episode: 900, MaxScore: 0.20(0.09/0.20),  Average Score: 0.0956
Episode: 1000, MaxScore: 0.30(0.30/0.19),   Average Score: 0.1509
Episode: 1100, MaxScore: 0.19(0.10/0.19),   Average Score: 0.1652
Episode: 1200, MaxScore: 0.10(0.09/0.10),   Average Score: 0.4035

Environment solved in 1109 episodes!  Average Score: 0.51
```

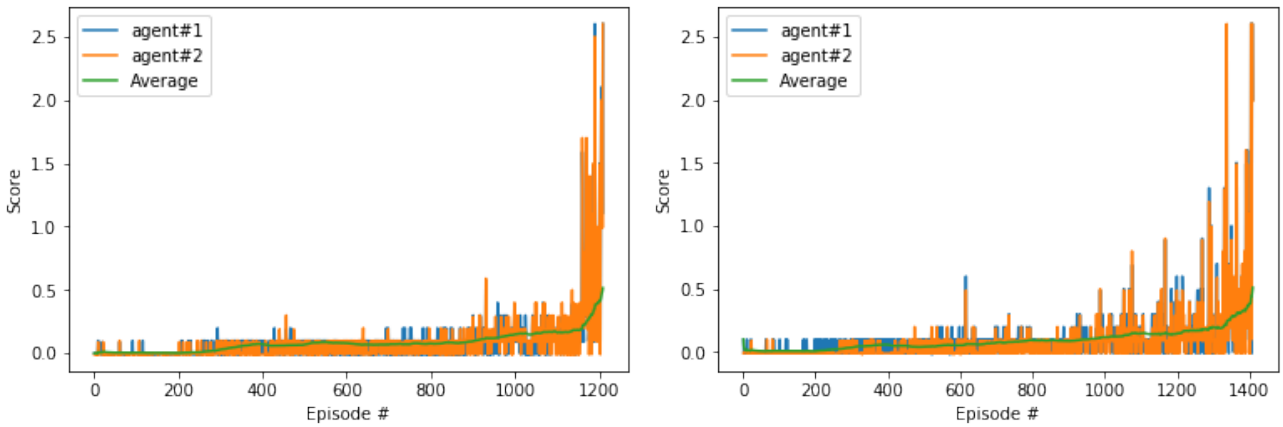
For TD3, in addition to delayed update (TD3) = 2, the identical parameters are applied:

Code Listing 11: Double deep Q-network training

```
agent_matd3 = Multi_Agent(num_agents=num_agents, state_size=state_size, action_size=
                                        action_size, random_seed=23, TD3=True)
td3_scores_agents, td3_scores_average = matd3(agent_matd3,n_episodes=3000, noise_start
                                        =1.0,noise_decay=0.995,noise_end=0.001,freq
                                        =2)

Episode: 100, MaxScore: 0.00(-0.01/0.00),    Average Score: 0.0089
Episode: 200, MaxScore: 0.10(0.10/-0.01),    Average Score: 0.0080
Episode: 300, MaxScore: 0.10(0.10/-0.01),    Average Score: 0.0358
Episode: 400, MaxScore: 0.10(0.10/-0.01),    Average Score: 0.0578
Episode: 500, MaxScore: 0.00(0.00/-0.01),    Average Score: 0.0415
Episode: 600, MaxScore: 0.00(0.00/-0.01),    Average Score: 0.0530
Episode: 700, MaxScore: 0.09(0.09/0.00),   Average Score: 0.0751
Episode: 800, MaxScore: 0.10(0.10/-0.01),    Average Score: 0.0933
Episode: 900, MaxScore: 0.10(0.10/-0.01),    Average Score: 0.0873
Episode: 1000, MaxScore: 0.10(0.10/-0.01),   Average Score: 0.1180
Episode: 1100, MaxScore: 0.10(0.10/-0.01),   Average Score: 0.1474
Episode: 1200, MaxScore: 0.10(0.10/-0.01),   Average Score: 0.1479
Episode: 1300, MaxScore: 0.20(0.20/0.09),    Average Score: 0.1965
Episode: 1400, MaxScore: 0.10(0.09/0.10),    Average Score: 0.3780

Environment solved in 1309 episodes!  Average Score: 0.51
```



(a) MADDPG (b) MATD3

Figure 1: Score comparison of the both agents of MADDPG and MATD3.

As a result, the MADDPG **solved in 1109 episodes with an average score of +0.51 over 100 consecutive episodes** as shown in Figure 1(a). With a similar performance, The MATD3 **solved in 1309 episodes with an average score of +0.51 over 100 consecutive episodes** as shown in Figure 1(b).

6 Future work

1. **Hyperparameter search**: There are still many combinations of hyperparameter reamin to be explored. This study focused on tuning the number of nodes in neural network, replay buffer size and minibatch size. Other parameters remain unexplored are learning rate, discount factor, noise decay, soft update of target parameters, and delayed update of TD3 networks.

2. **Prioritized replay buffer**: In the exploration of the environment, the experiences that occur rarely while being of high importance. The replay buffer can be sampled according to the training loss value, in order to fill the buffer in priority with experiences where the Q-value evaluation was poor, so that the network will have better training on "unexpected events".

3. **Hindsight Experience Replay**: Since this project is dealing with sparse rewards. Hindsight Experience Replay can be utilized to sample-efficient learning from rewards which are sparse and binary.

4. **Proximal Policy Optimization (PPO)**: PPO is an on-policy algorithm that is using a clipped surrogate objective while retaining excellent performance. PPO has been validated against various benchmark test cases and proved to produce good results with much greater simplicity.

5. **Distributed Distributional DDPG (D4PG)**: D4PG applies a set of improvements on DDPG by applying the distributional networks. The key ingredients include Distributional Critic, N-step returns, and Multiple Distributed Parallel Actors.