Tennis-done

August 3, 2022

1 Collaboration and Competition

In this notebook, you will learn how to use the Unity ML-Agents environment for the third project of the Deep Reinforcement Learning Nanodegree program.

1.0.1 1. Start the Environment

We begin by importing the necessary packages. If the code cell below returns an error, please revisit the project instructions to double-check that you have installed Unity ML-Agents and NumPy.

[1]:
```
from unityagents import UnityEnvironment
import numpy as np
```

Next, we will start the environment! ***Before running the code cell below***, change the `file_name` parameter to match the location of the Unity environment that you downloaded.

- **Mac**: `"path/to/Tennis.app"`
- **Windows** (x86): `"path/to/Tennis_Windows_x86/Tennis.exe"`
- **Windows** (x86_64): `"path/to/Tennis_Windows_x86_64/Tennis.exe"`
- **Linux** (x86): `"path/to/Tennis_Linux/Tennis.x86"`
- **Linux** (x86_64): `"path/to/Tennis_Linux/Tennis.x86_64"`
- **Linux** (x86, headless): `"path/to/Tennis_Linux_NoVis/Tennis.x86"`
- **Linux** (x86_64, headless): `"path/to/Tennis_Linux_NoVis/Tennis.x86_64"`

For instance, if you are using a Mac, then you downloaded `Tennis.app`. If this file is in the same folder as the notebook, then the line below should appear as follows:

```
env = UnityEnvironment(file_name="Tennis.app")
```

[2]:
```
env = UnityEnvironment(file_name="/data/Tennis_Linux_NoVis/Tennis")
```

```
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: ,
```

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

[3]:
```python
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

1.0.2 2. Examine the State and Action Spaces

In this 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. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

Run the code cell below to print some information about the environment.

[4]:
```python
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)

# examine the state space
states = env_info.vector_observations
state_size = states.shape[1]
print('There are {} agents. Each observes a state with length: {}'.
 ↪format(states.shape[0], state_size))
print('The state for the first agent looks like:', states[0])
```

```
Number of agents: 2
Size of each action: 2
```

```
There are 2 agents. Each observes a state with length: 24
The state for the first agent looks like: [ 0.          0.          0.
0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.         -6.65278625 -1.5
 -0.          0.          6.83172083  6.         -0.          0.        ]
```

1.0.3 3. Take Random Actions in the Environment

In the next code cell, you will learn how to use the Python API to control the agents and receive feedback from the environment.

Once this cell is executed, you will watch the agents' performance, if they select actions at random with each time step. A window should pop up that allows you to observe the agents.

Of course, as part of the project, you'll have to change the code so that the agents are able to use their experiences to gradually choose better actions when interacting with the environment!

[5]:
```
for i in range(1, 6):                                      # play game for 5␣
↪episodes
    env_info = env.reset(train_mode=False)[brain_name]     # reset the␣
↪environment
    states = env_info.vector_observations                  # get the current␣
↪state (for each agent)
    scores = np.zeros(num_agents)                          # initialize the␣
↪score (for each agent)
    while True:
        actions = np.random.randn(num_agents, action_size) # select an action␣
↪(for each agent)
        actions = np.clip(actions, -1, 1)                  # all actions␣
↪between -1 and 1
        env_info = env.step(actions)[brain_name]           # send all actions␣
↪to tne environment
        next_states = env_info.vector_observations         # get next state␣
↪(for each agent)
        rewards = env_info.rewards                         # get reward (for␣
↪each agent)
        dones = env_info.local_done                        # see if episode␣
↪finished
        scores += env_info.rewards                         # update the score␣
↪(for each agent)
        states = next_states                               # roll over states␣
↪to next time step
        if np.any(dones):                                  # exit loop if␣
↪episode finished
            break
    print('Score (max over agents) from episode {}: {}'.format(i, np.
↪max(scores)))
```

```
Score (max over agents) from episode 1: 0.0
Score (max over agents) from episode 2: 0.0
Score (max over agents) from episode 3: 0.0
Score (max over agents) from episode 4: 0.0
Score (max over agents) from episode 5: 0.0
```

1.0.4 4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! When training the environment, set `train_mode=True`, so that the line for resetting the environment looks like the following:

```
env_info = env.reset(train_mode=True)[brain_name]
```

1.0.5 Define Actor & Critic Models

Neural network architectures - Actor - Input layer (size 8) - FC layer (size 256) - Relu - Batch Normalization - FC layer (size 128) - Relu - Output layer (size 2) - Tanh - Critic - Input layer (size 8) - FC layer (size 256) - Relu - Batch Normalization - FC layer (size 128) - Relu - Output layer (size 1)

[6]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

[7]:
```
use_cuda = torch.cuda.is_available()

if use_cuda:
    print("CUDA Device:", torch.cuda.get_device_name(0))
    print("Memory Allocated:", round(torch.cuda.memory_allocated(0)/1024**3,1),␣
 ↪"GB")
else:
    print("Training on CPU")
```

```
CUDA Device: Tesla K80
Memory Allocated: 0.0 GB
```

[8]:
```
def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)
```

[9]:
```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=256,␣
 ↪fc2_units=128):
        """Initialize parameters and build model.
        Params
```

```
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.bn1 = nn.BatchNorm1d(fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = self.bn1(x)
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))
```

[10]:
```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=256,␣
 ↪fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
```

```
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs ->␣
↪Q-values."""
        x = F.relu(self.fcs1(state))
        x = self.bn1(x)
        x = torch.cat((x, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

1.0.6 Implement Deep Deterministic Policy Gradients (DDPG) Agent

The Deep Deterministic Policy Gradients (DDPG) algorithm below successfully solves the task of this project. It learns from the provided environment without any prior knowledge of it or data labels and maximizes reward by interacting with the environment.

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

(description courtesy of https://arxiv.org/pdf/1509.02971.pdf, see page 5)

Based on this implementation, adaptations were needed in the model and Noise process to adapt from one to multiple agents. Moreover, the learning rate of the critic was reduced to 1e-4 which lead to faster and more stable learning.

[11]:
```python
import random
import copy
from collections import namedtuple, deque
```

[12]:
```python
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.stack([e.state for e in experiences if e is not None])).float()
        actions = torch.from_numpy(np.stack([e.action for e in experiences if e is not None])).float()
        rewards = torch.from_numpy(np.stack([e.reward for e in experiences if e is not None])).float()
        next_states = torch.from_numpy(np.stack([e.next_state for e in experiences if e is not None])).float()
        dones = torch.from_numpy(np.stack([e.done for e in experiences if e is not None]).astype(np.uint8)).float()
        if use_cuda:
            states = states.cuda()
            actions = actions.cuda()
            rewards = rewards.cuda()
```

```
            next_states = next_states.cuda()
            dones = dones.cuda()

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

[13]:
```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.size = size
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.
↪standard_normal(self.size)
        self.state = x + dx
        return self.state
```

[14]:
```
class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, index, nb_agents, random_seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            random_seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
```

```
        if use_cuda:
            self.index = index = torch.tensor([index]).cuda()
        else:
            self.index = index = torch.tensor([index])

        self.seed = random.seed(random_seed)

        # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size, random_seed)
        self.actor_target = Actor(state_size, action_size, random_seed)
        if use_cuda:
            self.actor_local = self.actor_local.cuda()
            self.actor_target = self.actor_target.cuda()
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

        # Critic Network (w/ Target Network)
        self.critic_local = Critic(nb_agents * state_size, nb_agents * action_size, random_seed)
        self.critic_target = Critic(nb_agents * state_size, nb_agents * action_size, random_seed)
        if use_cuda:
            self.critic_local = self.critic_local.cuda()
            self.critic_target = self.critic_target.cuda()
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)

        # Noise process
        self.noise = OUNoise(action_size, random_seed)

        self.soft_update(self.critic_local, self.critic_target, 1)
        self.soft_update(self.actor_local, self.actor_target, 1)

    def act(self, state, i_episode=0, add_noise=True):
        """Returns actions for given state as per current policy."""
        if use_cuda:
            state = torch.from_numpy(state).float().cuda()
        else:
            state = torch.from_numpy(state).float()
        self.actor_local.eval()
        with torch.no_grad():
            action = self.actor_local(state).cpu().data.numpy()
        self.actor_local.train()
        if add_noise:
            action += self.noise.sample()
        return np.clip(action, -1, 1)
```

```
    def reset(self):
        self.noise.reset()

    def learn(self, experiences, gamma, actions_target, actions_pred):
        """Update policy and value parameters using given batch of experience tuples.
        Q_targets = r +   * critic_target(next_state, actor_target(next_state))
        where:
            actor_target(state) -> action
            critic_target(state, action) -> Q-value
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences
        rewards = rewards.unsqueeze(-1)
        dones = dones.unsqueeze(-1)

        # ---------------------------- update critic ----------------------------
        # Get predicted next-state actions and Q values from target models
        if use_cuda:
            actions_target = torch.cat(actions_target, dim=1).cuda()
        else:
            actions_target = torch.cat(actions_target, dim=1)
        Q_targets_next = self.critic_target(next_states.reshape(next_states.shape[0], -1), actions_target.reshape(next_states.shape[0], -1))
        # Compute Q targets for current states (y_i)
        Q_targets = rewards.index_select(1, self.index).squeeze(1) + (gamma * Q_targets_next * (1 - dones.index_select(1, self.index).squeeze(1)))
        # Compute critic loss
        Q_expected = self.critic_local(states.reshape(states.shape[0], -1), actions.reshape(actions.shape[0], -1))
        critic_loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # ---------------------------- update actor ---------------------------- #
        if use_cuda:
            actions_pred = torch.cat(actions_pred, dim=1).cuda()
        else:
```

```
            actions_pred = torch.cat(actions_pred, dim=1)
        actor_loss = -self.critic_local(states.reshape(states.shape[0], -1), actions_pred.reshape(actions_pred.shape[0], -1)).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        _target =  *_local + (1 -  )*_target
        Params
        ======
            local_model: PyTorch model (weights will be copied from)
            target_model: PyTorch model (weights will be copied to)
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

1.0.7 Implement Multi-Agent

The Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm below successfully solves the task of this project. It learns from the provided environment without any prior knowledge of it or data labels and maximizes reward by interacting with the environment.

This algorithm trains two separate agents to take actions based on their own observations as well as centralized critics with additional information of all agents. This algorithms builds on the concept of DDPG and brings it to multi-agent tasks by separating out their observations and thereby avoiding the apparent non-stationarity from the perspective of any individual agent in multi-agent environments. This approach does not contain the shortcoming of policy gradient algorithms, like REINFORCE, which exhibit high variance gradient estimates that are cumulating exponentially with the number of agents (leading to exponential decrease of the probability of taking a gradient step in the right direction), and render them unsuitable for problems with a large number of agents.

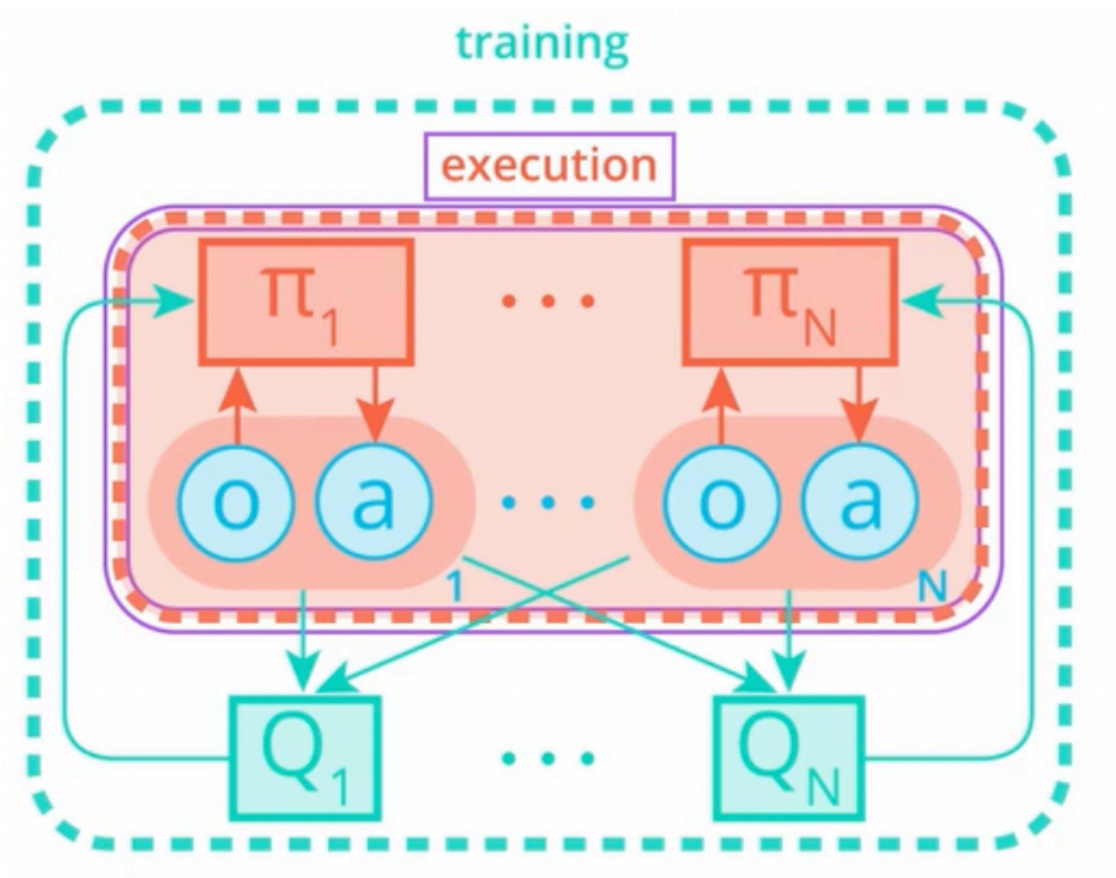
The official paper can be found here: https://arxiv.org/pdf/1706.02275.pdf and the official repo can be found here: https://github.com/openai/maddpg. The paper is exploring deep RL methods for multi-agent domains and introduces MADDPG to solve (among other challenges) the non-stationarity of single agent's environments.

The MADDPG in this repo is an off-policy multi-agent actor-critic approach that uses the concept of target networks with centralized training and decentralized execution. In the reference paper it is used for mixed cooperative-competitive environments.

If we dissect these terms, this means:

- Off-policy means that the agent updates its network using the expected return assuming a greedy policy will be followed (while on-policy approaches estimate the value of a policy while using it for control)

- Multi-agent means that we are working with a system with more than one agent (in this case 2). This leads to interactions between agents, introducing more complexity and needs for novel approaches
- Actor-critic means that the algorithm trains two networks at the same time for function approximation, where the actor learns the policy function mu which returns the optimal action(s) and the critic learns the value function to evaluate the actor's actions and helps improve the training ability
- Target networks means the idea of creating a local and a target network to break the correlation between the target from the actions and thereby stabilize the learning
- Mixed cooperative-competitive means a mixture of cooperative and competitive environments. In cooperative environments, the agents are only concerned about a group task with an across-agent reward. In a competitive environment, each agents is only concered about their own respective reward, where one agent's loss could be the other agent's gain (e.g., a game of two agents playing soccer against each other).
- Centralized training and decentralized execution means that extra information is used by the critic compared to the actor, like states observed and agents taken by other agents. The actors only have access to their own observations and actions:



In general, the approach considers a game with N agents with policies parametrized by Theta_1 to Theta_N and policies pi_1 to pi_N for all agents. The gradient of the expected return for agent i, is then, given...

- simple gradient update, which directly adjusts the policy parameters theta to maximize the objective function J
- where state s is assumed via greedy policy mu and the actions a_i come from policy pi_i,
- and Q_pi_i = Q_mu_i being the centralized action-value function, as explained above, that helps to consider extra information from other agents
- and an experience replay buffer D
- and working with N deterministic continuous policies mu_theta_i

the gradient can be formulated as

$$\nabla_{\theta_i} J(\boldsymbol{\mu}_i) = \mathbb{E}_{\mathbf{x},a\sim\mathcal{D}}[\nabla_{\theta_i}\boldsymbol{\mu}_i(a_i|o_i)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}, a_1, ..., a_N)|_{a_i=\boldsymbol{\mu}_i(o_i)}]$$

with the centralized action-value function being updated as

$$\mathcal{L}(\theta_i) = \mathbb{E}_{\mathbf{x},a,r,\mathbf{x}'}[(Q_i^{\boldsymbol{\mu}}(\mathbf{x}, a_1, \ldots, a_N) - y)^2], \quad y = r_i + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}', a_1', \ldots, a_N')\big|_{a_j'=\boldsymbol{\mu}_j'(o_j)}$$

The DDPG agent contains the following components and configs: - A replay buffer to store memories with the size of 1e5 - Minibatch sizes of 256 - A discount factor of 0.99 for value function approximation - A soft update to blend the regular into the target network of 1e-3 - Learning rates of the actor and critic each set to 1e-4 - Noise according to the Ornstein-Uhlenbeck process with theta=0.15, sigma=0.2 - Repetitions of learning per agent-step of 3

[15]:
```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
UPDATE_EVERY = 2        # Udpate every
LEARNING_REPS = 3       # Learning repetitions per step
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 1e-4        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
```

[16]:
```
class MultiAgents():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, n_agents, random_seed):
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(random_seed)

        self.ma = [Agent(state_size, action_size, i, n_agents, random_seed) for␣
 ↪i in range(n_agents)]

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE,␣
 ↪random_seed)
```

```
        self.t_step = 0

    def step(self, states, actions, rewards, next_states, dones):
        """Save experience in replay memory, and use random sample from buffer␣
↪to learn."""
        self.memory.add(states, actions, rewards, next_states, dones)

        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if len(self.memory) > BATCH_SIZE and self.t_step == 0:
            for _ in range(LEARNING_REPS):
                for agent in self.ma:
                    experiences = self.memory.sample()
                    self.learn(experiences, agent, GAMMA)

                for agent in self.ma:
                    agent.soft_update(agent.critic_local, agent.critic_target,␣
↪TAU)
                    agent.soft_update(agent.actor_local, agent.actor_target,␣
↪TAU)

    def learn(self, experiences, agent, gamma):
        states, actions, _, _, _ = experiences

        if use_cuda:
            actions_target = [agent_j.actor_target(states.index_select(1, torch.
↪tensor([j]).cuda()).squeeze(1)) for j, agent_j in enumerate(self.ma)]
        else:
            actions_target = [agent_j.actor_target(states.index_select(1, torch.
↪tensor([j])).squeeze(1)) for j, agent_j in enumerate(self.ma)]

        agent_action_pred = agent.actor_local(states.index_select(1, agent.
↪index).squeeze(1))
        if use_cuda:
            actions_pred = [agent_action_pred if j==agent.index.cpu().
↪numpy()[0] else actions.index_select(1, torch.tensor([j]).cuda()).squeeze(1)␣
↪for j, agent_j in enumerate(self.ma)]
        else:
            actions_pred = [agent_action_pred if j==agent.index.numpy()[0] else␣
↪actions.index_select(1, torch.tensor([j])).squeeze(1) for j, agent_j in␣
↪enumerate(self.ma)]
        agent.learn(experiences, gamma, actions_target, actions_pred)

    def act(self, states, i_episode=0, add_noise=True):
        actions = [np.squeeze(agent.act(np.expand_dims(state, axis=0),␣
↪i_episode, add_noise), axis=0) for agent, state in zip(self.ma, states)]
        return np.stack(actions)
```

```
    def reset(self):
        for agent in self.ma:
            agent.reset()
```

1.0.8 Train

[17]:
```
# Initialize the agents
m_agents = MultiAgents(state_size=state_size, action_size=action_size,␣
 ↪n_agents=num_agents, random_seed=1)
```

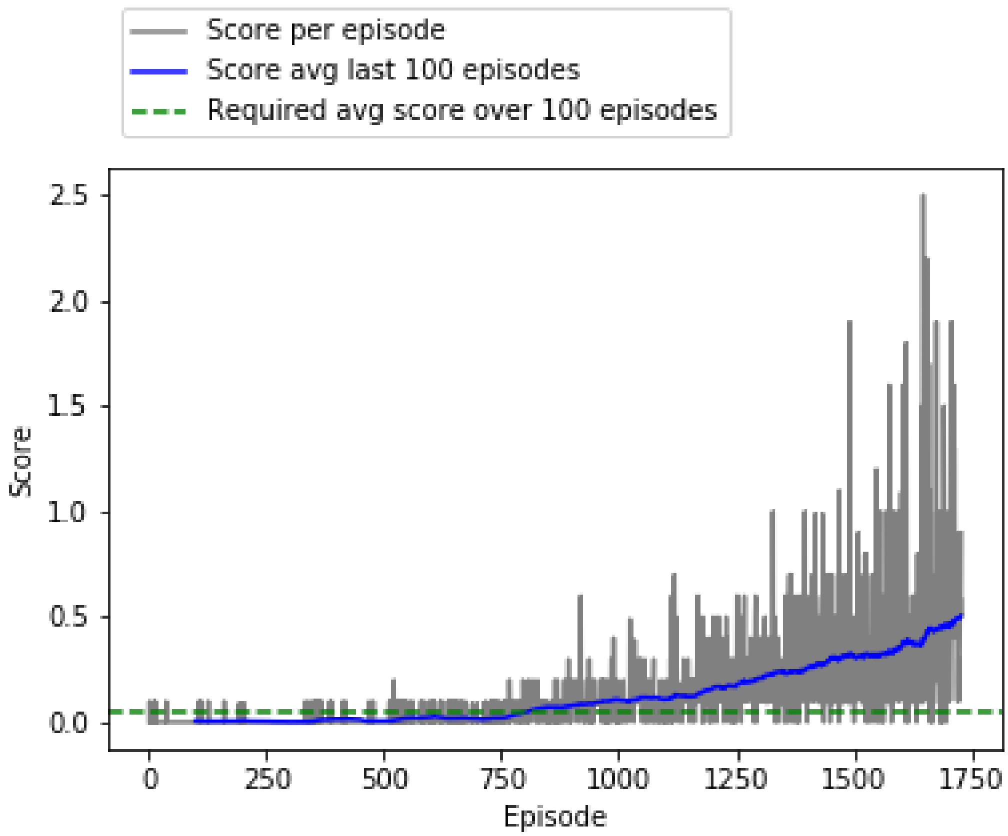
[18]:
```
def train(n_episodes=10000, print_every=100):
    # Create deque to track last 100 scores and scores for full list for plot
    scores_deque = deque(maxlen=print_every)
    scores = []

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name] #activate training
        states = env_info.vector_observations
        m_agents.reset()
        score = np.zeros(num_agents)
        while True:
            actions = m_agents.act(states, i_episode, add_noise=True) # get␣
 ↪MA-actions with noise
            env_info = env.step(actions)[brain_name]
            next_states = env_info.vector_observations         # get next state␣
 ↪(for each agent)
            rewards = env_info.rewards                         # get reward␣
 ↪(for each agent)
            dones = env_info.local_done                        # see if episode␣
 ↪finished

            m_agents.step(states, actions, rewards, next_states, dones)

            states = next_states
            score += rewards
            if any(dones):
                break

        scores_deque.append(np.max(score))
        scores.append(np.max(score))

        print('\rEpisode {}\tAverage Score: {:.3f}'.format(i_episode, np.
 ↪mean(scores_deque)), end="")
        if i_episode % print_every == 0:
```

```
            print('\rEpisode {}\tAverage Score: {:.3f}'.format(i_episode, np.
↪mean(scores_deque)))

        if (np.mean(scores_deque) >= 0.5):
            print('\nSolved in {:d} episodes, with an average score over the␣
↪last 100 episodes of: {:.2f}'.format(i_episode - print_every, np.
↪mean(scores_deque)))
            # Save final agent weights
            for i, agent in enumerate(m_agents.ma):
                torch.save(agent.actor_local.state_dict(),␣
↪f'checkpoint_actor_{i}.pth')
                torch.save(agent.critic_local.state_dict(),␣
↪f'checkpoint_critic_{i}.pth')
            break
        if (i_episode == n_episodes):
            print('\nNot solved in {:d} episodes. Achieved average score over␣
↪the last 100 episodes of: {:.2f}'.format(i_episode - print_every, np.
↪mean(scores_deque)))
            # Save final agent weights
            for i, agent in enumerate(m_agents.ma):
                torch.save(agent.actor_local.state_dict(),␣
↪f'unsolved_checkpoint_actor_{i}.pth')
                torch.save(agent.critic_local.state_dict(),␣
↪f'unsolved_checkpoint_critic_{i}.pth')

    return scores

scores = train()
```

```
Episode 100     Average Score: 0.006
Episode 200     Average Score: 0.007
Episode 300     Average Score: 0.001
Episode 400     Average Score: 0.014
Episode 500     Average Score: 0.004
Episode 600     Average Score: 0.023
Episode 700     Average Score: 0.019
Episode 800     Average Score: 0.046
Episode 900     Average Score: 0.077
Episode 1000    Average Score: 0.108
Episode 1100    Average Score: 0.109
Episode 1200    Average Score: 0.158
Episode 1300    Average Score: 0.209
Episode 1400    Average Score: 0.258
Episode 1500    Average Score: 0.313
Episode 1600    Average Score: 0.350
Episode 1700    Average Score: 0.462
Episode 1728    Average Score: 0.504
```

```
Solved in 1628 episodes, with an average score over the last 100 episodes of:
0.50
```

[19]:
```python
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# Plot scores of the learning trajectory
plt.plot(np.arange(1, len(scores)+1), scores, label="Score per episode",␣
 ↪c='gray')
plt.ylabel('Score')
plt.xlabel('Episode')
plt.plot(pd.Series(scores).rolling(100).mean(), label="Score avg last 100␣
 ↪episodes", c='blue')
plt.axhline(0.05, c='g', ls = '--', label='Required avg score over 100␣
 ↪episodes')
plt.legend(loc='upper left', bbox_to_anchor=(0, 1.3))
plt.show()
```



1.0.9 Watch your Trained Agent play

[20]:
```
# loading model - agent 0
actor_state_dict_0 = torch.load('checkpoint_actor_0.pth')
print(actor_state_dict_0.keys())
m_agents.ma[0].actor_local.load_state_dict(actor_state_dict_0)

critic_state_dict_0 = torch.load('checkpoint_critic_0.pth')
print(critic_state_dict_0.keys())
m_agents.ma[0].critic_local.load_state_dict(critic_state_dict_0)
```

```
odict_keys(['fc1.weight', 'fc1.bias', 'bn1.weight', 'bn1.bias',
'bn1.running_mean', 'bn1.running_var', 'fc2.weight', 'fc2.bias', 'fc3.weight',
'fc3.bias'])
odict_keys(['fcs1.weight', 'fcs1.bias', 'bn1.weight', 'bn1.bias',
'bn1.running_mean', 'bn1.running_var', 'fc2.weight', 'fc2.bias', 'fc3.weight',
'fc3.bias'])
```

[21]:
```
# loading model - agent 1
actor_state_dict_1 = torch.load('checkpoint_actor_1.pth')
print(actor_state_dict_1.keys())
m_agents.ma[1].actor_local.load_state_dict(actor_state_dict_1)

critic_state_dict_1 = torch.load('checkpoint_critic_1.pth')
print(critic_state_dict_1.keys())
m_agents.ma[1].critic_local.load_state_dict(critic_state_dict_1)
```

```
odict_keys(['fc1.weight', 'fc1.bias', 'bn1.weight', 'bn1.bias',
'bn1.running_mean', 'bn1.running_var', 'fc2.weight', 'fc2.bias', 'fc3.weight',
'fc3.bias'])
odict_keys(['fcs1.weight', 'fcs1.bias', 'bn1.weight', 'bn1.bias',
'bn1.running_mean', 'bn1.running_var', 'fc2.weight', 'fc2.bias', 'fc3.weight',
'fc3.bias'])
```

[22]:
```
for i in range(1, 6):                                      # play game for 5␣
 ↪episodes
    env_info = env.reset(train_mode=False)[brain_name]     # reset the␣
 ↪environment
    states = env_info.vector_observations                  # get the current␣
 ↪state (for each agent)
    scores = np.zeros(num_agents)                          # initialize the␣
 ↪score (for each agent)
    while True:
        actions = m_agents.act(states, add_noise=False) # select an action (for␣
 ↪each agent)
        env_info = env.step(actions)[brain_name]           # send all actions␣
 ↪to the environment
```

```
        next_states = env_info.vector_observations         # get next state␣
↪(for each agent)
        rewards = env_info.rewards                         # get reward (for␣
↪each agent)
        dones = env_info.local_done                        # see if episode␣
↪finished
        scores += env_info.rewards                         # update the score␣
↪(for each agent)
        states = next_states                               # roll over states␣
↪to next time step
        if np.any(dones):                                  # exit loop if␣
↪episode finished
            break
    print('Score (max over agents) from episode {}: {}'.format(i, np.
↪max(scores)))
```

```
Score (max over agents) from episode 1: 2.600000038743019
Score (max over agents) from episode 2: 0.10000000149011612
Score (max over agents) from episode 3: 2.600000038743019
Score (max over agents) from episode 4: 0.6000000089406967
Score (max over agents) from episode 5: 0.10000000149011612
```

When finished, you can close the environment.

[23]:
```
env.close()
```

1.0.10 Ideas for Future Work

- Test implementations of other modern multi-agent approaches, e.g.
  - MAA2C
  - IQL
  - IDDPG
  - IPPO
- Tune hyperparameters to accelerate training