# Collaboration-Competition - Multi-Agent DDPG Algorithm With The Unity Tennis Environment

**Howard Hyunjin Cho**

## ABSTRACT

This project explored deep reinforcement learning methods for multi-agent systems. Multi-agent systems are present everywhere around us such as autonomous car is driving to office and the car should considering other cars. The concept of multi-agent system is where multi agents interact with one another. Agents may or may not know everything about all the others in the systems. Therefore, this project MADDPG algorithm helps to solve the different kinds of interactions going on between agents to work in complex environments. In the Unity tennis environment, the MADDPG algorithm is used to train the two agents.

Keywords: MADDPG, DDPG, DQN, model-free, off-policy, replay buffer, soft updates, target networks

## INTRODUCTION

MADDPG is a multi-agent deep deterministic policy gradients, a model-free, off-policy, policy gradient-based algorithm that uses two separate deep neural networks (one actor, one critic) to both explore the stochastic environment and, separately, learn the best policy to achieve maximum reward. Deep deterministic policy gradients(DDPG). MADDPG is fit to train multi-agent since DDPG is off-policy and model-free actor-critic methods. In DDPG, it is used two deep neural networks actor and critic. In Policy-based method, agents have high variance where agent can only learn a new data or on new observations. Off-policy are used so the agent can learn from historical data which in this case, experience replay memory is used. The model-free is directly taking a data from environment, as opposed to making its own prediction about the environment. The actor-critic methods are useful in DDPG. All we are trying to do in actor-critic method is to continue to reduce the high variance commonly associated with policy-based agents. By utilizing actor-critic with experience replay is prevalent in continuous control tasks.

## BACKGROUND

In DDPG, we want the believed best action every single time we query the actor network. That is a deterministic policy. As shown in Figure1, the actor is basically learning the argmax a Q(S,a), which is the best action. Actor in DDPG is used to approximate the optimal policy deterministically. Therefore, output the best believed action for any given state. The critic learns to evaluate the optimal action value function by using the actors best believed action. Therefore, we use the actor which is an approximate maximize to calculate a new target value for training the action value function much in the way DQN does.

DDPG uses replay buffer and soft updates to the target networks. In DDPG, you have two copies of network weights for each network which are a regular for the actor, an irregular for the critic and a target for the actor and a target for the critic. The target networks are updated using a oft updates strategy. As shown in Figure 2, a soft update strategy consists of slowly blending the regular network weights with the target network weights. Therefore, every time step the target network be 99.99% of the target network weights and only a 0.01% of regular network weights.

In figure 3 Algorithm1 is about DDPG algorithm. In the first line initialize critic network with given weight Q network($\theta^{Q}$) and actor with given weight deterministic policy function $\theta^{\mu}$. Second line use weight of target Q network, $\theta^{Q'}$ and weight of target policy network, $\theta^{\mu'}$ to do off-policy updates. The

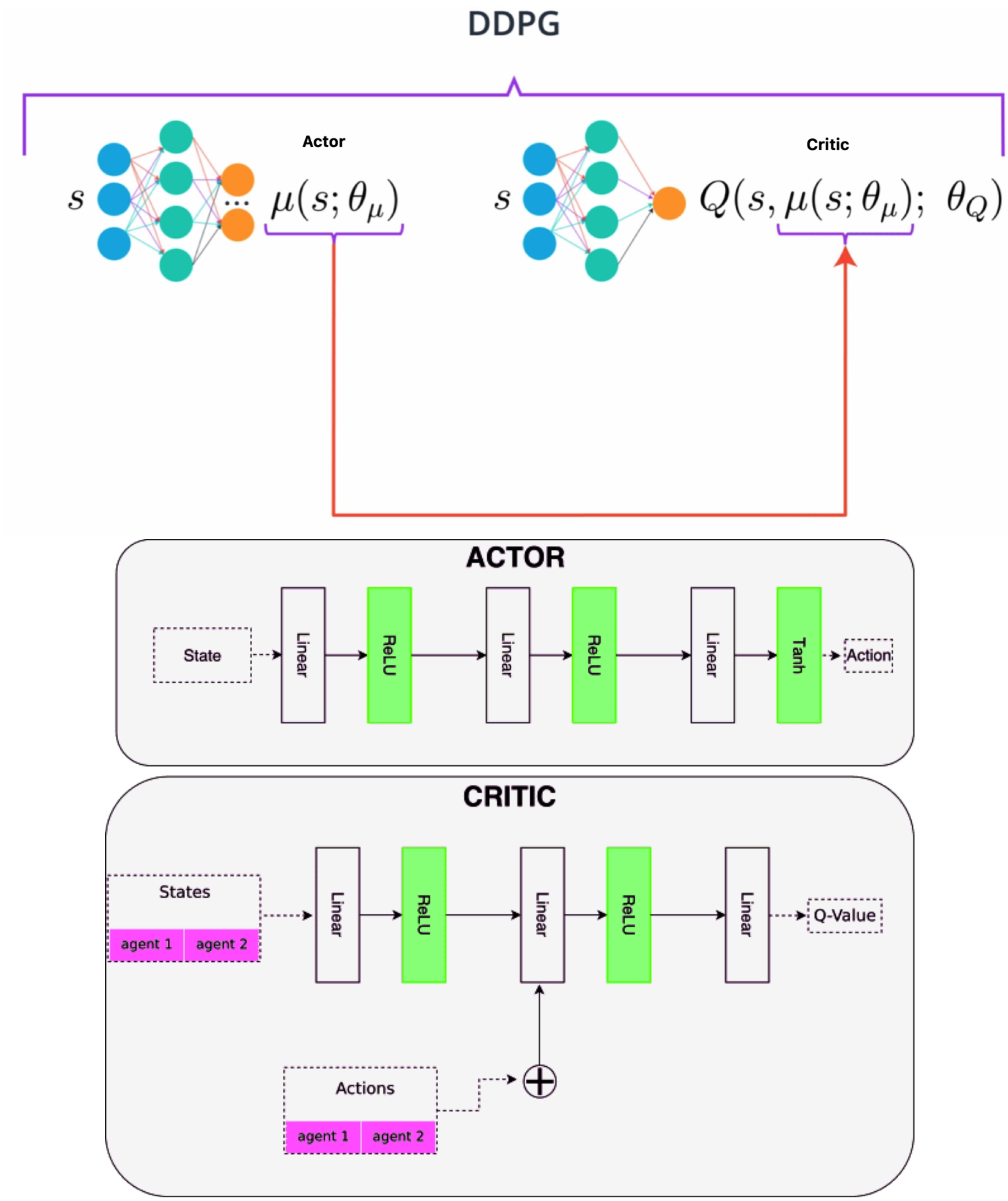


**Figure 1.** Actor-Critic Interaction

action-value function $Q^\pi(s,a)$(Q-function) can be described as recursive format by Bellman equation:

$$Q^\pi(s,a) = \underset{r,\ s'\sim E}{\mathbb{E}}[r(s,a) + \gamma \underset{a'\sim\pi}{\mathbb{E}}[Q^\pi(s',a')]]$$

If the target policy is deterministic we can describe it as a function $\mu$ : S $\leftarrow$ A and avoid the inner expectation:

$$Q^\mu(s,a) = \underset{r,\ s'\sim E}{\mathbb{E}}[r(s,a) + \gamma Q^\mu(s',\mu(s'))]$$

The expectation depends only on the environment. This means that it is possible to learn $Q^\mu$ off-policy, using transitions which are generated from a different stochastic behavior policy $\beta$.

$$\mu(s) = argmax_a Q(s,a)$$

The first code is shown at Listing 1 Actor and Critic networks are for the deterministic policy network and the Q network. During the implementation, ReLU activation is used.

$$f(a,x) = \begin{cases} 0 & \text{for x} < 0 \\ x & \text{for x} \geq 0 \end{cases}$$

We consider function approximators parameterized by $\theta^Q$, which we optimize by minimizing the loss:

$$L(\theta^Q) = \mathop{\mathbb{E}}_{s \sim p^\beta,\ a_t \sim \beta,\ r_t \sim E}[(Q(s,a|\theta^Q) - y_t)^2]$$

where Q target,

$$y_t = r(s,a) + \gamma Q(s', \mu(s')|\theta^Q)$$

To calculate $y_t$ it is necessary to use of a replay buffer and a separate target network. Replay buffer used in many other reinforcement learning algorithms to sample experience to update the parameters. The replay buffer contains a collection of experience tuples (S, A, R, S'). The tuples are gradually added to the buffer as we are interacting with environment. The replay buffer is used to break the correlation between immediate transitions in the episodes. The code indicates in the second code at Listing 2 Replay Buffer. DDPG optimizes the critic by minimizing the loss:

$$L(\theta^Q) = \frac{1}{N}\sum_i (y_i - (Q(s,a|\theta^Q))^2$$

Adding noise $N$ can overcome deterministic policy gradient to explore the full state and action space:

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

Code shows at Listing 3 OUNoise. In the deterministic policy gradient, we want to maximize the rewards (Q-values) received over the sampled mini-batch where gradient is given as:

$$\bigtriangledown_{\theta^\mu} J \approx \mathop{\mathbb{E}}_{s \sim p^\beta}[\bigtriangledown_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s,\ a=\mu(s|\theta^\mu)}]$$

by applying chain rule:

$$\bigtriangledown_{\theta^\mu} J \approx \mathop{\mathbb{E}}_{s \sim p^\beta}[\bigtriangledown_a Q(s,a|\theta^Q)|_{s=s,\ a=\mu(s)} \bigtriangledown_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s}]$$

In the last line in Figure 3, the target networks are slowly updated(soft updates for both actor and critic. The target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen some period of time. This above processes of the agent is indicated in code at the Listing 4 Agent.

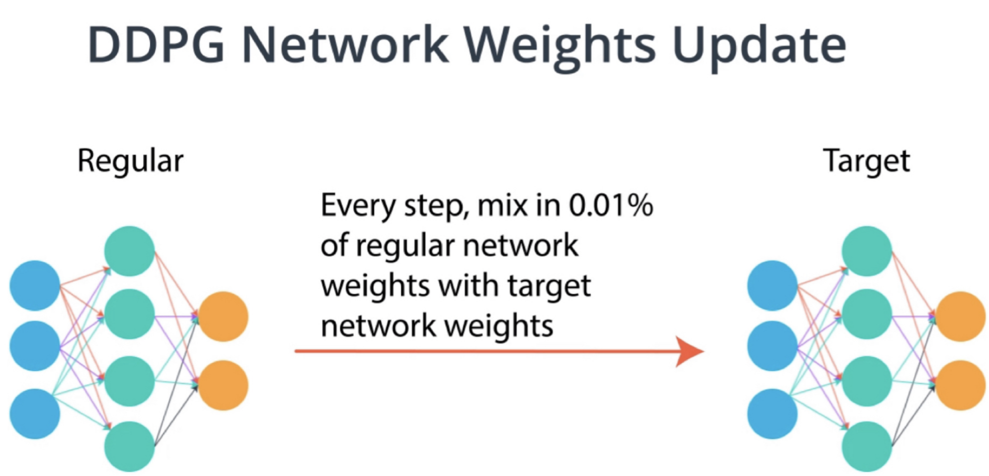


**Figure 2.** regular target

**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 Pseudocode

## MADDPG METHODS

Having multiple agent in a system brings in a few benefits. The agents can share their experiences with one another making each other. When multi-agent systems used reinforcement learning techniques to train the agents and make them learn their behaviors. MADDPG is the multi-agent counterpart of DDPG based on the actor critic methods. In the MADDPG, multiple agents works with their own actor and critic networks.

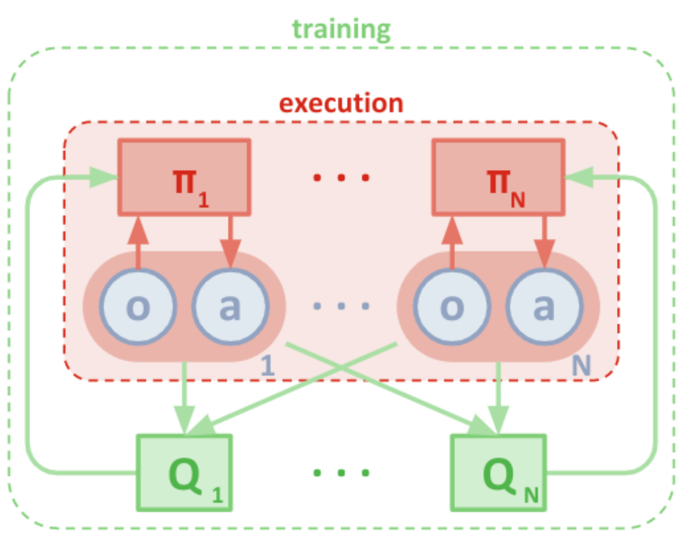


**Figure 4.** MADDPG Train

In Figure 5 below, it shows MADDPG psudocode

**Multi-Agent Deep Deterministic Policy Gradient Algorithm**

For completeness, we provide the MADDPG algorithm below.

---

**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 5.** MADDPG Pseudocode

## RESULT

As shown in Figure 6, the scores reach 0.5 to around 16 minutes. Figure 6 plot for Hyperparameters:

| | |
|---|---|
| Buffer Size | 1e6 |
| Batch Size | 384 |
| GAMMA | 0.99 |
| TAU | 1e-3 |
| LR ACTOR | 1e-4 |
| LR CRITIC | 1e-3 |
| WEIGHT DECAY | 0 |
| REWARD STEPS | 4 |

## IMPROVEMENT SUGGESTION

I would like to twig the hyperparameter to fit better model and implement D4PG and A2C. Also A3C to see how the performance different compare to D4PG and DDPG.

## REFERENCES

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments.

Lillicrap et al. (2015) Lowe et al. (2017)

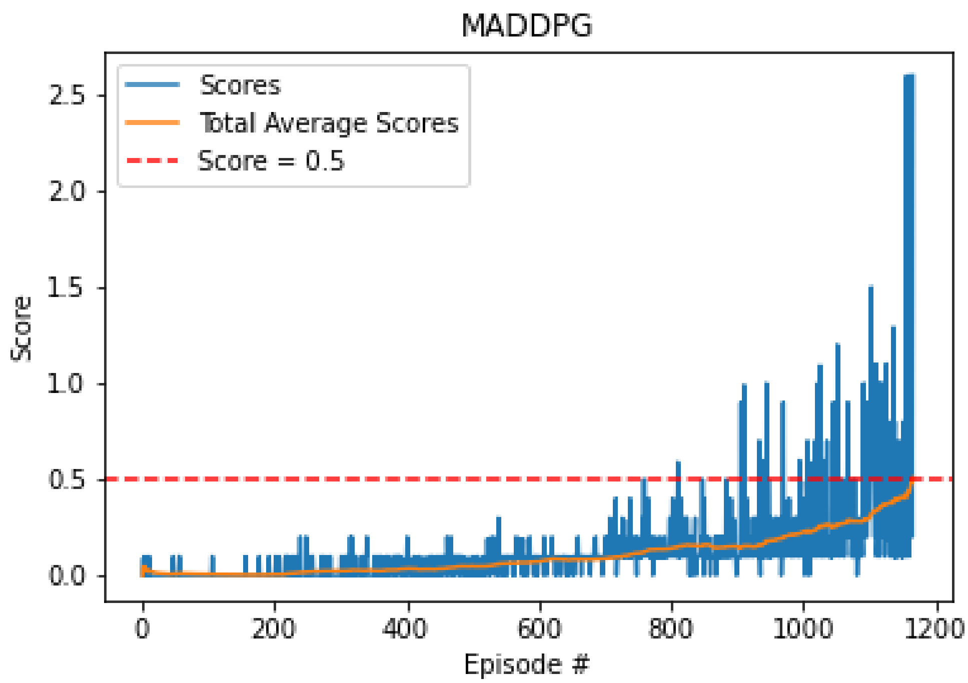


**Figure 6.** MADDPG Plot

```python
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=512, fc2_units=256):
        """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)#.to(ddpg_agent.device)
        # self.bn1 = nn.BatchNorm1d(fc1_units)#.to(ddpg_agent.device) # not much change.
        self.fc2 = nn.Linear(fc1_units, fc2_units)#.to(ddpg_agent.device)
        self.fc3 = nn.Linear(fc2_units, action_size)#.to(ddpg_agent.device)
        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.bn1(self.fc1(state)))
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        # x = F.selu(self.fc1(state))
        # x = F.selu(self.fc2(x))
        return torch.tanh(self.fc3(x))


class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=512, fc2_units=256):
        """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 the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
```

```python
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.fc1 = nn.Linear(state_size, fc1_units)#.to(ddpg_agent.device)
        self.bn1 = nn.BatchNorm1d(fc1_units)#.to(ddpg_agent.device)
        self.fc2 = nn.Linear(fc1_units+action_size, fc2_units)#.to(ddpg_agent.device)
        self.fc3 = nn.Linear(fc2_units, 1)#.to(ddpg_agent.device)
        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, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        # xs = F.relu(self.fc1(state))
        xs = F.relu(self.bn1(self.fc1(state)))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        # xs = F.selu(self.fc1(state))
        # x = torch.cat((xs, action), dim=1)
        # x = F.selu(self.fc2(x))
        return self.fc3(x)
```

Listing 1: Actor and Critic networks

```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)
```

```python

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).float().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

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

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

Listing 2: Replay Buffer

```python

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.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.array([np.random.randn() for i in range(len(x))])
        self.state = x + dx
        return self.state
```

Listing 3: OUNoise

```python

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, 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
            #
            num_agents (int): number of agents where effects on the critic
        """
        self.state_size = state_size
        self.action_size = action_size
        # self.num_agents = num_agents
        self.seed = random.seed(random_seed)

        # 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)
        # add num of agents effects
        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)

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

        # Replay memory
        self.memory = ReplayBuffer(BUFFER_SIZE, BATCH_SIZE, random_seed)


    def step(self, states, actions, rewards, next_states, dones, t_step, num_learn=4):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward
        # collect multiple agent to learn
        for state, action, reward, next_state, done in zip(states, actions, rewards, next_states, dones):
            self.memory.add(state, action, reward, next_state, done)
        # self.memory.add(states, actions, rewards, next_states, dones)

        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE and t_step%num_learn == 0:

            experiences = self.memory.sample()
            for _ in range(num_learn):
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

```

```python
    def act(self, state, add_noise=True):
        """Returns actions for given state as per current policy."""
        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 += self.noise.sample()
        return np.clip(action, -1, 1)

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

    def learn(self, experiences, gamma):
        """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




        # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        next_actions = self.actor_target(next_states)
        Q_targets_next = self.critic_target(next_states, next_actions)

        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Compute critic loss
        Q_expected = self.critic_local(states, actions)
        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 ---------------------------- #
```

```
        # Compute actor loss
        pred_actions = self.actor_local(states)
        actor_loss = -self.critic_local(states, pred_actions).mean() #make sure use negative

        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

    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)
```

Listing 4: Agent