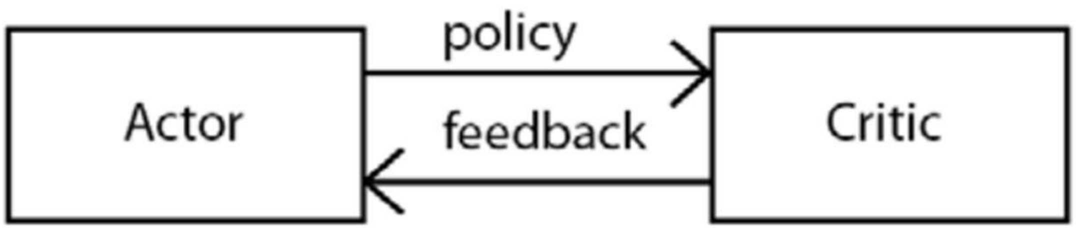
Actor-Critic Methods

Overview

- The actor-critic method lies in the intersection of value-based and policy-based methods.

- The actor-critic method uses two types of neural network
  - The actor network
    - A policy network
    - Finds an optimal policy
  - The critic network
    - A value network (estimates state value)
    - Evaluates the policy produced by the actor network



Overview

- Relation to REINFORCE with baseline
  - In REINFORCE with baseline, we had a value network that estimates state values.
  - The state values were used as baseline for reducing variance of policy gradient.

- In the actor-critic method, the critic network reduces variance of the gradients as well, but it also helps to improve the policy iteratively in an online fashion.

Understanding Actor-Critic Methods

- In REINFORCE with baseline, the network parameters were updated at the end of an episode.
- In the actor-critic method, parameters are updated at every step of the episode.
- REINFORCE with baseline
  - We generate N trajectories using policy π_θ and compute gradient as:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)(R_t - V(s_t))\right]$$

  - In order to obtain R_t, we need a complete trajectory.
    - It is similar to Monte Carlo methods, in which we needed the whole trajectory to update parameters.

Understanding Actor-Critic Methods

- Instead of generating the complete trajectory, we would like to make use of bootstrapping, as in TD learning.
- In the actor-critic method, we approximate the return by taking the immediate reward and the discounted value of the next state.

$$R \approx r + \gamma V(s')$$

- Using this, we can rewrite the policy gradient as follows.

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta (a_t | s_t)\big(r + \gamma V_\phi(s_t') - V_\phi(s_t)\big)$$

- With this gradient, we can update the parameters of the actor network at every step of the episode.

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

Understanding Actor-Critic Methods

- Similar to the actor network, we can update the parameter of the critic network at every step of the episode.
- The loss of the critic network is the mean squared error between target value and the predicted value.

$$J(\phi) = \left(r + \gamma V_\phi(s_t') - V_\phi(s_t)\right)^2$$

- With this loss function, we compute the gradients and update parameter ϕ of the critic network using gradient descent.

$$\phi = \phi - \alpha \nabla_\phi J(\phi)$$

The Actor-Critic Algorithm

1. Initialize the actor network parameter θ and the critic network parameter ϕ
2. For N number of episodes, repeat *step 3*
3. For each step in the episode, that is, for $t = 0, \ldots, T-1$:
    1. Select an action using the policy, $a_t \sim \pi_\theta(s_t)$
    2. Take the action a_t in the state s_t, observe the reward r, and move to the next state s'_t
    3. Compute the policy gradients:
    $$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta (a_t|s_t)\left(r + \gamma V_\phi(s'_t) - V_\phi(s_t)\right)$$
    4. Update the actor network parameter θ using gradient ascent:
    $$\theta = \theta + \alpha \nabla_\theta J(\theta)$$
    5. Compute the loss of the critic network:
    $$J(\phi) = \left(r + \gamma V_\phi(s'_t) - V_\phi(s_t)\right)^2$$
    6. Compute gradients $\nabla_\phi J(\phi)$ and update the critic network parameter ϕ using gradient descent:
    $$\phi = \phi - \alpha \nabla_\phi J(\phi)$$

Advantage Actor-Critic (A2C)

- The actor-critic algorithm in the previous slide is also referred to as the Advantage Actor-Critic (A2C).

- Advantage function is the difference between the Q function and the value function.

$$A(s, a) = Q(s, a) - V(s)$$

  - It is an indication of how good an action a is compared to average actions.

- Now, what is $Q(s, a)$? It is the expected return of the episode when we start from state s and takes the action a in the state. That is the same as R_t. Thus,

$$Q(s, a) \approx r + \gamma V(s')$$

$$A(s, a) = r + \gamma V(s') - V(s)$$

- So essentially our policy gradient is computed using the advantage function.

Asynchronous Advantage Actor-Critic (A3C)

- Asynchronous Advantage Actor-Critic (A3C) is an advanced version of actor-critic methods.
- It uses multiple agents for learning in parallel aggregates their overall experience.

- In A3C, there are two types of networks
  - global network (global agent)
  - worker network (worker agent)

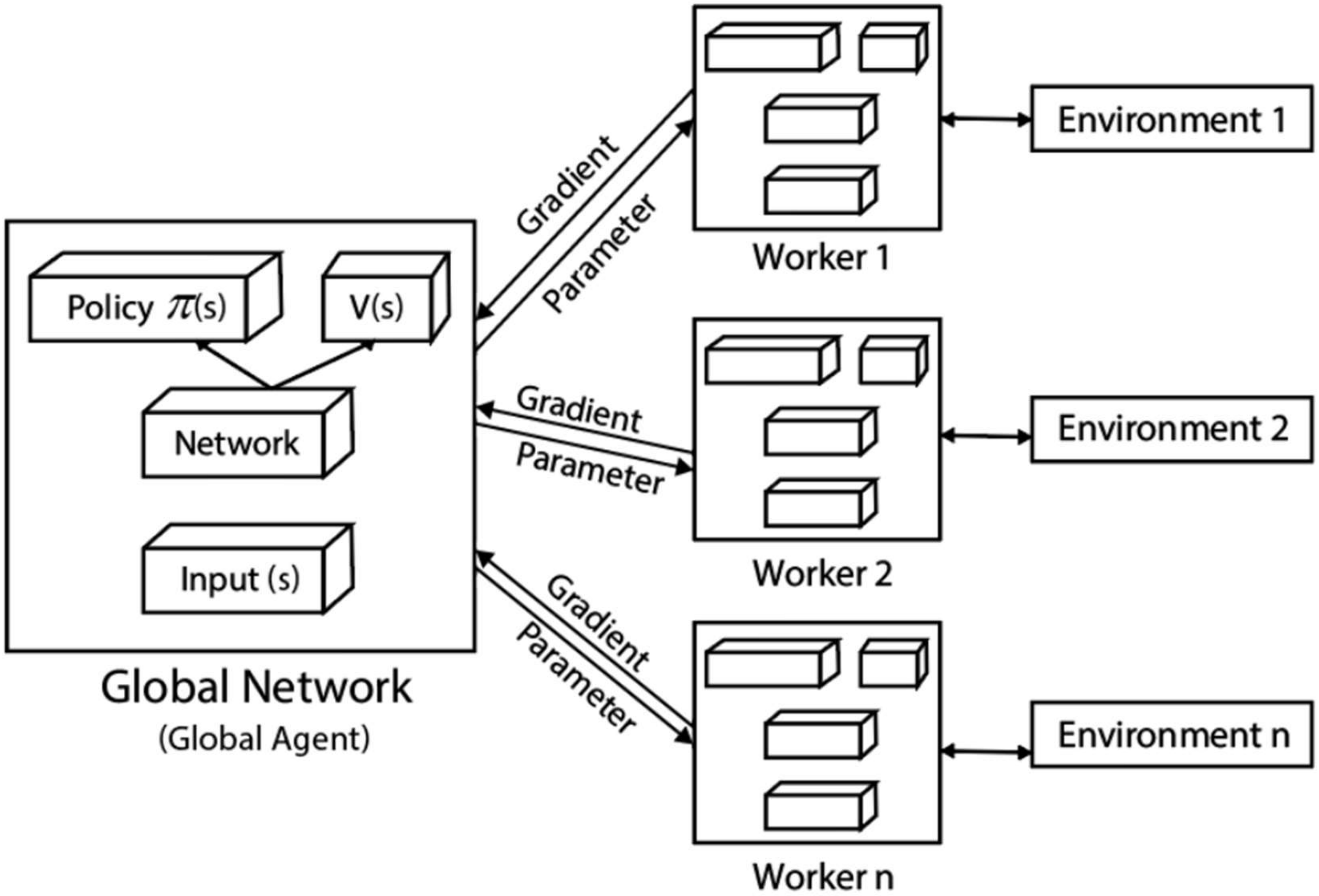
- Many worker agents are used, and each worker uses a different exploration policy to collect experience.
- The experience of the workers are aggregated and sent to the global agent who aggregates the learning.

Asynchronous Advantage Actor-Critic (A3C)

- asynchronous
  - Instead of having a single agent, multiple agents interact with the enrivonment.
  - We provide copies of the environment to every agent so that each agent can interact with its own copy of the environment.
  - The workers interact with the environment asynchronously, and report to the global agent asynchronously.
- advantage
  - Advantage function is used to calculate policy gradient.
- actor-critic
  - Each agent consists of an actor network for estimating the policy and the critic network for evaluating the policy produced by the actor network.

Asynchronous Advantage Actor-Critic (A3C)

- Architecture of A3C



Asynchronous Advantage Actor-Critic (A3C)

- Each worker agent interacts with its own copy of the environment.
- A worker agent computes the actor network loss (policy loss) and the critic network loss (value loss).
- The actor loss:

$$J(\theta) = \log \pi_\theta (a_t|s_t)\big(r + \gamma V_\phi(s_t') - V_\phi(s_t)\big)$$

- We add a new term to our actor loss called the entropy (measure of randomness) of the policy.

$$J(\theta) = \log \pi_\theta (a_t|s_t)\big(r + \gamma V_\phi(s_t') - V_\phi(s_t)\big) + \beta H(\pi(s))$$

- Adding the entropy of the policy promotes exploration.
- $H(\pi)$ denotes the entropy of the policy.
- Parameter β is used to control the significance of the entropy.

- The critic loss is just the mean squared error.

Asynchronous Advantage Actor-Critic (A3C)

- After computing the policy loss and the value loss, worker agents compute the gradients of the losses and send the gradients to the global agent.
- The gradients are asynchronously accumulated at the global agent.
- The global agent updates their parameters using the received gradients.
- The global agent sends the updated parameter periodically to the worker agents.
- The worker agents update their parameters as given by the global agent.

Asynchronous Advantage Actor-Critic (A3C)

- Summary

1. The worker agent interacts with their own copies of the environment.
2. Each worker follows a different policy and collects the experience.
3. Next, the worker agents compute the losses of the actor and critic networks.
4. After computing the loss, they calculate gradients of the loss, and send those gradients to the global agent asynchronously.
5. The global agent updates their parameters with the gradients received from the worker agents.
6. Now, the updated parameter from the global agent will be sent to the worker agents periodically.

Cart Pole with Actor-Critic Method (A2C) [ex018]

- Library imports

```
import gym
import torch
import torch.nn as nn
from itertools import count
from torch.distributions import Bernoulli
import numpy as np
import torch.nn.functional as F
▶ Launch TensorBoard Session
from tensorboardX import SummaryWriter
from collections import deque
import random
```

- We use gpu if gpu is available. Otherwise, we use cpu.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Solving Cart Pole with Actor-Critic Method (A2C)

- The policy network (Actor)

```python
class PolicyNetwork(nn.Module):
    def __init__(self):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(4, 64)
        self.fc2 = nn.Linear(64, 128)
        self.fc3 = nn.Linear(128, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

    def select_action(self, state):
        with torch.no_grad():
            prob = self.forward(state)
            b = Bernoulli(prob)
            action = b.sample()
        return action.item()
```

Solving Cart Pole with Actor-Critic Method (A2C)

- The value network (Critic)

```
class ValueNetwork(nn.Module):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(4, 64)
        self.fc2 = nn.Linear(64, 256)
        self.fc3 = nn.Linear(256, 1)

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

Solving Cart Pole with Actor-Critic Method (A2C)

- The replay buffer

```python
class Memory(object):
    def __init__(self, memory_size: int) -> None:
        self.memory_size = memory_size
        self.buffer = deque(maxlen=self.memory_size)

    def add(self, experience) -> None:
        self.buffer.append(experience)

    def size(self):
        return len(self.buffer)

    def sample(self, batch_size: int, continuous: bool = True):
        if batch_size > len(self.buffer):
            batch_size = len(self.buffer)
        if continuous:
            rand = random.randint(0, len(self.buffer) - batch_size)
            return [self.buffer[i] for i in range(rand, rand + batch_size)]
        else:
            indexes = np.random.choice(np.arange(len(self.buffer)), size=batch_size, replace=False)
            return [self.buffer[i] for i in indexes]

    def clear(self):
        self.buffer.clear()
```

Solving Cart Pole with Actor-Critic Method (A2C)

- Preparation
  - create environment
  - create policy and value network
  - define optimizers for the policy and the value network
  - set hyperparameters

```
env = gym.make('CartPole-v0')
policy = PolicyNetwork().to(device)
value = ValueNetwork().to(device)
optim = torch.optim.Adam(policy.parameters(), lr=1e-4)
value_optim = torch.optim.Adam(value.parameters(), lr=3e-4)
gamma = 0.99
writer = SummaryWriter('a2c_logs')
memory = Memory(200)
batch_size = 32
is_learn = False
steps = 0
```

Solving Cart Pole with Actor-Critic Method (A2C)

- Training
  - Run an episode go through the steps
  - Add each transaction in the replay buffer

```
for epoch in range(3000):
    state = env.reset()
    episode_reward = 0

    for time_steps in range(200):
        k += 1
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(device)
        action = policy.select_action(state_tensor)
        next_state, reward, done, _ = env.step(int(action))
        episode_reward += reward
        memory.add((state, next_state, action, reward, done))
```

Solving Cart Pole with Actor-Critic Method (A2C)

- Training
  - After k steps, we update parameters of the actor and the critic

```
if k == batch_size:
    k = 0
    experiences = memory.sample(batch_size)
    batch_state, batch_next_state, batch_action, batch_reward, batch_done = zip(*experiences)
    batch_state = torch.FloatTensor(batch_state).to(device)
    batch_next_state = torch.FloatTensor(batch_next_state).to(device)
    batch_action = torch.FloatTensor(batch_action).unsqueeze(1).to(device)
    batch_reward = torch.FloatTensor(batch_reward).unsqueeze(1).to(device)
    batch_done = torch.FloatTensor(batch_done).unsqueeze(1).to(device)
    with torch.no_grad():
        value_target = batch_reward + gamma * (1 - batch_done) * value(batch_next_state)
        advantage = value_target - value(batch_state)
    prob = policy(batch_state)
    b = Bernoulli(prob)
    log_prob = b.log_prob(batch_action)
    loss = - log_prob * advantage
    loss = loss.mean()
    optim.zero_grad()
    loss.backward()
    optim.step()
    value_loss = F.mse_loss(value_target, value(batch_state))
    value_optim.zero_grad()
    value_loss.backward()
    value_optim.step()
```

Solving Cart Pole with Actor-Critic Method (A2C)

- Training
  - After every 10 epochs, we print out the reward
  - We also record the reward on the tensorboard

```
        if done:
            break
        state = next_state

    writer.add_scalar('episode reward', episode_reward, epoch)
    if epoch % 10 == 0:
        print('Epoch:{}, episode reward is {}'.format(epoch, episode_reward))
        #torch.save(policy.state_dict(), 'a2c-policy.para')
```

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG)

- DDPG is an off-policy actor-critic methods, designed for environments where the action space is continuous.

- The difference between DDPG and the actor-critic algorithms from the previous chapter (A2C and A3C) is that DDPG tries to learn a deterministic policy instead of a stochastic policy.
- Deterministic vs. Stochastic policy
  - In a deterministic policy, an agent always performs the same action in a particular state.
  - A deterministic policy maps the state to one particular action
    - $a = \mu(s)$
  - With a stochastic policy, the agent performs different action each time in a particular state, based on a probability distribution over the action space.
  - A stochastic policy maps the state to the probability distribution.
    - $a = \pi(s)$

Deep Deterministic Policy Gradient (DDPG)

- Actor
  - a policy network
  - tries to learn the mapping between the state and the action
  - given the state as an input, the actor outputs an action (which is a value in the continuous action space)
  - uses the policy gradient method to learn the optimal policy that achieves the maximum return

- Critic
  - a value network
  - evaluates the action produced by the actor network using a Q function
  - uses a deep Q network (DQN) to learn the Q function

Deep Deterministic Policy Gradient (DDPG)

- How the critic network evaluates an action
  - Q function gives the expected return the agent would obtain starting from state s and performing an action a following a particular policy.
  - Given a state and an action, we obtain a Q value.
    - If the Q value is high, we can say that the action performed in the state is a good action. In other words, the expected return will be high.
    - If the Q value is low, we can say that the action performed in the state is not a good action. In other words, the expected return will be low.

Deep Deterministic Policy Gradient (DDPG)

- How the critic network evaluates an action (cont.)
  - Suppose the actor network performs action "down" in state A.
  - The critic computes the Q value of "down" in state A.
  - If the Q value is high, the critic network gives feedback to the actor network.
  - If the Q value is low, the critic network gives feedback to the actor network so that the actor tries a different action.

DDPG: The Critic Network

- Overview
  - The critic network calculates the Q value of a state-action pair.
  - We use a DQN to calculate the Q value, $Q_\theta(s, a)$.
    - θ is the parameter of the critic network.

  - The action comes from the actor network. Since we learn a deterministic policy in DDPG, the action can be denoted as $a = \mu_\phi(s)$.
    - ϕ is the parameter of the actor network.

DDPG: The Critic Network

- Training the critic network
  - The target Q value can be obtained from the Bellman equation.

$$Q^*(s,a) = r + \gamma \max_{a'} Q(s',a')$$

  - We can define "error" as the difference between the target value and the predicted value.

$$r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)$$

  - For the loss function, we use MSE between the target Q value and the predicted Q value.

$$J(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_\theta(s_i',a') - Q_\theta(s_i,a_i)\right)^2$$

서강대학교
SOGANG UNIVERSITY

DDPG: The Critic Network

- Training the critic network
  - The target Q value can be obtained from the Bellman equation.

$$Q^*(s,a) = r + \gamma \max_{a'} Q(s',a')$$

  - We can define "error" as the difference between the target value and the predicted value.

$$r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)$$

  - For the loss function, we use MSE between the target Q value and the predicted Q value.

$$J(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_\theta(s_i',a') - Q_\theta(s_i,a_i)\right)^2$$

DDPG: The Critic Network

- The problem with this approach is that both the target and the predicted Q functions are parameterized by the same parameter θ.

- We introduce another neural network to learn the target value, referred to as the target critic network.

- The main critic network θ updates parameters using gradient descent.

- The target critic network θ' updates parameters by copying the parameters of the main critic network.

- Now, the loss function of the critic network is changed to:

$$J(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a') - Q_\theta(s_i, a_i)\right)^2$$

DDPG: The Critic Network

- When computing the target value, we have a problem due to the max term.

$$J(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a') - Q_\theta(s_i, a_i)\right)^2$$

- When the action space is continuous, we cannot compute the Q value of all possible actions a' in state s'.
- To address this problem, we use a target actor network, denoted by ϕ'.
- Now, instead of selecting the action a' as the one that has the maximum Q value, we can generate an action a' using the target actor network.
  - $a' = \mu_{\phi'}(s')$

DDPG: The Critic Network

- To compute the Q value of the next state-action pair in the target critic network, we feed state s' and the action a' produced by the target actor network ϕ' to the target critic network to get the Q value of the next state-action pair.

- Now, our loss function becomes:

$$J(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma Q_{\theta'}\left(s_i', \mu_{\phi'}(s_i')\right) - Q_\theta(s_i, a_i)\right)^2$$

DDPG: The Critic Network

- With this loss function, we update the parameters θ of the main critic network.

$$\theta = \theta - \alpha \nabla_{\theta} J(\theta)$$

- To update the parameters of the target critic network θ', we use the soft replacement as the following equation.

$$\theta' = \tau\theta + (1 - \tau)\theta'$$

DDPG: The Actor Network

- The actor network is a policy network.
- It uses policy gradient to calculate the optimal policy.
- The actor network takes state s as an input and returns the action a
  - $a = \mu_\phi(s)$

- In DDPG, we use a deterministic policy. Because of that, we need to address the exploration-exploitation dilemma.
- DDPG is designed for environments with continuous action spaces. Thus, we are using a deterministic policy in the continuous action spaces.

- To explore, we modified the action by adding some noise $\mathcal{N}$ to the action produced by the actor network.
  - $a = \mu_\phi(s) + \mathcal{N}$
- The noise is generated using a process called Ornstein-Uhlenbeck random process.

DDPG: The Actor Network

- In DDPG, the goal of the actor is to select an action which gets a good feedback from the critic.
- The actor wants to maximize $Q_\theta(s, a)$, where $a = \mu_\phi(s)$.

- Thus, the objective function of the actor is:

$$J(\phi) = \frac{1}{K} \sum_i Q_\theta(s_i, a)$$

- where action $a = \mu_\phi(s_i)$.
- To maximize the objective function, we perform gradient ascent.

$$\phi = \phi + \alpha \nabla_\phi J(\phi)$$

DDPG: The Actor Network

- We used a target actor network in order to calculate the target value of the critic network.

- To update the target actor network, we use the soft replacement as we did for the target critic network.

$$\phi' = \tau\phi + (1 - \tau)\phi'$$

DDPG: Putting It All Together

- In DDPG, we use four neural networks.
  - The main critic network θ
  - The target critic network θ'
  - The main actor network ϕ
  - The target actor network ϕ'

- First, we initialize θ and ϕ with random values.
- Then, we copy θ to θ', and ϕ to ϕ'.
- We initialize the replay buffer $\mathcal{D}$.

DDPG: Putting It All Together

- For each step in the episode, we first select an action using the actor network.

$$a = \mu_\phi(s)$$

- Instead of using the action a directly, we add some noise for exploration.

$$a = \mu_\phi(s) + \mathcal{N}$$

- Now we perform action a, move to the next state s', and get reward r. We store this transition information in the replay buffer $\mathcal{D}$.

DDPG: Putting It All Together

- Next, we randomly sample a minibatch of K transitions (s, a, r, s') from the replay buffer.
- We compute the loss of the critic network.

$$J(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma Q_{\theta'}\left(s_i', \mu_{\phi'}(s_i')\right) - Q_\theta(s_i, a_i)\right)^2$$

  - To get $\mu_{\phi'}(s_i')$, we need to use the target actor network.
  - To calculate $Q_{\theta'}\left(s_i', \mu_{\phi'}(s_i')\right)$, we need to use the target critic network.
  - To calculate $Q_\theta(s_i, a_i)$, we need to use the main critic network.

- After computing the loss, we compute the gradient and update the critic network θ using gradient descent.

$$\theta = \theta - \alpha \nabla_\theta J(\theta)$$

DDPG: Putting It All Together

- Then, we compute the loss of the actor network.

$$J(\phi) = \frac{1}{K}\sum_{i} Q_\theta(s_i, a)$$

- For this loss function, we are only using the state s_i from the sampled transition (s, a, r, s'). The action a is selected by the main actor network.

- To calculate a, we need to use the main actor network.
- To calculate $Q_\theta(s_i, a)$, we need to use the main critic network.

- We maximize the objective function by calculating the gradient and performing gradient ascent.

$$\phi = \phi + \alpha \nabla_\phi J(\phi)$$

DDPG: Putting It All Together

- In the final step, we update the parameters of the target critic network θ' and the target actor network ϕ' by soft replacement.

$$\theta' = \tau\theta + (1 - \tau)\theta'$$

$$\phi' = \tau\phi + (1 - \tau)\phi'$$

DDPG: The Algorithm

1. Initialize the main critic network parameter θ and the main actor network parameter ϕ
2. Initialize the target critic network parameter θ' by just copying the main critic network parameter θ
3. Initialize the target actor network parameter ϕ' by just copying the main actor network parameter ϕ'
4. Initialize the replay buffer $\mathcal{D}$
5. For N number of episodes, repeat steps 6 and 7
6. Initialize an Ornstein-Uhlenbeck random process $\mathcal{N}$ for an action space exploration

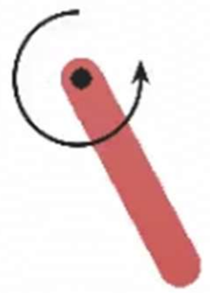
DDPG: The Algorithm (cont.)

7. For each step in the episode, that is, for $t = 0,\ldots,T-1$:

    1. Select action a based on the policy $\mu_\phi(s)$ and exploration noise, that is, $a = \mu_\phi(s) + \mathcal{N}$.
    2. Perform the selected action a, move to the next state s', get the reward r, and store this transition information in the replay buffer $\mathcal{D}$.
    3. Randomly sample a minibatch of K transitions from the replay buffer $\mathcal{D}$.
    4. Compute the target value of the critic, that is, $y_i = r_i + \gamma Q_{\theta'}\left(s_i', \mu_{\phi'}(s_i')\right)$.
    5. Compute the loss of the critic network, $J(\theta) = \frac{1}{K}\sum_i \left(y_i - Q_\theta(s_i, a_i)\right)^2$.
    6. Compute the gradient of the loss $\nabla_\theta J(\theta)$ and update the critic network parameter using gradient descent, $\theta = \theta - \alpha\nabla_\theta J(\theta)$.
    7. Compute the gradient of the actor network $\nabla_\phi J(\phi)$ and update the actor network parameter by gradient ascent, $\phi = \phi + \alpha\nabla_\phi J(\phi)$.
    8. Update the target critic and target actor network parameter as $\theta' = \tau\theta + (1-\tau)\theta'$ and $\phi' = \tau\phi + (1-\tau)\phi'$.

Inverted Pendulum Swingup using DDPG [ex019]

- The Pendulum-v0 environment models an inverted pendulum swingup problem in the control literature.
- The pendulum starts in a random position, and the goal is to swing it up so it stays upright.



Solving Inverted Pendulum Swingup using DDPG

- Observation: Box(3)

Num	State	Min	Max
0	$\cos(\theta)$	-1.0	1.0
1	$\sin(\theta)$	-1.0	1.0
2	θ dot	-8	8

- Actions: Box(1)

Num	Action	Min	Max
0	Joint Effort	-2.0	2.0

- Reward
  - -(theta^2 + 0.1*theta_dt^2 + 0.001*action^2)

Solving Inverted Pendulum Swingup using DDPG

- Library imports

```
import argparse
import pickle
from collections import namedtuple
import matplotlib.pyplot as plt
import numpy as np
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Normal
```

Solving Inverted Pendulum Swingup using DDPG

- Command-line argument processing

```
parser = argparse.ArgumentParser(description='Solve the Pendulum-v0 with DDPG')
parser.add_argument(
    '--gamma', type=float, default=0.9, metavar='G', help='discount factor (default: 0.9)')
parser.add_argument('--seed', type=int, default=0, metavar='N', help='random seed (default: 0)')
parser.add_argument('--render', action='store_true', help='render the environment')
parser.add_argument(
    '--log-interval',
    type=int,
    default=10,
    metavar='N',
    help='interval between training status logs (default: 10)')
args = parser.parse_args()
```

- random number generator seeds and data structures

```
torch.manual_seed(args.seed)
np.random.seed(args.seed)

TrainingRecord = namedtuple('TrainingRecord', ['ep', 'reward'])
Transition = namedtuple('Transition', ['s', 'a', 'r', 's_'])
```

Solving Inverted Pendulum Swingup using DDPG

- The network architectures

```
class ActorNet(nn.Module):

    def __init__(self):
        super(ActorNet, self).__init__()
        self.fc = nn.Linear(3, 100)
        self.mu_head = nn.Linear(100, 1)

    def forward(self, s):
        x = F.relu(self.fc(s))
        u = 2.0 * F.tanh(self.mu_head(x))
        return u


class CriticNet(nn.Module):

    def __init__(self):
        super(CriticNet, self).__init__()
        self.fc = nn.Linear(4, 100)
        self.v_head = nn.Linear(100, 1)

    def forward(self, s, a):
        x = F.relu(self.fc(torch.cat([s, a], dim=1)))
        state_value = self.v_head(x)
        return state_value
```

Solving Inverted Pendulum Swingup using DDPG

- The replay buffer

```python
class Memory():

    data_pointer = 0
    isfull = False

    def __init__(self, capacity):
        self.memory = np.empty(capacity, dtype=object)
        self.capacity = capacity

    def update(self, transition):
        self.memory[self.data_pointer] = transition
        self.data_pointer += 1
        if self.data_pointer == self.capacity:
            self.data_pointer = 0
            self.isfull = True

    def sample(self, batch_size):
        return np.random.choice(self.memory, batch_size)
```

Solving Inverted Pendulum Swingup using DDPG

- The agent (1/3)

```python
class Agent():

    max_grad_norm = 0.5

    def __init__(self):
        self.training_step = 0
        self.var = 1.
        self.eval_cnet, self.target_cnet = CriticNet().float(), CriticNet().float()
        self.eval_anet, self.target_anet = ActorNet().float(), ActorNet().float()
        self.memory = Memory(2000)
        self.optimizer_c = optim.Adam(self.eval_cnet.parameters(), lr=1e-3)
        self.optimizer_a = optim.Adam(self.eval_anet.parameters(), lr=3e-4)

    def select_action(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0)
        mu = self.eval_anet(state)
        dist = Normal(mu, torch.tensor(self.var, dtype=torch.float))
        action = dist.sample()
        action.clamp(-2.0, 2.0)
        return (action.item(),)

    def save_param(self):
        torch.save(self.eval_anet.state_dict(), 'ddpg_anet_params.pkl')
        torch.save(self.eval_cnet.state_dict(), 'ddpg_cnet_params.pkl')

    def store_transition(self, transition):
        self.memory.update(transition)
```

Solving Inverted Pendulum Swingup using DDPG

- The agent (2/3)

```
def update(self):
    self.training_step += 1

    transitions = self.memory.sample(32)
    s = torch.tensor([t.s for t in transitions], dtype=torch.float)
    a = torch.tensor([t.a for t in transitions], dtype=torch.float).view(-1, 1)
    r = torch.tensor([t.r for t in transitions], dtype=torch.float).view(-1, 1)
    s_ = torch.tensor([t.s_ for t in transitions], dtype=torch.float)

    with torch.no_grad():
        q_target = r + args.gamma * self.target_cnet(s_, self.target_anet(s_))
    q_eval = self.eval_cnet(s, a)

    # update critic net
    self.optimizer_c.zero_grad()
    c_loss = F.smooth_l1_loss(q_eval, q_target)
    c_loss.backward()
    nn.utils.clip_grad_norm_(self.eval_cnet.parameters(), self.max_grad_norm)
    self.optimizer_c.step()
```

Solving Inverted Pendulum Swingup using DDPG

- The agent (3/3)

```python
# update actor net
self.optimizer_a.zero_grad()
a_loss = -self.eval_cnet(s, self.eval_anet(s)).mean()
a_loss.backward()
nn.utils.clip_grad_norm_(self.eval_anet.parameters(), self.max_grad_norm)
self.optimizer_a.step()

if self.training_step % 200 == 0:
    self.target_cnet.load_state_dict(self.eval_cnet.state_dict())
if self.training_step % 201 == 0:
    self.target_anet.load_state_dict(self.eval_anet.state_dict())

self.var = max(self.var * 0.999, 0.01)

return q_eval.mean().item()
```

Solving Inverted Pendulum Swingup using DDPG

- The main function (1/2)

```
def main():
    env = gym.make('Pendulum-v0')
    env.seed(args.seed)

    agent = Agent()

    training_records = []
    running_reward, running_q = -1000, 0
    for i_ep in range(1000):
        score = 0
        state = env.reset()

        for t in range(200):
            action = agent.select_action(state)
            state_, reward, done, _ = env.step(action)
            score += reward
            if args.render:
                env.render()
            agent.store_transition(Transition(state, action, (reward + 8) / 8, state_))
            state = state_
            if agent.memory.isfull:
                q = agent.update()
                running_q = 0.99 * running_q + 0.01 * q
```

Solving Inverted Pendulum Swingup using DDPG

- The main function (2/2)

```
        running_reward = running_reward * 0.9 + score * 0.1
        training_records.append(TrainingRecord(i_ep, running_reward))

        if i_ep % args.log_interval == 0:
            print('Step {}\tAverage score: {:.2f}\tAverage Q: {:.2f}'.format(
                i_ep, running_reward, running_q))
        if running_reward > -200:
            print("Solved! Running reward is now {}!".format(running_reward))
            env.close()
            agent.save_param()
            with open('ddpg_training_records.pkl', 'wb') as f:
                pickle.dump(training_records, f)
            break

    env.close()

    plt.plot([r.ep for r in training_records], [r.reward for r in training_records])
    plt.title('DDPG')
    plt.xlabel('Episode')
    plt.ylabel('Moving averaged episode reward')
    plt.savefig("ddpg.png")
```

End of Class

Questions?

Email: jso1@sogang.ac.kr