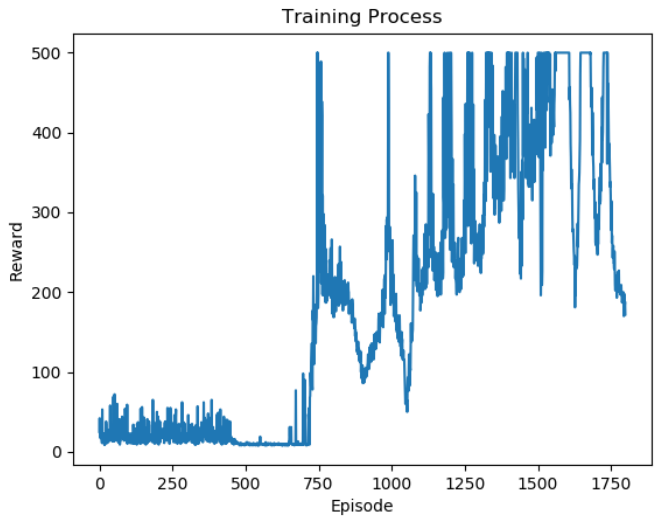
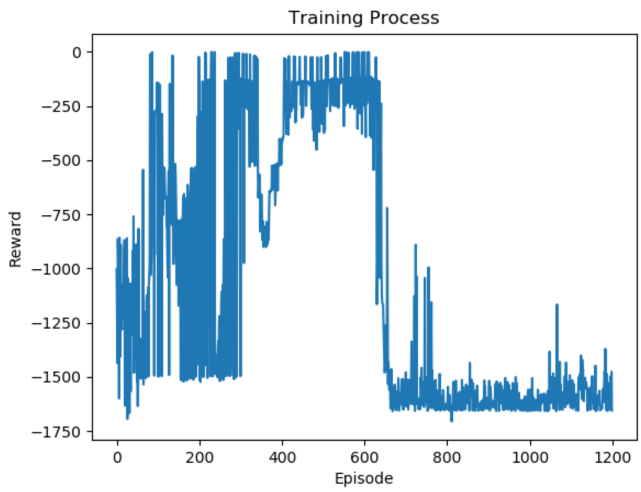
Lab 6: Deep Q-Network and Deep Deterministic Policy Gradient

Name: 鄭謝廷揚 StudentID: A073501

- **A plot shows episode rewards of at least 1000 training episodes in CartPole-v1**



- **A plot shows episode rewards of at least 1000 training episodes in Pendulum-v0**



- **Describe your major implementation of both algorithms in detail.**

Implement DQN:

Using nn.Linear to define layers. With DQN we can input a state to get a action return.

```
class DQN(nn.Module):
  def __init__(self, state_dim=4, action_dim=2, hidden_dim=24):
    super().__init__()
    self.conv1 = nn.Linear(state_dim, hidden_dim)
    self.conv2 = nn.Linear(hidden_dim, hidden_dim)
    self.conv3 = nn.Linear(hidden_dim, action_dim)
    # Called with either one element to determine next action, or a batch

  def forward(self, x):
    x = torch.nn.functional.relu(self.conv1(x))
    x = torch.nn.functional.relu(self.conv2(x))
    x = self.conv3(x)
    return x
```

Implement select_action(DQN):

Sometimes selecting action randomly

```
def select_action(epsilon, state, action_dim=2):
  """epsilon-greedy based on behavior network"""
  sample = random.random()
  if sample > epsilon:
    return behavior_net(state).max(0)[1].view(1, 1).item()
  else:
    return random.randrange(env.action_space.n)
```

Implement update network(DQN):

Zero out q_next for terminal states, since their value is only the reward, and compute with Huber loss

```
def update_behavior_network():
  def transitions_to_tensors(transitions, device=args.device):
    """convert a batch of transitions to tensors"""
    return (torch.Tensor(x).to(device) for x in zip(*transitions))

  # sample a minibatch of transitions
  transitions = memory.sample(args.batch_size)
  state, action, reward, next_state, done = transitions_to_tensors(transitions)
  q_value = behavior_net(state).gather(1, action.long())
  q_next = target_net(next_state).detach()*args.gamma+reward
  loss = F.smooth_l1_losscriterion(q_value, q_next*abs(1-done))
  # optimize
  optimizer.zero_grad()
  loss.backward()
  nn.utils.clip_grad_norm_(behavior_net.parameters(), 5)
  optimizer.step()
```

Implement ActorNet:

Using ActorNet to play an actor in training.

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=3, action_dim=1, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        h1, h2 = hidden_dim
        self.L1 = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(),
        )
        self.L2 = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
        )
        self.L3 = nn.Sequential(
            nn.Linear(h2, action_dim)
        )
        #raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        x = self.L1(x)
        x = self.L2(x)
        x = self.L3(x)
        return x
        #raise NotImplementedError
```

Implement select_action(DDPG):

Select an action by action and use exploration noise letting training process more randomly

```
def select_action(state, low=-2, high=2):
    """based on the behavior (actor) network and exploration noise"""
    random_process = OrnsteinUhlenbeckProcess()
    random_process.reset()
    action = actor_net(state)
    noise = random_process.sample()
    actionnoise = action + noise
    actionnoise = actionnoise.item()
    return max(min(actionnoise, high), low)
    #raise NotImplementedError
```

Implement update network(DDPG):

Using actor_net and target_critic_net to update the network

```
def update_behavior_network():
    def transitions_to_tensors(transitions, device=args.device):
        """convert a batch of transitions to tensors"""
        return (torch.Tensor(x).to(device) for x in zip(*transitions))

    # sample a minibatch of transitions
    transitions = memory.sample(args.batch_size)
    state, action, reward, state_next, done = transitions_to_tensors(transitions)

    ## update critic ##
    q_value = critic_net.forward(state, action)
    with torch.no_grad():
        a_next = actor_net.forward(state_next)
        q_next = target_critic_net.forward(state_next,a_next)
    q_next = (q_next*args.gamma+reward) * abs(1-done)
    critic_loss = F.smooth_l1_loss(q_value, q_next)
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()

    actor_loss = -critic_net.forward(state, actor_net.forward(state)).mean()
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

Updating target_network with behavior, target network and tau factor

```
def update_target_network(target_net, net):
    tau = args.tau
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        target.data.copy_(behavior.data * tau + target.data * (1.0 - tau))
        #raise NotImplementedError
```

Implement test function:

Running test ten times and output average reward

```
def test(env, render):
    print('Start Testing')
    seeds = (20190813 + i for i in range(10))
    result = 0.0
    total_reward = 0
    for i in range(10):
        for seed in seeds:
            total_reward = 0
            env.seed(seed)
            state = env.reset()
            done = False
            while not done:
                env.render()
                state_tensor = torch.Tensor(state).to(args.device)
                action = select_action(state_tensor)
                print(action)
                state, reward, done, info = env.step([action])
                print(state,reward)
                total_reward +=reward
        result += total_reward*0.1
        ## TODO ##

        #raise NotImplementedError
    env.close()
```

Implement optimizer:

Setting optimizer with individual learning rate.

```
    if not args.restore:
        # target network
        target_actor_net = ActorNet().to(args.device)
        target_critic_net = CriticNet().to(args.device)
        # initialize target network
        target_actor_net.load_state_dict(actor_net.state_dict())
        target_critic_net.load_state_dict(critic_net.state_dict())
        actor_opt = torch.optim.Adam(actor_net.parameters(), lr=args.lra)
        critic_opt = torch.optim.Adam(critic_net.parameters(), lr=args.lrc)
        #raise NotImplementedError
```

- ## **Describe differences between your implementation and algorithms.**

My implementation are mainly according to algorithms.

Instead of using nn.MSE as loss function, I choose nn.smooth_L1_loss as my loss function.

$$Smooth \quad L_1 = \begin{array}{l} 0.5x^2, \quad |x| < 1 \\ |x| - 0.5, \quad x < -1 \, or \, x > 1 \end{array}$$

$$Smooth \quad L_1' = \begin{array}{l} x, \quad |x| < 1 \\ -1, \quad x < -1 \\ 1, \quad x > 1 \end{array}$$

- ## **Describe your implementation and the gradient of actor updating.**

For the actor function, our objective is to maximize the expected return:

$$J(\theta) = \mathbb{E}[Q(s,a)|_{s=s_t, a_t=\mu(s_t)}]$$

Taking the derivative of the objective function with respect to the policy parameter:

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s,a) \nabla_{\theta^\mu} \mu(s|\theta^\mu)$$

Taking the mean of the sum of gradients calculated from the mini-batch:

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i [\nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}]$$

```
## update critic ##
q_value = critic_net.forward(state, action)
with torch.no_grad():
    a_next = actor_net.forward(state_next)
    q_next = target_critic_net.forward(state_next,a_next)
q_next = (q_next*args.gamma+reward) * abs(1-done)
critic_loss = F.smooth_l1_loss(q_value, q_next)
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()

actor_loss = -critic_net.forward(state, actor_net.forward(state)).mean()
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

- ## **Describe your implementation and the gradient of critic updating.**

The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

The next-state Q values are calculated with the target value network and target policy network

Then, we minimize the smooth L1 loss between the updated Q value and the original Q value

$$Loss = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

```
## update critic ##
q_value = critic_net.forward(state, action)
with torch.no_grad():
    a_next = actor_net.forward(state_next)
    q_next = target_critic_net.forward(state_next,a_next)
q_next = (q_next*args.gamma+reward) * abs(1-done)
critic_loss = F.smooth_l1_loss(q_value, q_next)
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()

actor_loss = -critic_net.forward(state, actor_net.forward(state)).mean()
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

● Results :

Result of DQN training episodes in CartPole-v1:

Test ten times and get average reward is **186**

```
Start Testing
186.0
186.0
186.0
186.0
186.0
186.0
186.0
186.0
186.0
186.0
```

Result of DDPG training episodes in Pendulum-v0 :

Test ten times and get average reward is **-5.007**

```
Step: 108400    Episode: 541    Length: 200    Total reward: -237.18[illegible]
Step: 108600    Episode: 542    Length: 200    Total reward: -128.28853966523377
Step: 108800    Episode: 543    Length: 200    Total reward: -235.84321307335924
Step: 109000    Episode: 544    Length: 200    Total reward: -129.6672317653406
Step: 109200    Episode: 545    Length: 200    Total reward: -131.40203307671578
Step: 109400    Episode: 546    Length: 200    Total reward: -128.75431354408147
Step: 109600    Episode: 547    Length: 200    Total reward: -13.816581654153186
Step: 109800    Episode: 548    Length: 200    Total reward: -123.16842299927654
Step: 110000    Episode: 549    Length: 200    Total reward: -124.36147069509961
Start Testing
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
-5.0072112485
```