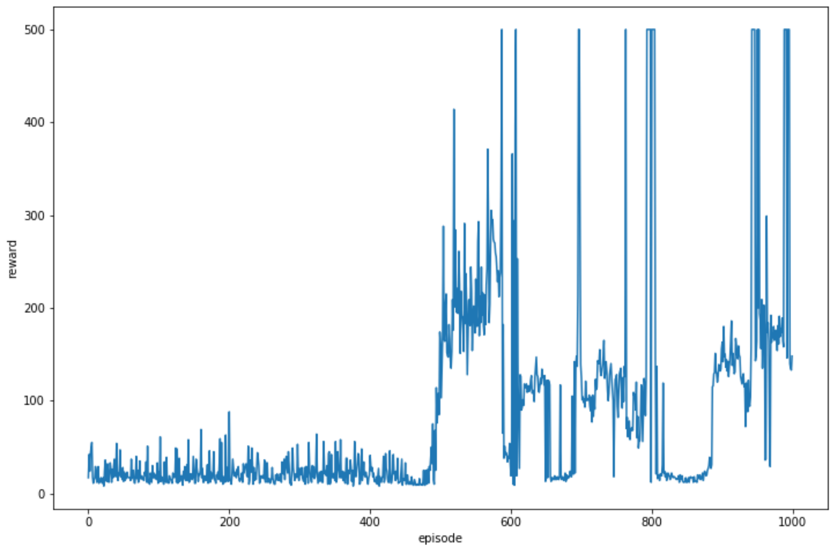
Lab 6: Deep Q-Network and Deep Deterministic Policy Gradient

Report

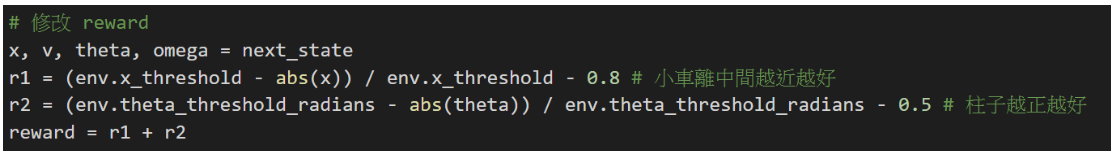
1. A plot shows episode rewards of at least 1000 training episodes in CartPole-v1 (5%)

Original reward

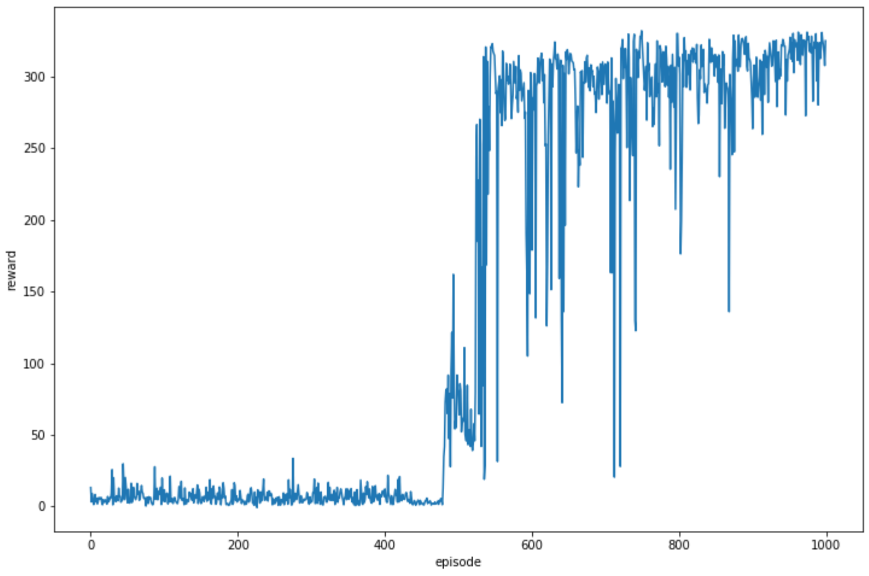


Although the reward has increased but it will be unstable when episode be longer.

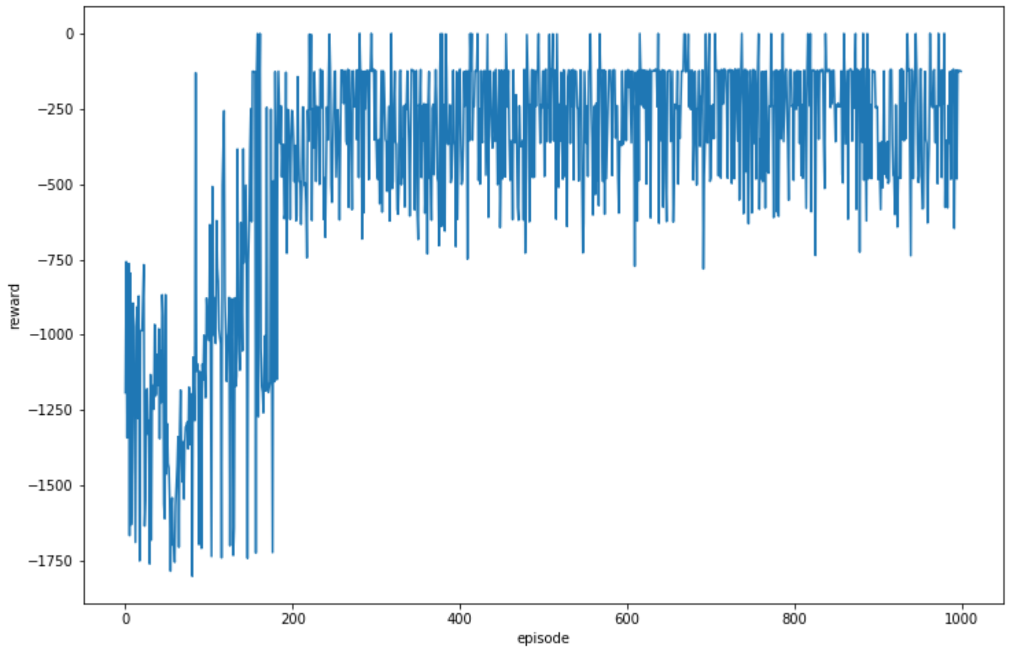
Change reward function :



```
# 修改 reward
x, v, theta, omega = next_state
r1 = (env.x_threshold - abs(x)) / env.x_threshold - 0.8 # 小車離中間越近越好
r2 = (env.theta_threshold_radians - abs(theta)) / env.theta_threshold_radians - 0.5 # 柱子越正越好
reward = r1 + r2
```

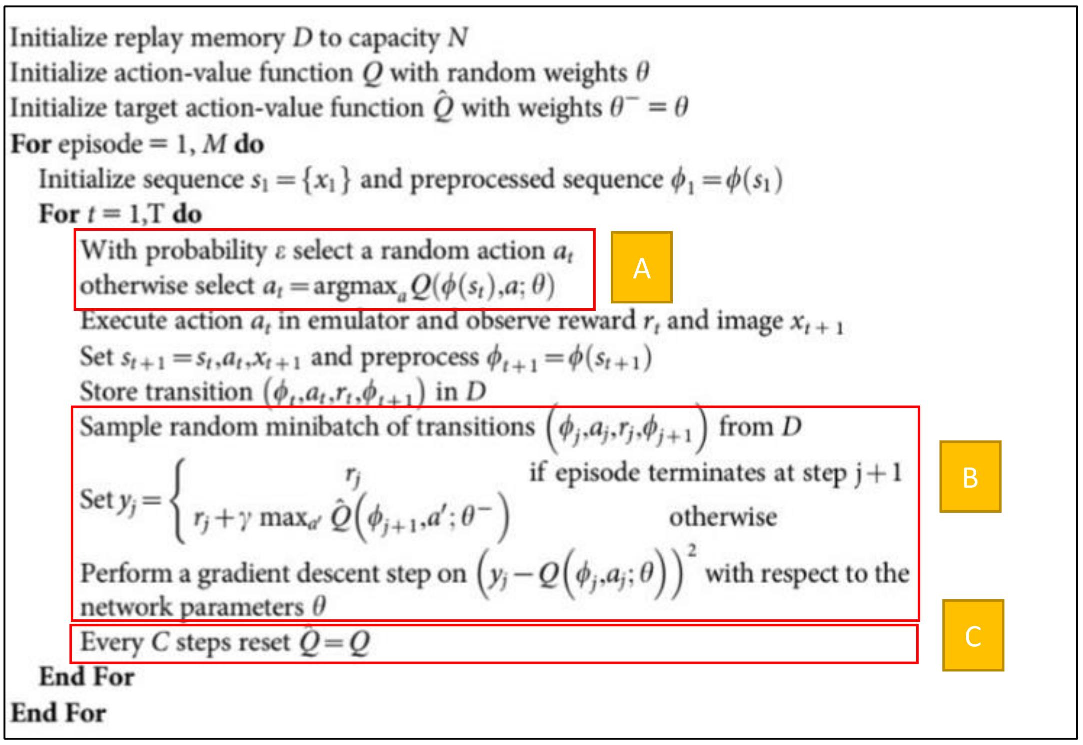


2. **A plot shows episode rewards of at least 1000 training episodes in Pendulum-v0 (5%)**



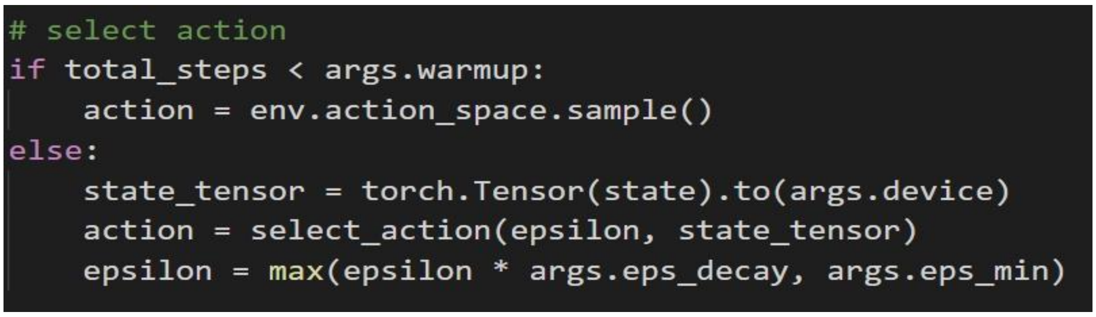
3. **Describe your major implementation of both algorithms in detail. (20%)**

DQN:



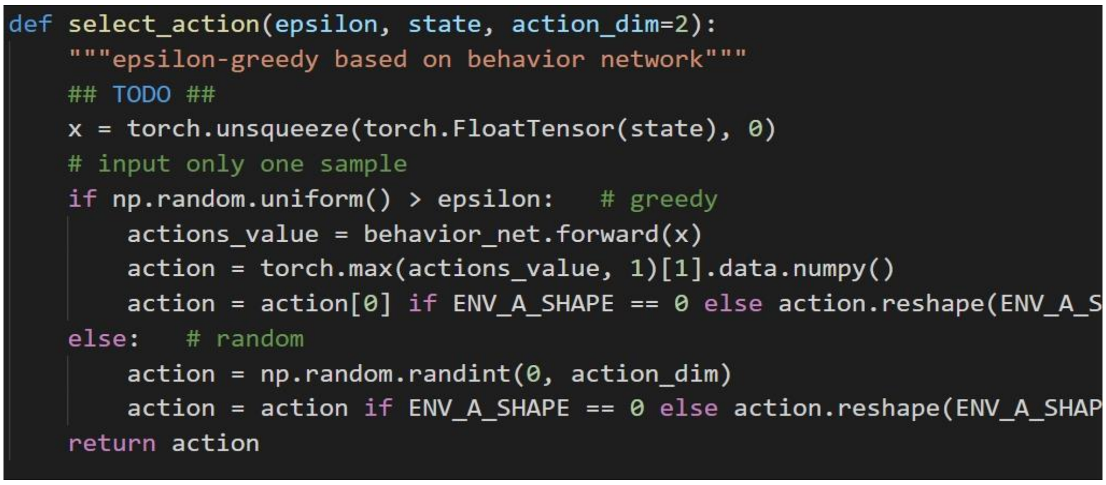
```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

A:



```
# select action
if total_steps < args.warmup:
    action = env.action_space.sample()
else:
    state_tensor = torch.Tensor(state).to(args.device)
    action = select_action(epsilon, state_tensor)
    epsilon = max(epsilon * args.eps_decay, args.eps_min)
```

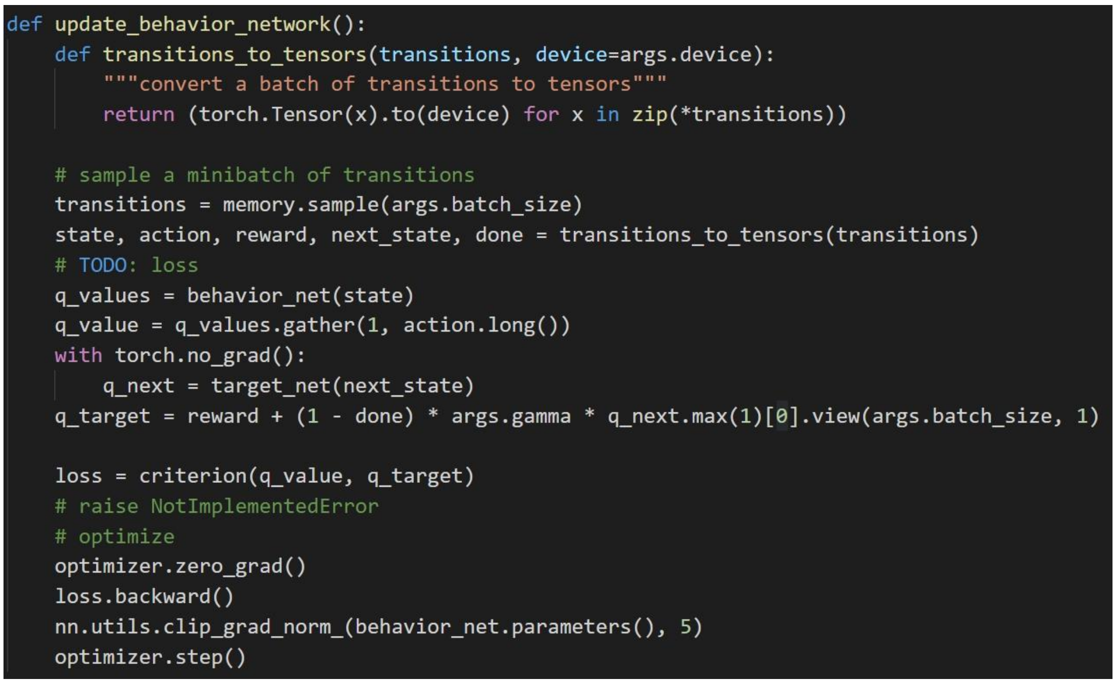
In the very beginning, the behavior_net did not have any experience, so it must take action in random when warmup. After that, action can be taken by actor_net in epsilon greedy and decay the epsilon.



```
def select_action(epsilon, state, action_dim=2):
    """epsilon-greedy based on behavior network"""
    ## TODO ##
    x = torch.unsqueeze(torch.FloatTensor(state), 0)
    # input only one sample
    if np.random.uniform() > epsilon:   # greedy
        actions_value = behavior_net.forward(x)
        action = torch.max(actions_value, 1)[1].data.numpy()
        action = action[0] if ENV_A_SHAPE == 0 else action.reshape(ENV_A_S
    else:   # random
        action = np.random.randint(0, action_dim)
        action = action if ENV_A_SHAPE == 0 else action.reshape(ENV_A_SHAP
    return action
```

Use epsilon greedy can let action_net make the right action, but sometimes try different action.

B:

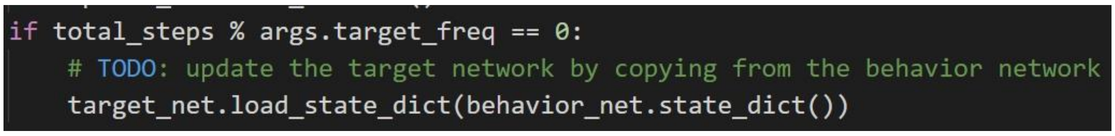


```
def update_behavior_network():
    def transitions_to_tensors(transitions, device=args.device):
        """convert a batch of transitions to tensors"""
        return (torch.Tensor(x).to(device) for x in zip(*transitions))

    # sample a minibatch of transitions
    transitions = memory.sample(args.batch_size)
    state, action, reward, next_state, done = transitions_to_tensors(transitions)
    # TODO: loss
    q_values = behavior_net(state)
    q_value = q_values.gather(1, action.long())
    with torch.no_grad():
        q_next = target_net(next_state)
    q_target = reward + (1 - done) * args.gamma * q_next.max(1)[0].view(args.batch_size, 1)

    loss = criterion(q_value, q_target)
    # raise NotImplementedError
    # optimize
    optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(behavior_net.parameters(), 5)
    optimizer.step()
```

When update behavior net, we need to sample some experience from the memory. To calculate the loss, we will put state in behavior net and put next_state in target net which is fixed. The reason why the target net should be fixed is because if both of the network are not fixed it will be very different to train , the target value is always unstable.
Loss function we will calculate is:

$$\text{MSELoss}(\ Q^{\pi}(S_t, a_t)\ ,\ r_t + Q^{\pi}(s_{t+1}, \pi(S_{t+1}))\ )$$

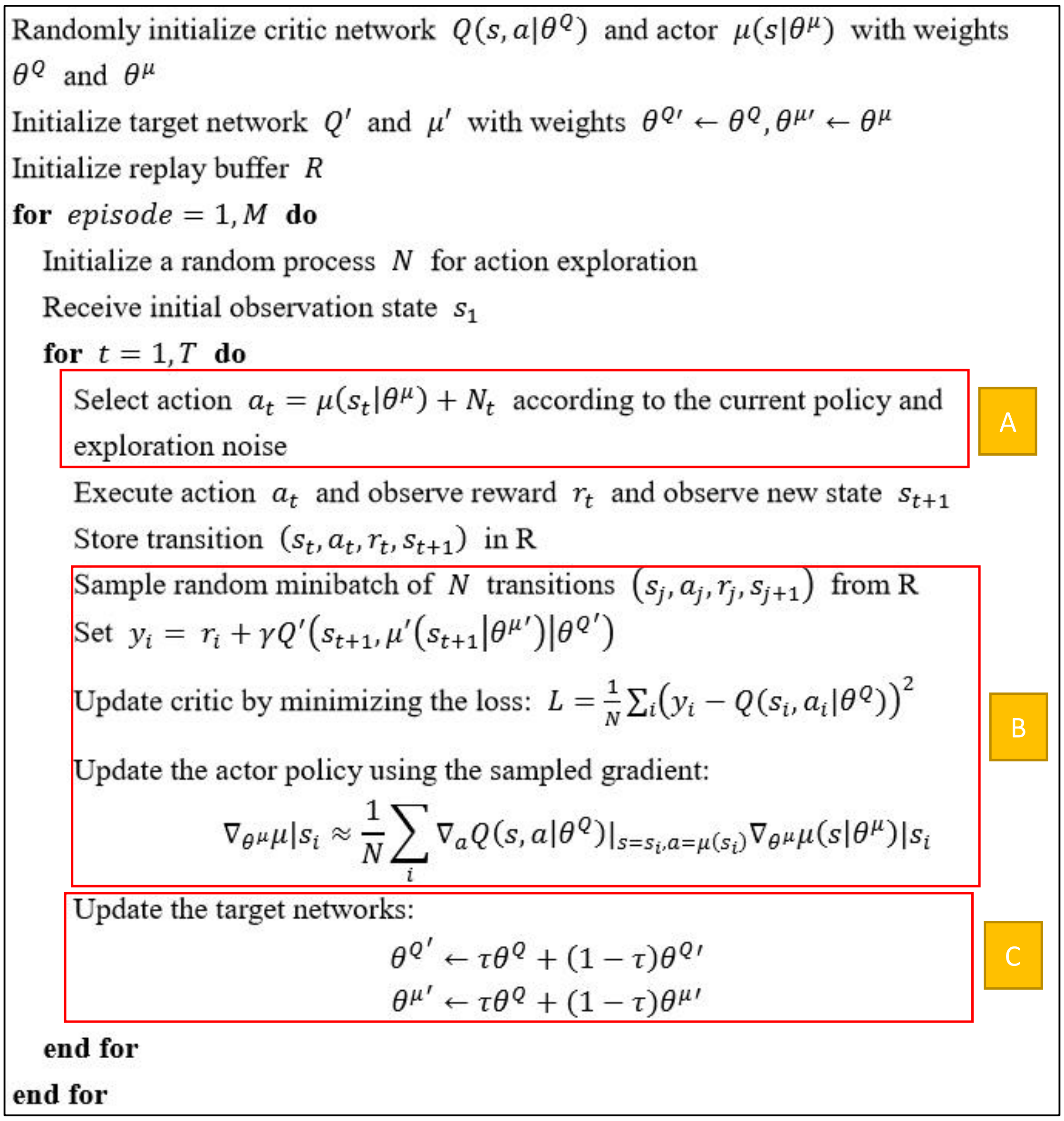
C:



```
if total_steps % args.target_freq == 0:
    # TODO: update the target network by copying from the behavior network
    target_net.load_state_dict(behavior_net.state_dict())
```

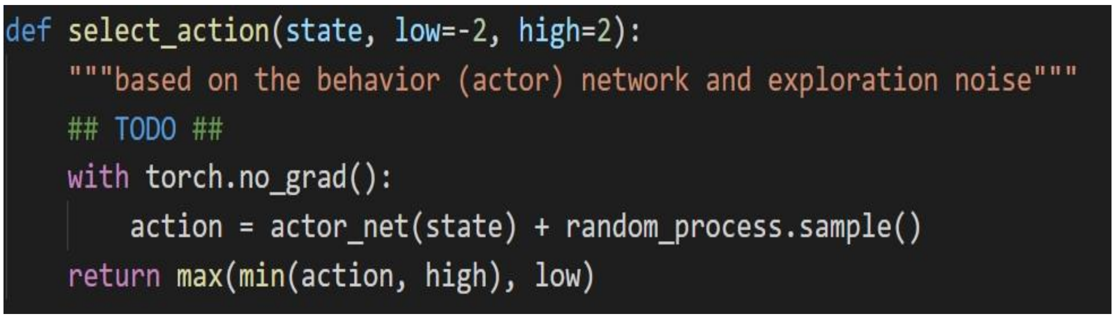
Every target_freq times update the target_net weight from behavior_net.

DDPG



```
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 $N$ for action exploration
    Receive initial observation state $s_1$
    for $t = 1, T$ do
        Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ 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 random minibatch of $N$ transitions $(s_j, a_j, r_j, s_{j+1})$ from R
        Set $y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$
        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 gradient:
            $\nabla_{\theta^\mu}\mu|s_i \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$
        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{\mu'}$
    end for
end for
```

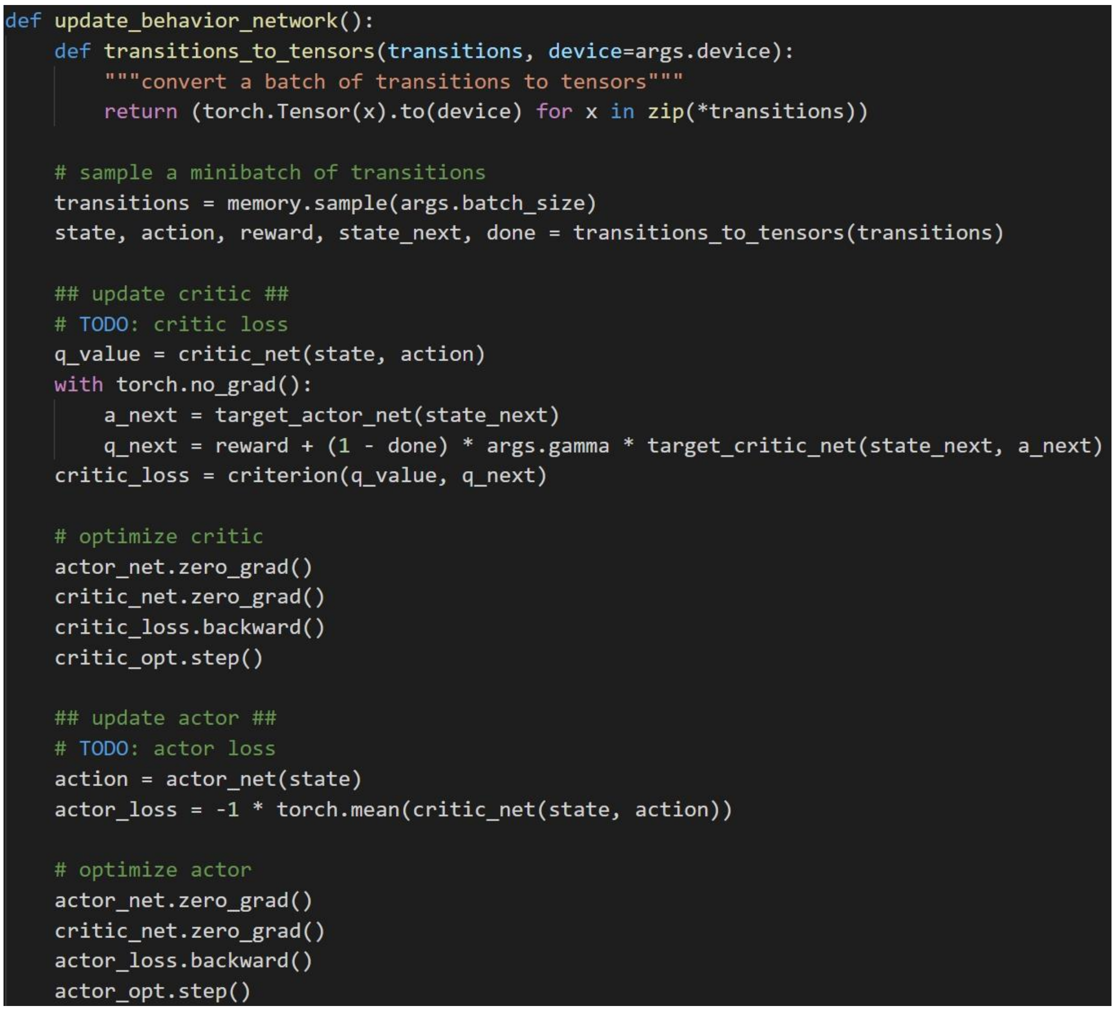
A:



```python
def select_action(state, low=-2, high=2):
    """based on the behavior (actor) network and exploration noise"""
    ## TODO ##
    with torch.no_grad():
        action = actor_net(state) + random_process.sample()
    return max(min(action, high), low)
```

When we select action we add some noise (random process) to let the behavior net can do a little change from the action. Sometimes can learn new action from the noise.

B:



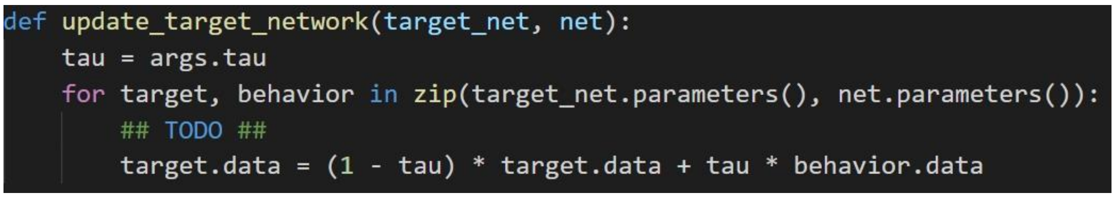
```python
def update_behavior_network():
    def transitions_to_tensors(transitions, device=args.device):
        """convert a batch of transitions to tensors"""
        return (torch.Tensor(x).to(device) for x in zip(*transitions))

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

    ## update critic ##
    # TODO: critic loss
    q_value = critic_net(state, action)
    with torch.no_grad():
        a_next = target_actor_net(state_next)
        q_next = reward + (1 - done) * args.gamma * target_critic_net(state_next, a_next)
    critic_loss = criterion(q_value, q_next)

    # optimize critic
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()

    ## update actor ##
    # TODO: actor loss
    action = actor_net(state)
    actor_loss = -1 * torch.mean(critic_net(state, action))

    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

Sample some experience from memory and try to update the critic network. It is similar like DQN, it need to estimate the value between

$$Q_{target} = r_i + \gamma Q'\left(s_{t+1}, \mu'\left(s_{t+1}\middle|\theta^{\mu'}\right)\middle|\theta^{Q'}\right) \;, Q(s_i, a_i|\theta^{Q})$$

MSELoss(Q_target , Q) can let critic know the reward in specific stae and action. Because of it, actor depends on critic to know which action to generate better reward.

C:



```python
def update_target_network(target_net, net):
    tau = args.tau
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        target.data = (1 - tau) * target.data + tau * behavior.data
```

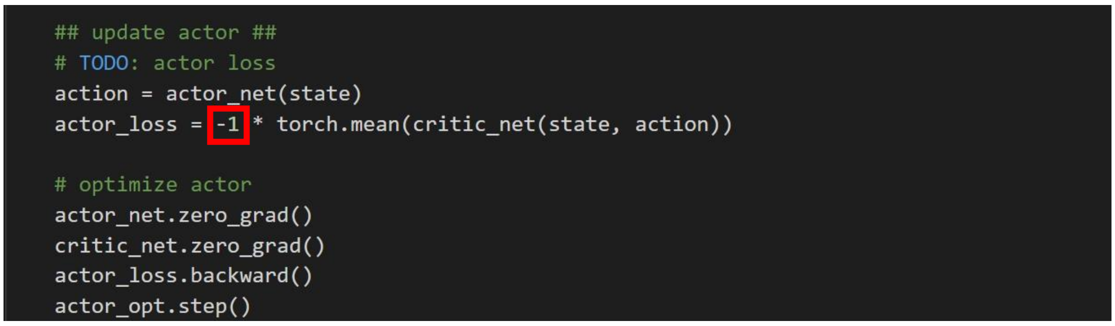
We make a copy of the target network parameters and have them slowly track those of the learned networks via “soft updates”.

4. Describe differences between your implementation and algorithms. (10%)

In DDPG, we need to update the actor policy using the sampled gradient

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

We need to maximize the actor weight, but in pytoch function is used to get minimize loss function, so we need to time -1.

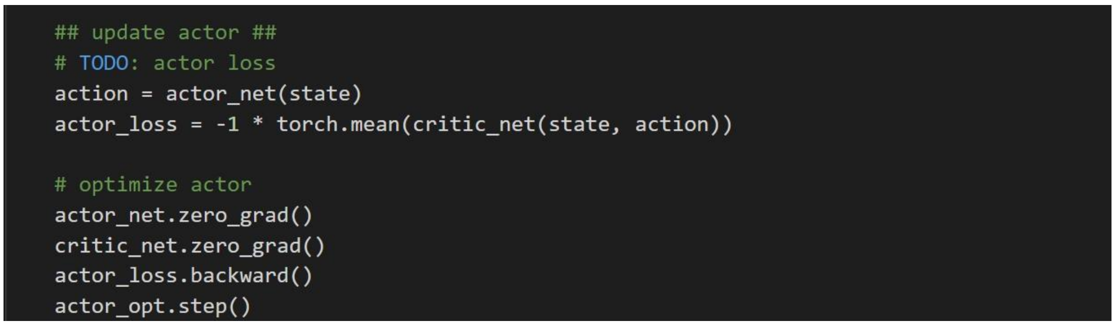


```
## update actor ##
# TODO: actor loss
action = actor_net(state)
actor_loss = -1 * torch.mean(critic_net(state, action))

# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

In both DQN and DDPG, we need to update our target network and get the better policy function. The different between implementation and algorithm is we need to fixed one of our target network in order to let the target value stable.

5. Describe your implementation and the gradient of actor updating. (10%)

Take DDPG for example:



```
## update actor ##
# TODO: actor loss
action = actor_net(state)
actor_loss = -1 * torch.mean(critic_net(state, action))

# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

Action = $\mu(s|\theta_\mu)$

Loss = $-Q(s, \text{action}|\theta_Q)$

The gradient of actor by chain rule :

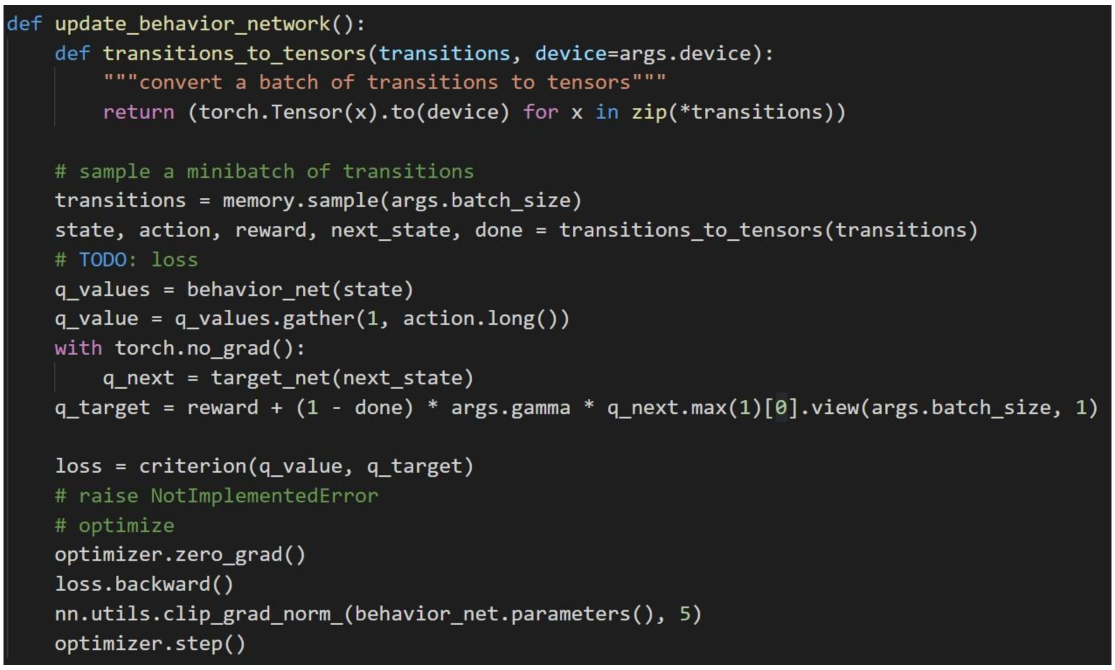
$$\begin{aligned}\frac{\nabla L}{\nabla \theta_u} &= -\frac{\nabla Q(s,a|\theta_Q)}{\nabla a}\frac{\nabla a}{\nabla u(s|\theta_u)}\frac{\nabla u(s|\theta_u)}{\nabla \theta_u} \\ &= -\frac{\nabla Q(s,a|\theta_Q)}{\nabla u(s|\theta_u)}\frac{\nabla u(s|\theta_u)}{\nabla \theta_u}\end{aligned}$$

Later, we need to update with tau:

$$\theta_{\hat{Q}} = (1-\tau)\theta_{\hat{Q}} + \tau\theta_Q \theta_{\hat{u}} = (1-\tau)\theta_{\hat{u}} + \tau\theta_u$$

6. Describe your implementation and the gradient of critic updating. (10%)

Take DQN for example:

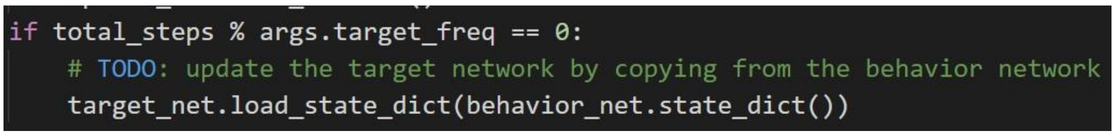


```python
def update_behavior_network():
    def transitions_to_tensors(transitions, device=args.device):
        """convert a batch of transitions to tensors"""
        return (torch.Tensor(x).to(device) for x in zip(*transitions))

    # sample a minibatch of transitions
    transitions = memory.sample(args.batch_size)
    state, action, reward, next_state, done = transitions_to_tensors(transitions)
    # TODO: loss
    q_values = behavior_net(state)
    q_value = q_values.gather(1, action.long())
    with torch.no_grad():
        q_next = target_net(next_state)
    q_target = reward + (1 - done) * args.gamma * q_next.max(1)[0].view(args.batch_size, 1)

    loss = criterion(q_value, q_target)
    # raise NotImplementedError
    # optimize
    optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(behavior_net.parameters(), 5)
    optimizer.step()
```

When update behavior net, we need to sample some experience from the memory. To calculate the loss, we will put state in behavior net and put next_state in target net which is fixed. The reason why the target net should be fixed is because if both of the network are not fixed it will be very different to train , the target value is always unstable.

Loss function we will calculate is:

$$\text{MSELoss}(\ Q^{\pi}(S_t, a_t)\ ,\ r_t + Q^{\pi}(s_{t+1}, \pi(S_{t+1}))\ )$$



```python
if total_steps % args.target_freq == 0:
    # TODO: update the target network by copying from the behavior network
    target_net.load_state_dict(behavior_net.state_dict())
```

Every target_freq times update the target_net weight from behavior_net.

7. Explain effects of the discount factor. (5%)

If discount factor is smaller, it means we do not care about the future reward and we care about the current reward.

If discount factor is higher, it means the future reward will effect more in reward and leads to far-sighted evalution.

8. **Explain benefits of epsilon-greedy in comparison to greedy action selection. (5%)**

    The benefits of epsilon-greedy is that it can explore new action it have never seen and try some differnet. But if we use greedy action we will not robust when the environment change or may lead to a short-sighted.

9. **Explain the necessity of the target network. (5%)**

Help us to calculate and update the network easily, if we didn't have target network, the target value will be unstable to calculate , and hard to train the model.

10. **Explain the effect of replay buffer size in case of too large or too small. (5%)**

    If the replay buffer size too large, the calculate time will be huge but you can get the better reward quickly.

    If the replay buffer size too small, you will get better reward after very long episode.

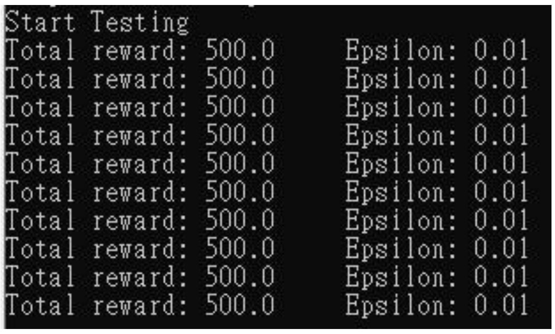
Report Bonus (10%)

1. **Explain the choice of the random process rather than normal distribution. (5%)**

Because normal distribution can only get the action from the same distribution, but random process can get different action from different distribution.

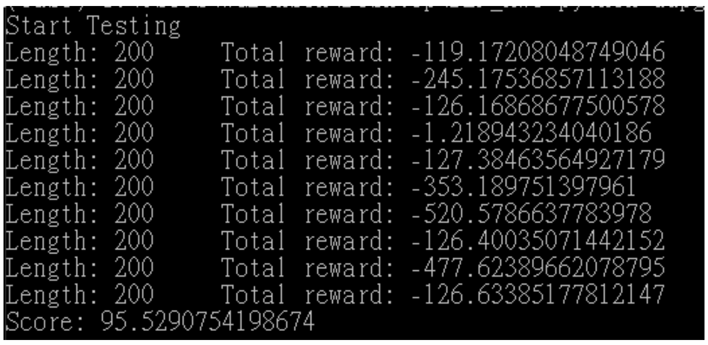
Performance (20%)

1. **[CartPole-v1] Average reward of 10 testing episodes: Average ÷ 5**



```
Start Testing
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
Total reward: 500.0     Epsilon: 0.01
```

2. **[Pendulum-v0] Average reward of 10 testing episodes: (Average + 700) ÷ 5**



```
Start Testing
Length: 200     Total reward: -119.17208048749046
Length: 200     Total reward: -245.17536857113188
Length: 200     Total reward: -126.16868677500578
Length: 200     Total reward: -1.218943234040186
Length: 200     Total reward: -127.38463564927179
Length: 200     Total reward: -353.189751397961
Length: 200     Total reward: -520.5786637783978
Length: 200     Total reward: -126.40035071442152
Length: 200     Total reward: -477.62389662078795
Length: 200     Total reward: -126.63385177812147
Score: 95.5290754198674
```