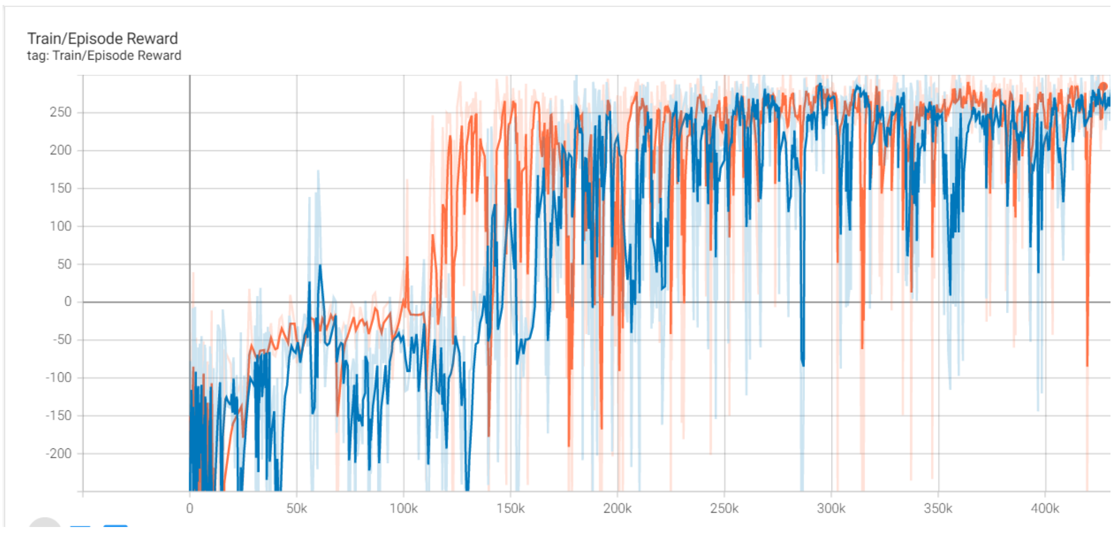
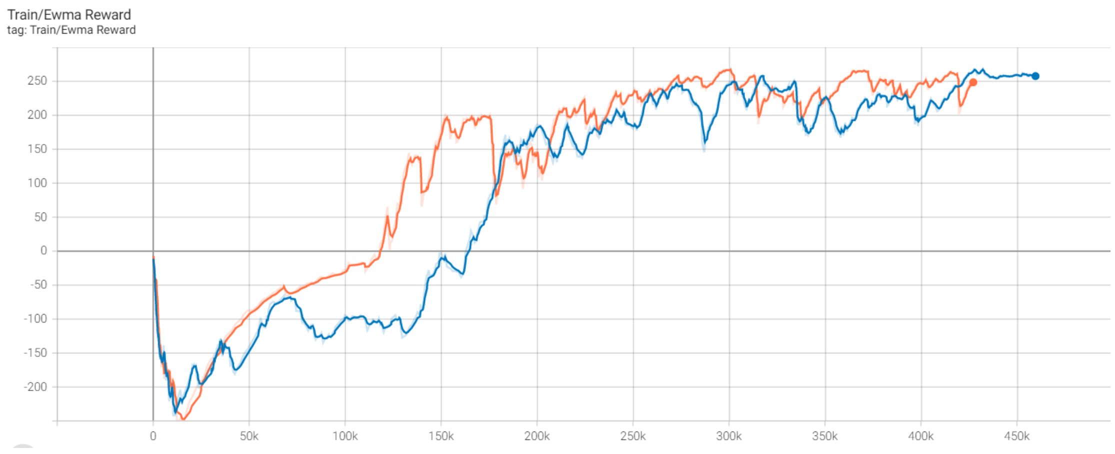
# Report

■ A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2 (5%) (Orange)

■ A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2 (5%) (Blue)

Both of them using more hidden layer than origin.

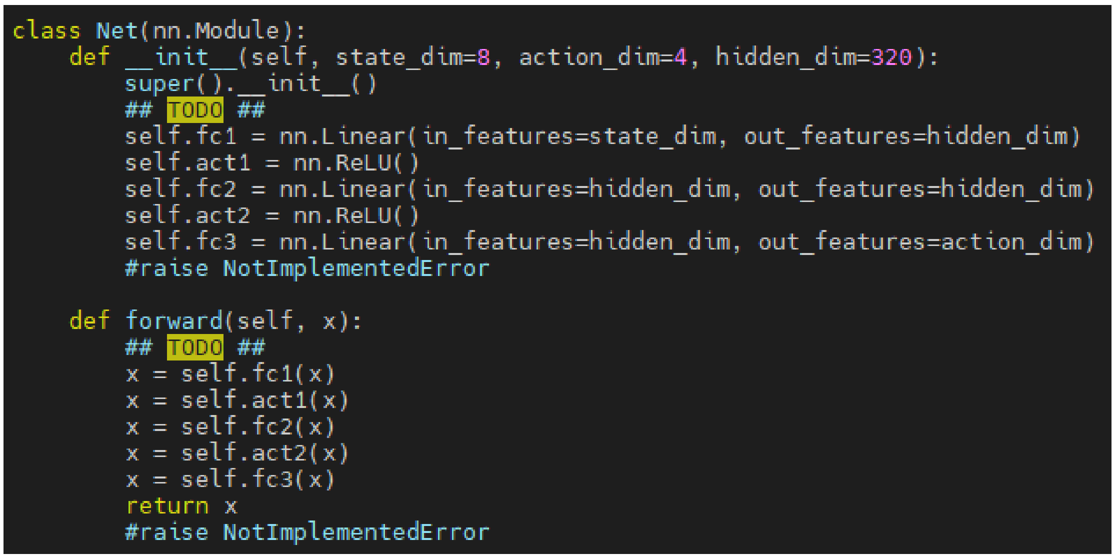




■ Describe your major implementation of both algorithms in detail. (20%)

Create the net by slide

DQN:



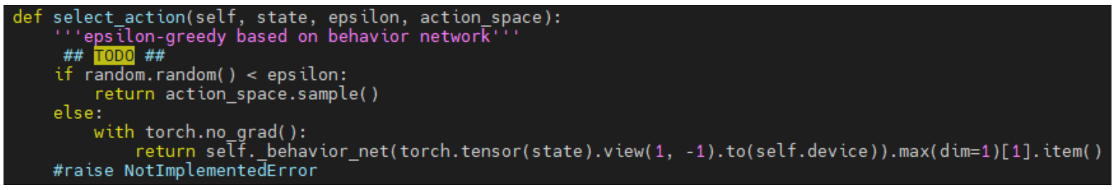
```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=320):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(in_features=state_dim, out_features=hidden_dim)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(in_features=hidden_dim, out_features=hidden_dim)
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(in_features=hidden_dim, out_features=action_dim)
        #raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        x = self.fc1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.act2(x)
        x = self.fc3(x)
        return x
        #raise NotImplementedError
```

The epsilon-greedy part. With the probability we random take action, also input the state and count the q-value and action, then take the max one (key is action).

With probability $\varepsilon$ select a random action $a_t$

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$



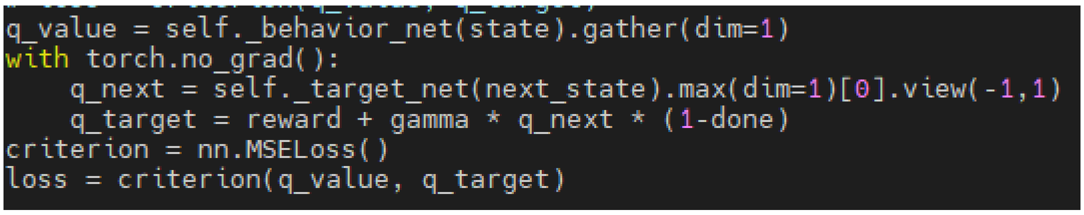
```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        with torch.no_grad():
            return self._behavior_net(torch.tensor(state).view(1, -1).to(self.device)).max(dim=1)[1].item()
    #raise NotImplementedError
```

The q value in target net and behavior net to complete the loss. The target net is stable to compare.

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters $\theta$

Every $C$ steps reset $\hat{Q} = Q$

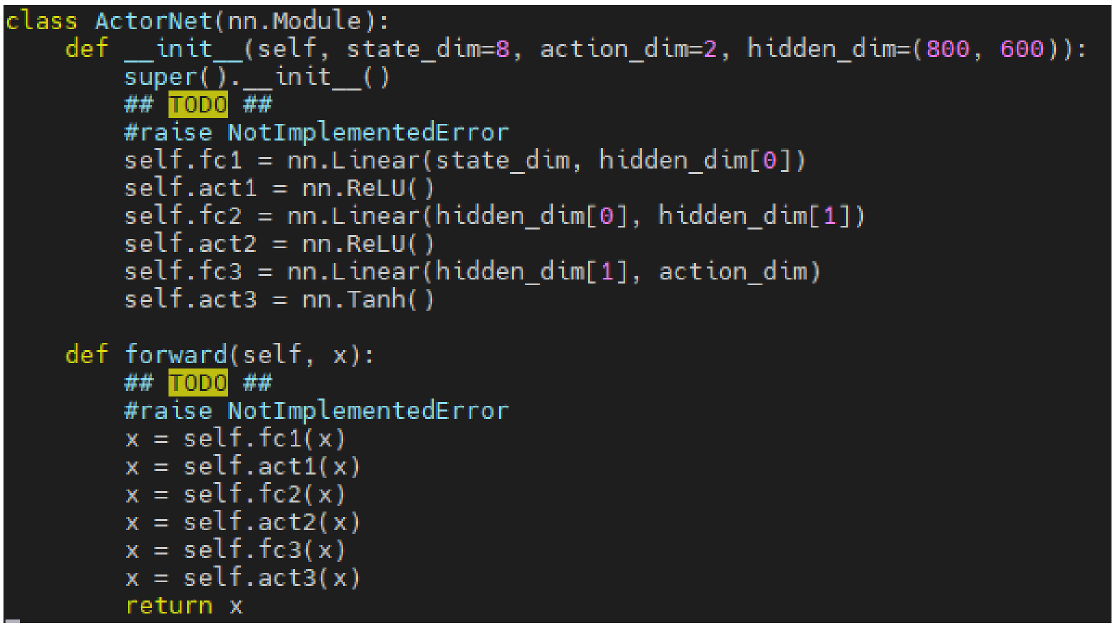


```python
q_value = self._behavior_net(state).gather(dim=1)
with torch.no_grad():
    q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1)
    q_target = reward + gamma * q_next * (1-done)
criterion = nn.MSELoss()
loss = criterion(q_value, q_target)
```

Worth to mention, the “state” variant in two upper function have different data type (narray and tensor)

DDQG:

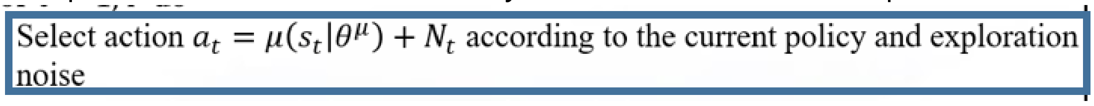
The net is also base on slide.



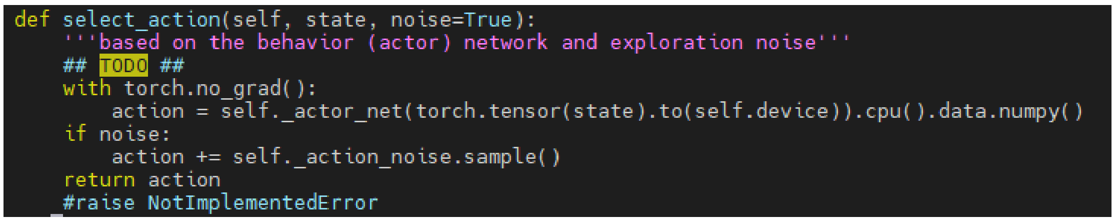
```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(800, 600)):
        super().__init__()
        ## TODO ##
        #raise NotImplementedError
        self.fc1 = nn.Linear(state_dim, hidden_dim[0])
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_dim[1], action_dim)
        self.act3 = nn.Tanh()

    def forward(self, x):
        ## TODO ##
        #raise NotImplementedError
        x = self.fc1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.act2(x)
        x = self.fc3(x)
        x = self.act3(x)
        return x
```

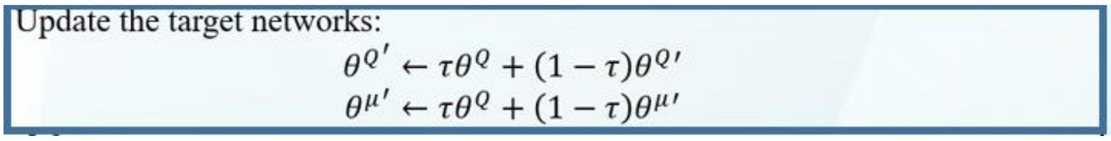
This part also same to slide. The only different is that noise is optional.



Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise

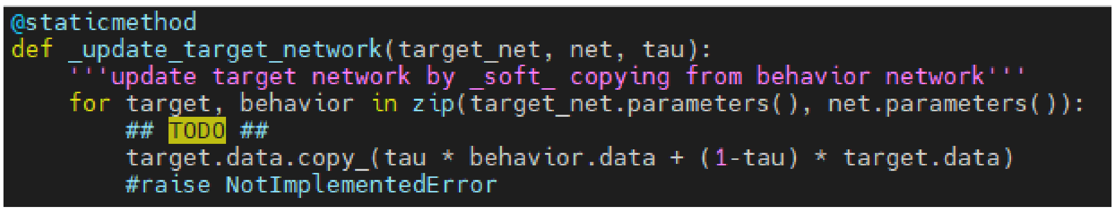


```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        action = self._actor_net(torch.tensor(state).to(self.device)).cpu().data.numpy()
    if noise:
        action += self._action_noise.sample()
    return action
    #raise NotImplementedError
```



Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{\mu'}$$

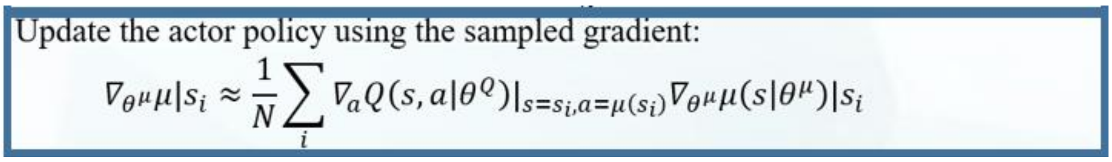


```python
@staticmethod
def _update_target_network(target_net, net, tau):
    '''update target network by _soft_ copying from behavior network'''
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        target.data.copy_(tau * behavior.data + (1-tau) * target.data)
        #raise NotImplementedError
```

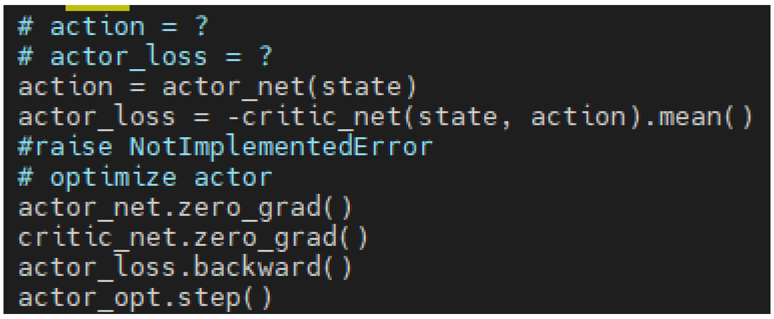
■ Describe differences between your implementation and algorithms. (10%)

The implementation base on the algorithms in outline.

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


Update the actor policy using the sampled gradient:

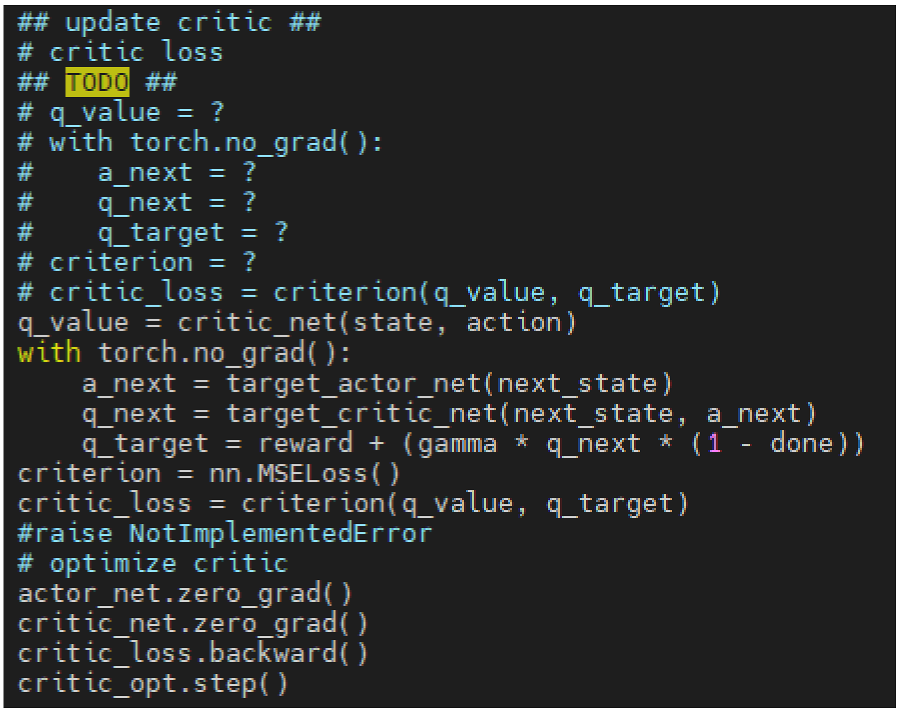
$$\nabla_{\theta^\mu}\mu|s_i \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)}\nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$$


```
# action = ?
# actor_loss = ?
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()
#raise NotImplementedError
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

(The update part is given, not our implementation.)

The action is from the actor net. And the q-value is from the critic net (with the input action and state) then use mean to get the scalar for the gradient ascent.

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


```
## update critic ##
# critic loss
## TODO ##
# q_value = ?
# with torch.no_grad():
#     a_next = ?
#     q_next = ?
#     q_target = ?
# criterion = ?
# critic_loss = criterion(q_value, q_target)
q_value = critic_net(state, action)
with torch.no_grad():
    a_next = target_actor_net(next_state)
    q_next = target_critic_net(next_state, a_next)
    q_target = reward + (gamma * q_next * (1 - done))
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
#raise NotImplementedError
# optimize critic
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

(The update part is given, not our implementation.)

Similar to the actor and dqn's behavior net. Take the value from target net to get more stable value and compute the loss to do backpropagate.

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

The reward from the farer future should be less Influential. So, we multiple the factor to make the farer future reward to make the reward be smaller.

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

An explore-exploit tradeoff, to make us choose a way seen not such good. Avoid to be too greedy and converge to a local optimal position.

■ Explain the necessity of the target network. (5%)

In Q-Learning, we update a guess with a guess. So, we using a non-trained target network to preserve the parameters of behavior network to avoid the network update be too unstable.

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

Replay buffer size used to sample the correlated elements. When the size too large, the learning speed would be slow and need a lot of memory. When the size too small, it would learn the recent data and could be overfitting.

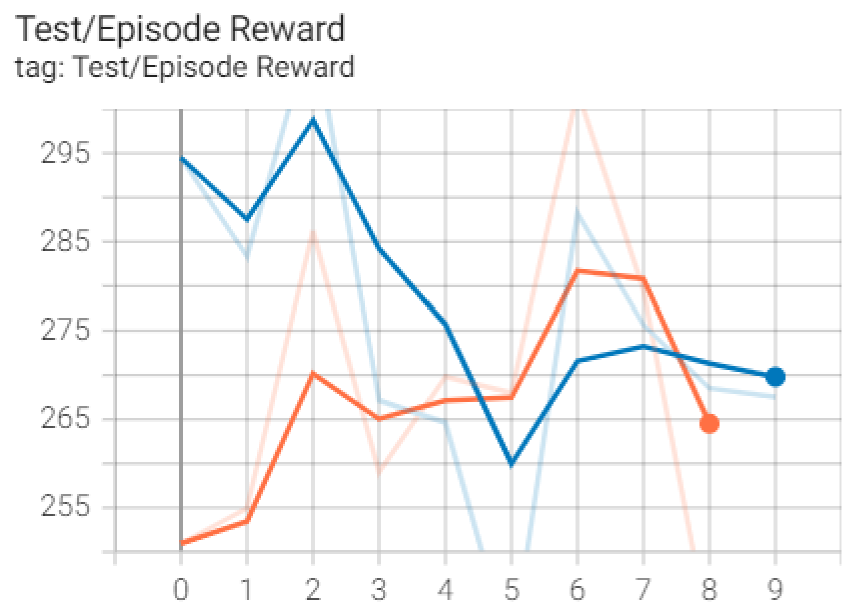
● Performance (20%)

■ [LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30

Average Reward 236.66

■ [LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30

Average Reward 275.72


Test/Episode Reward
tag: Test/Episode Reward