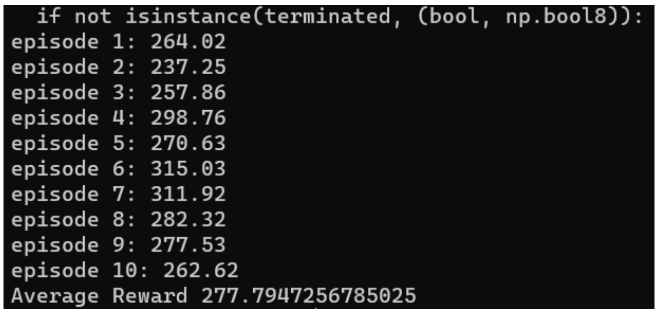
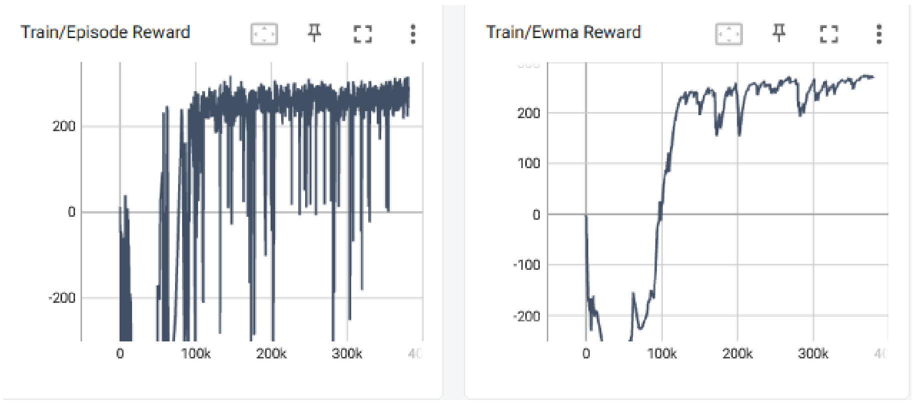
312554012 王偉誠 Lab5: DQN and DDPG

◆ Experimental Results

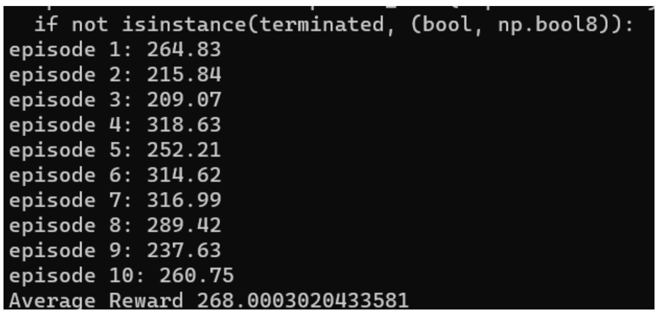
Your screenshot of tensorboard and testing results on LunarLander-v2 using DQN.



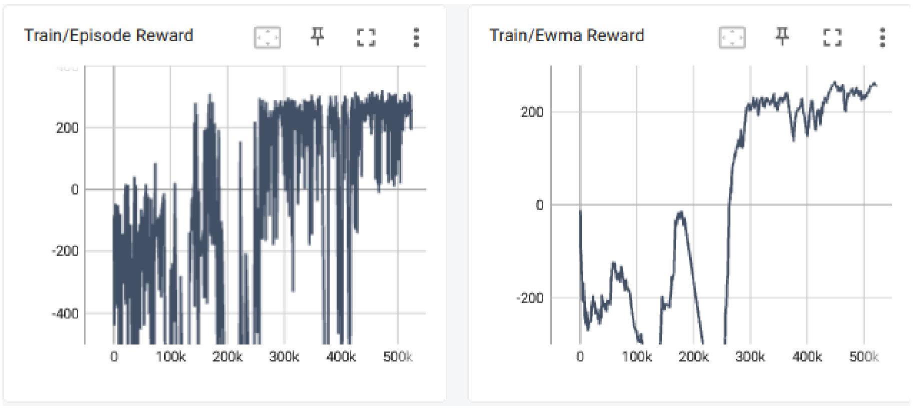
```
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 264.02
episode 2: 237.25
episode 3: 257.86
episode 4: 298.76
episode 5: 270.63
episode 6: 315.03
episode 7: 311.92
episode 8: 282.32
episode 9: 277.53
episode 10: 262.62
Average Reward 277.7947256785025
```



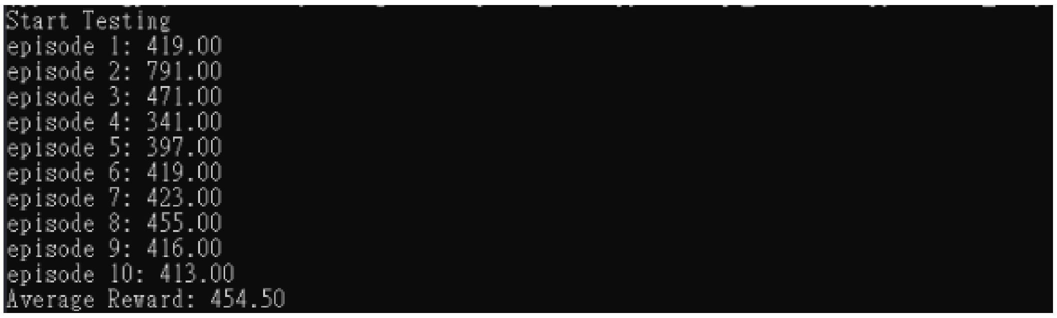
Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2 using DDPG.



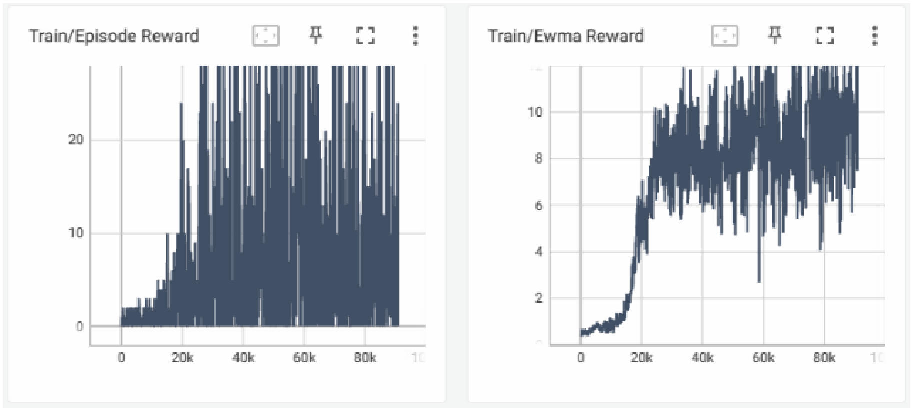
```
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 264.83
episode 2: 215.84
episode 3: 209.07
episode 4: 318.63
episode 5: 252.21
episode 6: 314.62
episode 7: 316.99
episode 8: 289.42
episode 9: 237.63
episode 10: 260.75
Average Reward 268.0003020433581
```



Your screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4.

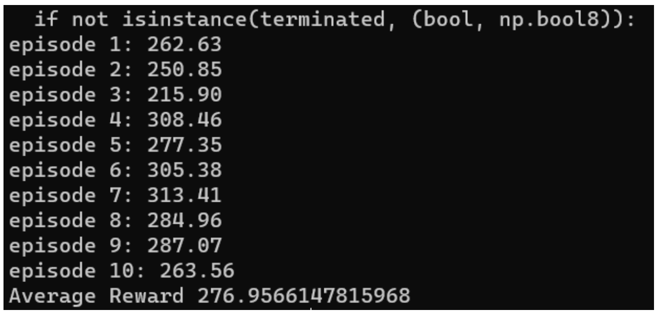


```
Start Testing
episode 1: 419.00
episode 2: 791.00
episode 3: 471.00
episode 4: 341.00
episode 5: 397.00
episode 6: 419.00
episode 7: 423.00
episode 8: 455.00
episode 9: 416.00
episode 10: 413.00
Average Reward: 454.50
```

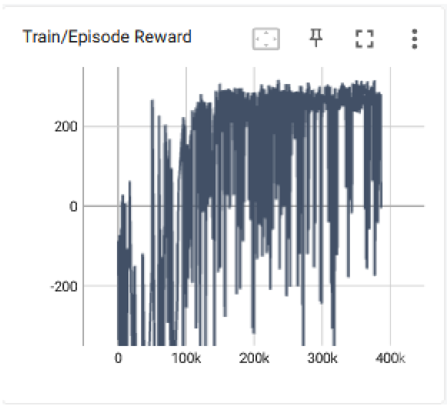


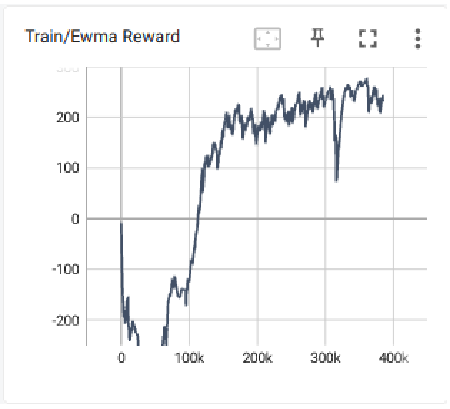
(Bonus)

Your screenshot of tensorboard and testing results on LunarLander-v2 using DDQN.

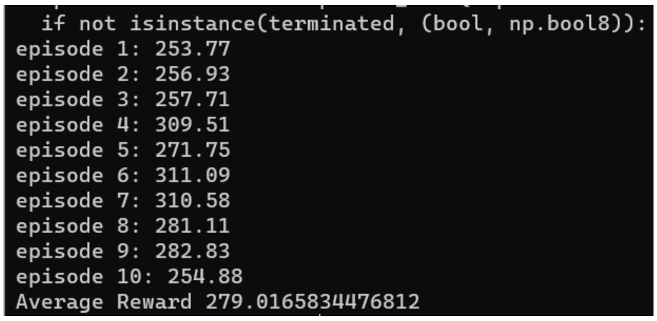


```
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 262.63
episode 2: 250.85
episode 3: 215.90
episode 4: 308.46
episode 5: 277.35
episode 6: 305.38
episode 7: 313.41
episode 8: 284.96
episode 9: 287.07
episode 10: 263.56
Average Reward 276.9566147815968
```

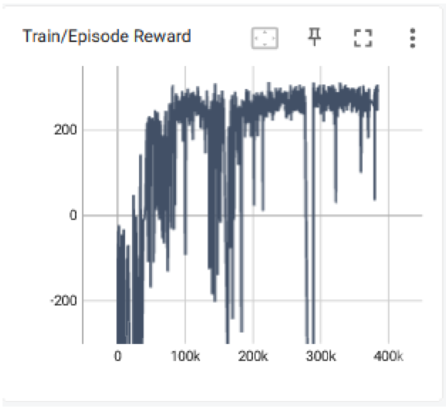


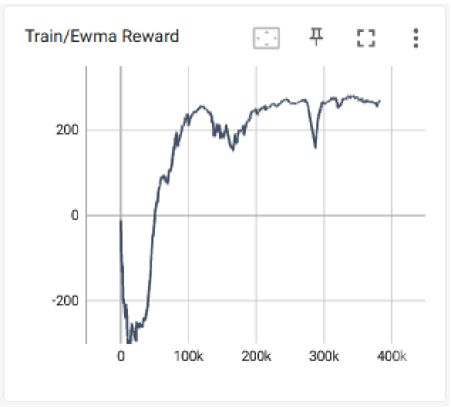


Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2 using TD3.



```
  if not isinstance(terminated, (bool, np.bool8)):
episode 1: 253.77
episode 2: 256.93
episode 3: 257.71
episode 4: 309.51
episode 5: 271.75
episode 6: 311.09
episode 7: 310.58
episode 8: 281.11
episode 9: 282.83
episode 10: 254.88
Average Reward 279.0165834476812
```

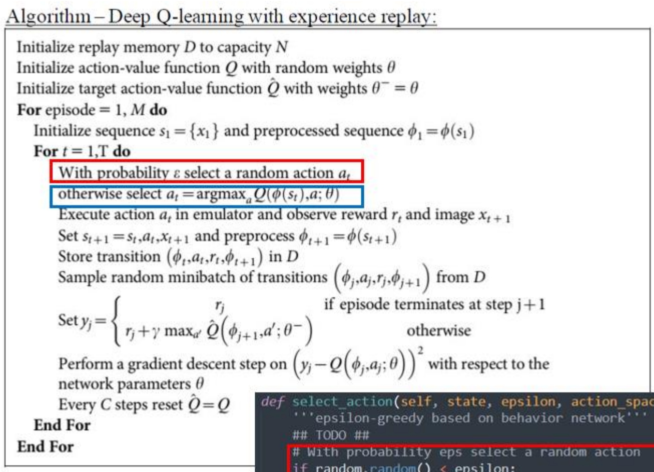




◆ Questions

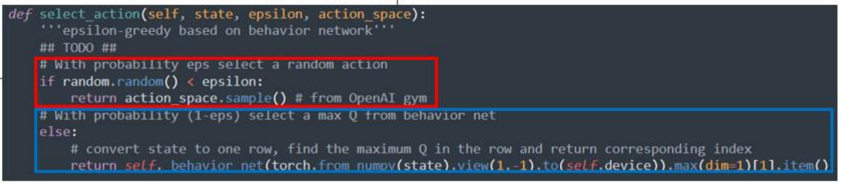
1. Describe your major implementation of both DQN and DDPG in detail.

DQN:



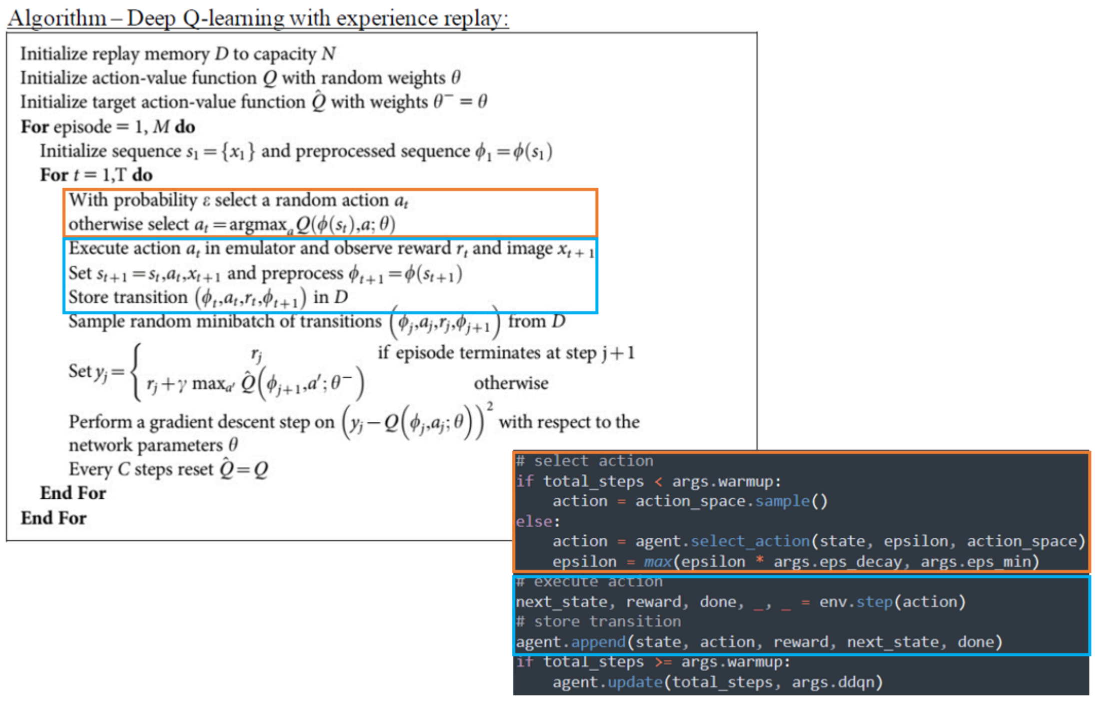
Algorithm – Deep Q-learning with experience replay:

```
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
```



```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    # With probability eps select a random action
    if random.random() < epsilon:
        return action_space.sample() # from OpenAI gym
    # With probability (1-eps) select a max Q from behavior net
    else:
        # convert state to one row, find the maximum Q in the row and return corresponding index
        return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)[1].item()
```

The boxed portion in the diagram represents the epsilon-greedy strategy. With a probability of epsilon, a random action is chosen (depicted by the red box). The action_space.sample() function provided by the OpenAI Gym environment can be used to randomly sample an action from the environment's action space. Conversely, to determine the action with the highest Q-value from the behavior network, one needs to identify the action with the maximum Q-value (indicated by the blue box).

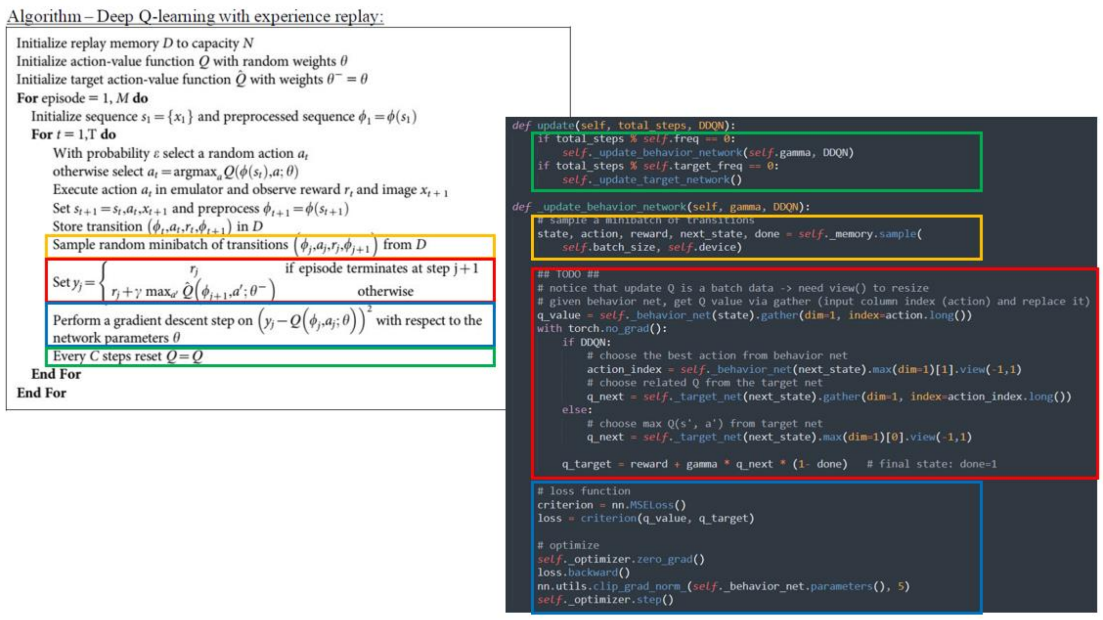


Algorithm – Deep Q-learning with experience replay:

```
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
```

```python
# select action
if total_steps < args.warmup:
    action = action_space.sample()
else:
    action = agent.select_action(state, epsilon, action_space)
    epsilon = max(epsilon * args.eps_decay, args.eps_min)
# execute action
next_state, reward, done, _, _ = env.step(action)
# store transition
agent.append(state, action, reward, next_state, done)
if total_steps >= args.warmup:
    agent.update(total_steps, args.ddqn)
```

Following that, the light blue box in the diagram depicts the execution of an action resulting in the next state. The entire transition is then stored in the replay buffer. Utilizing the step function from the Gym library provides the reward obtained by performing the action in the current state, as well as the next state. The boolean value "done" informs us whether this state is a terminal state or not.



Algorithm – Deep Q-learning with experience replay:

```
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
```

```python
def update(self, total_steps, DDQN):
    if total_steps % self.freq == 0:
        self._update_behavior_network(self.gamma, DDQN)
    if total_steps % self.target_freq == 0:
        self._update_target_network()

def _update_behavior_network(self, gamma, DDQN):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## TODO ##
    # notice that update Q is a batch data -> need view() to resize
    # given behavior net, get Q value via gather (input column index (action) and replace it)
    q_value = self._behavior_net(state).gather(dim=1, index=action.long())
    with torch.no_grad():
        if DDQN:
            # choose the best action from behavior net
            action_index = self._behavior_net(next_state).max(dim=1)[1].view(-1,1)
            # choose related Q from the target net
            q_next = self._target_net(next_state).gather(dim=1, index=action_index.long())
        else:
            # choose max Q(s', a') from target net
            q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1)

        q_target = reward + gamma * q_next * (1- done)   # final state: done=1

    # loss function
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)

    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

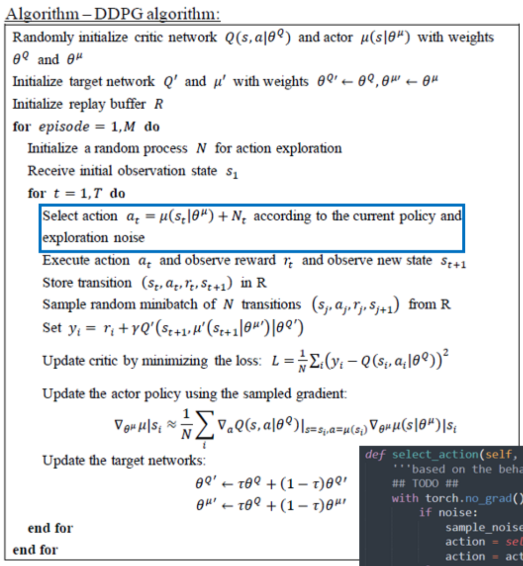
Next, a minibatch of transitions is sampled from the replay buffer (depicted by the orange box). Subsequently, the Q-value and Q-target are determined (illustrated by the red box), with the goal of minimizing the discrepancy between them (highlighted by the blue box). The Q-value is directly retrieved from the behavior network. In terms of implementation, the PyTorch gather function is often employed to substitute the index, transforming the action into an index for input into the behavior network, resulting in the Q-value as the output.

The concept of the Q-target bears similarities to the TD target, particularly in the context of the DQN. By feeding the next state into the target network to acquire the Q-value for the next state (q next), the Q-target is calculated as the reward added to the product of the discount factor gamma and q next, multiplied by (1 - done). The inclusion of an additional multiplication by (1 - done) serves to account for the fact that if the next state is a terminal state, done equals 1, leading the Q-target to be directly equivalent to the reward. Conversely, when done equals 0, multiplying by 1 has no effect on the result.

Subsequently, the backpropagation process follows (depicted by the blue box). As the objective is to minimize the disparity between the Q-value and Q-target, employing the mean squared error of these two values as the loss facilitates the backpropagation procedure.

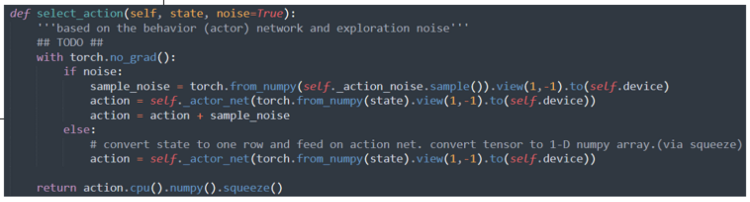
Finally, the update frequency is addressed, as indicated by the green box.

DDPG:



Algorithm – DDPG algorithm:

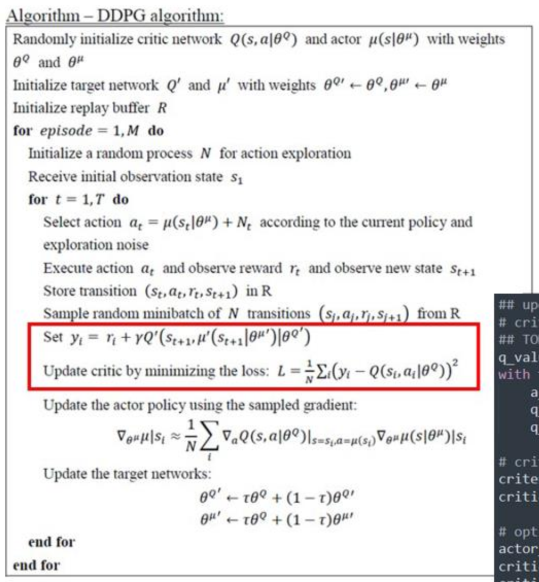
```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for $episode = 1,M$ do
  Initialize a random process $N$ 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
```



```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            sample_noise = torch.from_numpy(self._action_noise.sample()).view(1,-1).to(self.device)
            action = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
            action = action + sample_noise
        else:
            # convert state to one row and feed on action net. convert tensor to 1-D numpy array.(via squeeze)
            action = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))

    return action.cpu().numpy().squeeze()
```

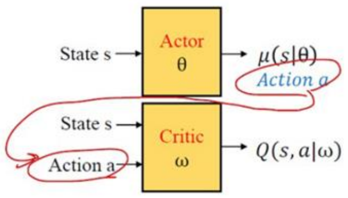
Firstly, let's discuss the action selection process. In the context of DQN, the epsilon-greedy strategy involves adding a Gaussian noise term directly to the action in DDPG, thereby introducing perturbation to achieve exploration. In terms of implementation, because noise is not required during testing, a distinction is made between the scenarios where noise is set to true and false.

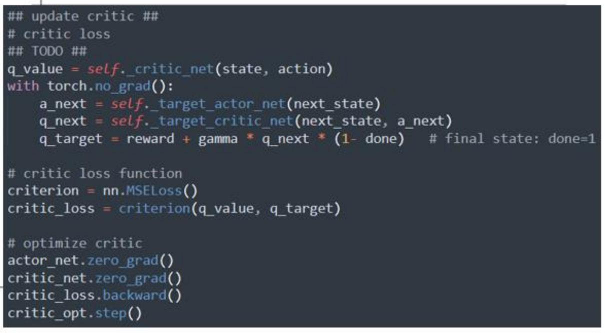
When noise is set to true, the action is adjusted as follows: action = action + sampled_noise. On the other hand, when noise is set to false, the action is returned as is without any additional modification.



Algorithm – 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 $N$ 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 $(s_i, a_i, r_i, s_{i+1})$ 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
```



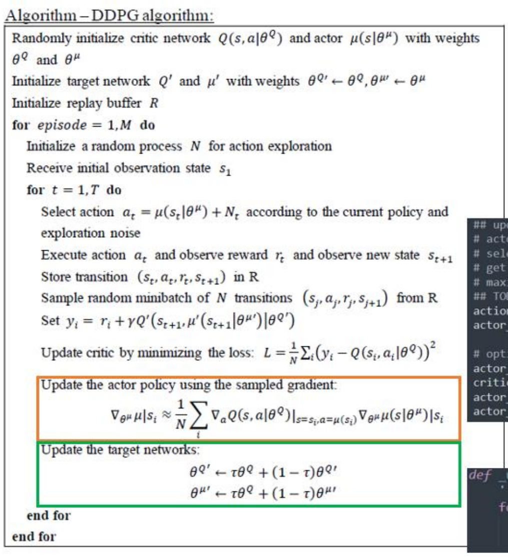


```python
## update critic ##
# critic loss
## TODO ##
q_value = self._critic_net(state, action)
with torch.no_grad():
    a_next = self._target_actor_net(next_state)
    q_next = self._target_critic_net(next_state, a_next)
    q_target = reward + gamma * q_next * (1- done)   # final state: done=1

# critic loss function
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)

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

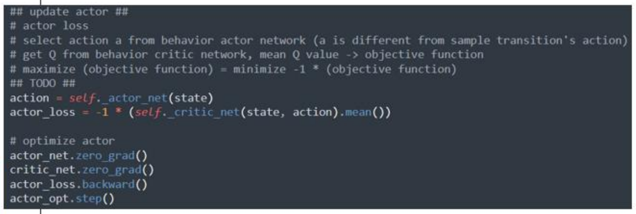
Moving on, let's discuss the process of updating the critic network. Similar to before, this involves both the Q-value and the Q-target. For the Q-value, you can directly input the current state and action into the critic network to obtain the Q-value.

For the Q-target, the following steps are involved: First, pass the next state through the target_actor network to get the next action (a_next). Then, combine a_next and the next state, and input them into the target_critic network to obtain the Q-value for the next state (q_next). This process is illustrated in the upper right corner of the diagram you mentioned.

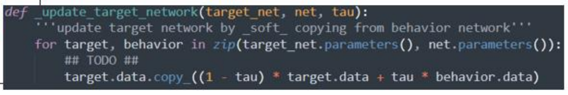
Subsequently, calculate the Q-target using the formula:
q_target = reward + gamma * q_next * (1 - done), and the procedure of using Mean Squared Error (MSE) loss is analogous to what you have already described for DQN. Given that, there's no need to elaborate further on this aspect.



```
Algorithm – 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 $N$ 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
```



```python
## update actor ##
# actor loss
# select action a from behavior actor network (a is different from sample transition's action)
# get Q from behavior critic network, mean Q value -> objective function
# maximize (objective function) = minimize -1 * (objective function)
## TODO ##
action = self._actor_net(state)
actor_loss = -1 * (self._critic_net(state, action).mean())

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



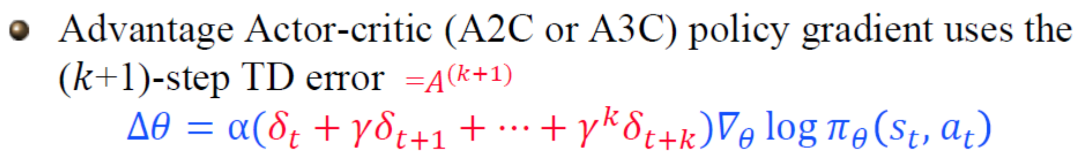
```python
def _update_target_network(target_net, net, tau):
    '''update target network by _soft_ copying from behavior network'''
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        target.data.copy_((1 - tau) * target.data + tau * behavior.data)
```

After updating the critic, the next step is to update the actor. The policy gradient approach aims to maximize the objective function, as indicated by the orange boxed area in the diagram. In this context, the current state is passed to the actor network to obtain an action (distinct from the initial action retrieved from transitions). Subsequently, this action and the state are fed into the updated critic, and the goal is to increase the Q-value provided by the critic. In practice, the obtained Q-values are averaged and negated to serve as the loss function. Backpropagation using this loss facilitates the ascent of the gradient.

Finally, we address the process of soft target updates depicted in the green boxed area. Here, the parameter tau is set to 0.005. This implies that the target value is updated as follows: target = 0.995 * target + 0.005 * new. This can be understood as making the target value change by a small fraction each time (0.5%). This approach helps maintain a degree of stability in the target value, preventing abrupt and large fluctuations that could be caused by significant changes in a single update.

2. Explain effects of the discount factor.

The discount factor refers to the principle that TD errors originating farther in the future have a decreasing impact on the present. In the context of this lab, only one-step TD errors are utilized, so the effect of the discount factor might not be as prominent. However, if considering k-step TD errors, the situation could be as depicted in the diagram:



- Advantage Actor-critic (A2C or A3C) policy gradient uses the $(k+1)$-step TD error $= A^{(k+1)}$

$$\Delta\theta = \alpha(\delta_t + \gamma\delta_{t+1} + \cdots + \gamma^k\delta_{t+k})\nabla_\theta \log \pi_\theta(s_t, a_t)$$

The variable "r" represents the discount factor, which is a value that is typically less than 1. As a result, as the exponent of "r" increases, its value becomes smaller. When this smaller value is multiplied by future TD errors, the impact of those errors on the current state diminishes.

3. Explain benefits of epsilon-greedy in comparison to greedy action selection.

In reinforcement learning, striking a balance between exploration and exploitation is crucial. The epsilon-greedy strategy is one such approach. If we consistently choose the best action with the highest Q-value (a greedy approach), we'll never learn whether there are potentially better actions that we haven't discovered or tried yet. Hence, it's essential to allocate a certain proportion of selections to the best-known actions (exploitation) while also dedicating a portion to randomly selecting actions other than the best-known ones (exploration). This way, we can systematically explore the environment and optimize our decision-making process.

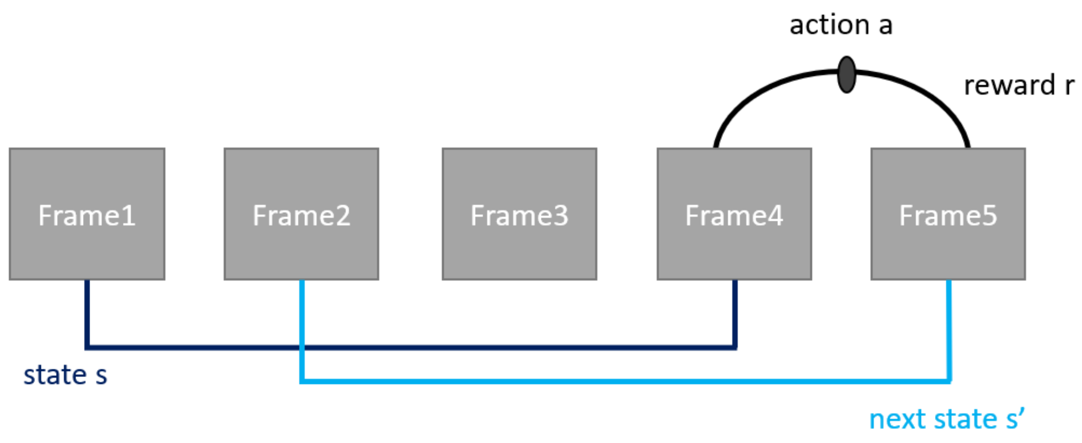
4. Explain the necessity of the target network.

Utilizing a target network serves the purpose of preventing frequent updates to the behavior network, which could lead to volatile value estimations. By extracting values from the target network, the obtained values are more stable and less prone to fluctuations. This stability contributes to more reliable and smoother learning in the reinforcement learning process.

5. Describe the tricks you used in Breakout and their effects, and how they differ from those used in LunarLander.

The most significant difference between implementing Breakout and LunarLander lies in the fact that Breakout takes in the entire image as input. To account for this, a key adjustment is to stack a sequence of four frames together. This stacking enables the observation of the ball's trajectory for training purposes. Consequently, additional modifications are needed in the Replay Memory to accommodate this.

In the Replay Memory, we need to store information like state, action, reward, next_state, and done. A single state consists of four stacked frames. To handle this, a deque(maxlen=5) is used to temporarily store frames. Subsequently, the frames from the deque, along with action, reward, and done, are placed into the Replay Memory. The Replay Memory then divides these five frames into two parts: the first four and the last four frames, which are treated as the state and next_state respectively. This process is illustrated in the diagram:



Additionally, at the beginning of the game when the deque doesn't contain any frames, a few no-op actions are taken to gather frames and fill the deque. During action selection, the last four frames in the deque constitute the current state. The action is chosen based on this state. The rest of the implementation process aligns closely with the LunarLander approach and doesn't require further elaboration.