Deep Learning Lab6 –

Deep Q-Network and Deep Deterministic Policy Gradient Report

智能所 311581006 林子淩

1. Experimental Results

screenshot of tensorboard and testing results on LunarLander-v2

Figure 1 tensorboard of LunarLander-v2

```
(hw6) pp037@ec037:~/Desktop/hw6/src$ python dqn.py --test_only
Start Testing
Episode:0, Reward:234.6862345676707
Episode:1, Reward:252.1798577150574
Episode:2, Reward:274.7925940533863
Episode:3, Reward:149.50941795218296
Episode:4, Reward:291.10684234339163
Episode:5, Reward:265.5909730575328
Episode:6, Reward:273.91324834957646
Episode:7, Reward:278.6824486327986
Episode:8, Reward:280.68010498198186
Episode:9, Reward:282.77866021567667
Average Reward 258.39203818692556
```

Figure 2 testing result of LunarLander-v2

screenshot of tensorboard and testing results on LunarLanderContinuous-v2

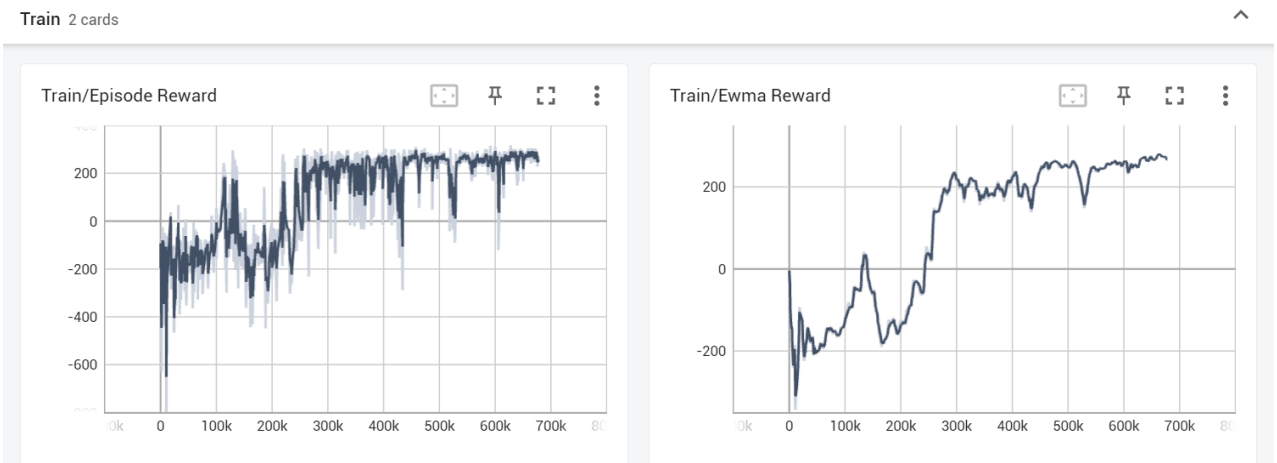


Figure 3 tensorboard of LunarLanderContinuous-v2

```
(hw6) pp037@ec037:~/Desktop/hw6/src$ python ddpg.py --test_only
Start Testing
Episode:0, Reward:230.06281835716948
Episode:1, Reward:278.5419132398696
Episode:2, Reward:268.983744452275
Episode:3, Reward:274.15104661537566
Episode:4, Reward:287.76808797824543
Episode:5, Reward:254.4746248662808
Episode:6, Reward:281.46213604239847
Episode:7, Reward:277.9789605466485
Episode:8, Reward:294.14516108347306
Episode:9, Reward:273.3270045217545
Average Reward 272.08954977034904
```

Figure 4 testing result of LunarLanderContinuous-v2

screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4

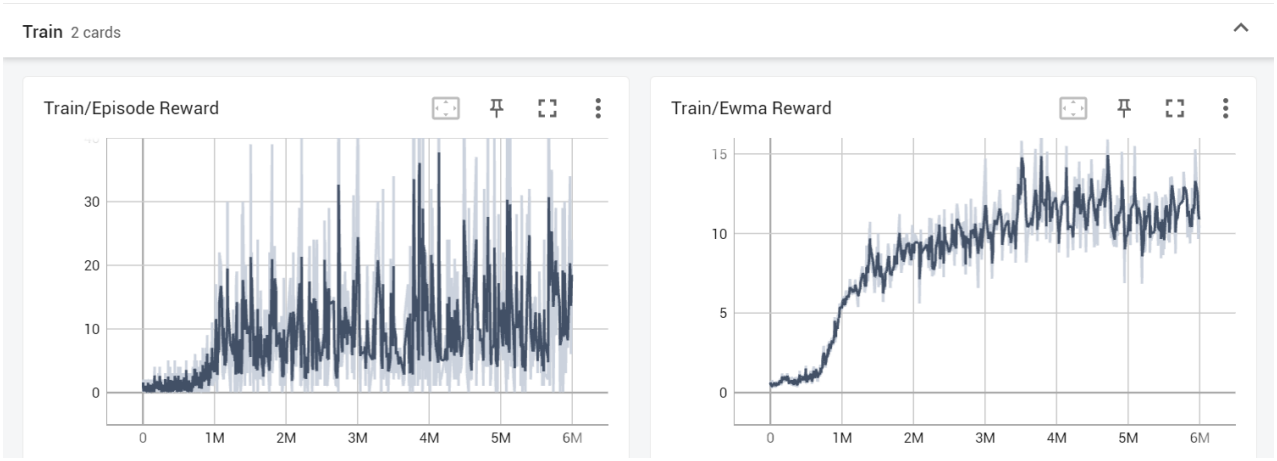
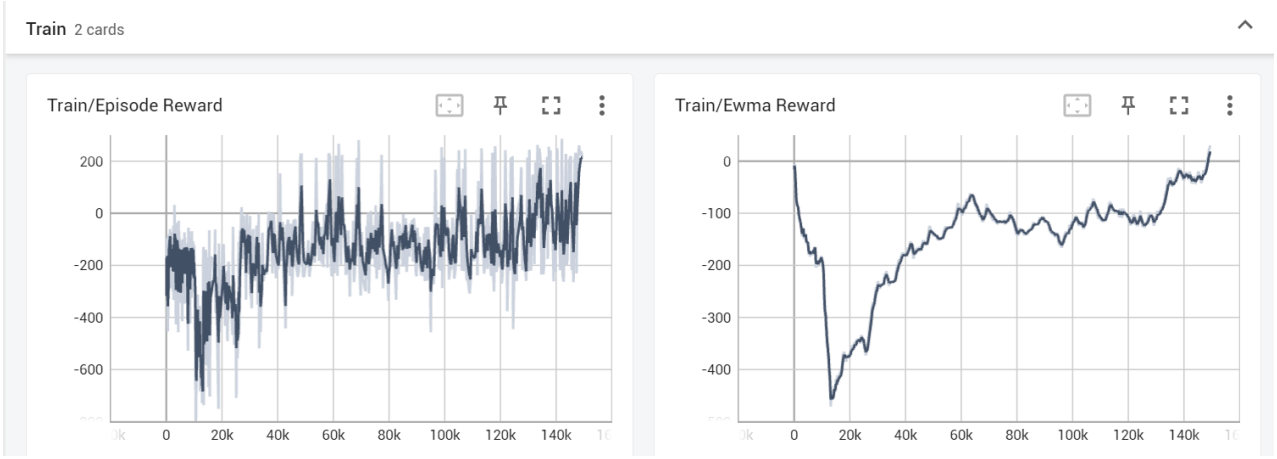


Figure 5 tensorboard of BreakoutNoFrameskip-v4 (橫軸為 total step)

```
(hw6) pp037@ec037:~/Desktop/hw6/src$ python dqn_breakout.py --test_only
Start Testing
episode 1: 422.00
episode 2: 423.00
episode 3: 422.00
episode 4: 422.00
episode 5: 383.00
episode 6: 421.00
episode 7: 421.00
episode 8: 656.00
episode 9: 422.00
episode 10: 418.00
Average Reward: 441.00
```

Figure 6 testing result of BreakoutNoFrameskip-v4

(bonus) screenshot of tensorboard and testing results on DDQN



```
(hw6) pp037@ec037:~/Desktop/hw6/src$ python ddqn.py --test_only
Start Testing
Episode:0, Reward:211.45249293813924
Episode:1, Reward:252.23085181921323
Episode:2, Reward:160.85867805823352
Episode:3, Reward:220.97176063616348
Episode:4, Reward:224.8199330980246
Episode:5, Reward:-205.8874734094282
Episode:6, Reward:-183.2989966308245
Episode:7, Reward:-27.582009591084145
Episode:8, Reward:-160.01375124160057
Episode:9, Reward:-1.3226414640854216
Average Reward 49.222884421275126
```

2. Question

Q1: Describe your major implementation of both DQN and DDPG in detail

在這部分討論 **d**eep **Q**-**n**etwork (DQN)和 **d**eep **d**eterministic **p**olicy **g**radient (DDPG)的程式碼實作細節。這兩個實驗的設定如下：DQN 使用 Adam optimizer，learning rate 為 0.0005，batch size 為 128，warmup iteration 為 10000 iteration，共訓練 1200 個 episodes。DDPG 也使用 Adam optimizer，其 actor 和 critic network 的 learning rate 皆設為 0.001，batch size 為 64，warmup iteration 同樣為 10000 iteration，共訓練 2000 個 episodes。兩者的 discount factor γ皆設置為 0.99。

- **Your implementation of Q network updating in DQN**

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(400,300)):# default hidden_dim=32
        super().__init__()

        ## TODO ##
        self.fc1 = nn.Linear(in_features=state_dim, out_features=hidden_dim[0])
        self.fc2 = nn.Linear(in_features=hidden_dim[0], out_features=hidden_dim[1])
        self.fc3 = nn.Linear(in_features=hidden_dim[1], out_features=action_dim)
        # self.fc1 = nn.Linear(in_features=state_dim, out_features=hidden_dim)
        # self.fc2 = nn.Linear(in_features=hidden_dim, out_features=hidden_dim)
        # self.fc3 = nn.Linear(in_features=hidden_dim, out_features=action_dim)
        self.relu = nn.ReLU()


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

Figure 7 implementation of value network in DQN

Figure 7 為 DQN 中的 value network 架構，可以看到 network 由三個全連接層(fully-connected layer)組成，除了最後一層之外，前兩層全連接層後面都加上 ReLU 作為 activation function。Q network 的輸入為 dimension=8 的 state 向量，通過三層全連接層把 state 轉換到維度更高的 hidden representation，最後再輸出 dimension=4 的向量(因為 LunarLander-v2 遊戲中，action space 中共有四種動作可選擇，分別為 No-op, Fire left engine, Fire main engine, Fire right engine)。原本 sample code 中 default 是設定 hidden representation 的 dimension 為 32，但是在這個設定之下我發現實驗結果不如預期，在訓練中後期 agent 的決策很不穩定，得到的 reward 也很低。推測是因為 hidden dimension 太低，mapping 空間太少不足以代表 state 到 action 之間的關係，因此我參考 DDPG sample code 中的方法，將 network 的第一個 hidden dimension 設定為 400，第二個設定為 300。這樣一來，state (輸入)到 action (輸出)的維度轉換就變成 8→400→300→4，可以達到理想的訓練結果。

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##

    # exploitation: input state into behavior net and choose the action with highest probability
    if random.random() > epsilon:
        with torch.no_grad():
            input_state = torch.from_numpy(state).to(self.device)
            action_prob = self._behavior_net(input_state)
            action = torch.argmax(action_prob).item()
    # exploration: random choose an action
    else:
        action = action_space.sample()

    return action
```

Figure 8 implementation of ε-greedy action selection in DQN

Figure 8 為 DQN 中 ε-greedy action selection 的程式碼實作。根據預先定義好的探索機率 ε，agent 會有 ε 的機率隨機從 action space (四種動作)中選擇一個動作當作決策，這個行為被稱作 exploration，目的是讓 agent 有機會去探索沒有嘗試過的決策，而不是一直完全照著過往經驗做決策。至於另外(1-ε)的機率，agent 會根據 behavior network 估計出來的 value，選擇按照過往經驗最好的 action 當作決策，這個行為稱為 exploitation。

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

    # get minibatch q_value prediction for their corresponding action
    q_value = torch.gather(input=self._behavior_net(state), index=action.long(), dim=1)
    with torch.no_grad():
        # get highest action value of minibatch with next state as input of target net
        q_next = torch.max(self._target_net(next_state), dim=1)[0]
        # *(1-done), if done=True=1, q_target should be reward only
        q_target = reward + gamma * torch.unsqueeze(q_next, dim=1) * (1-done)
    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()

def _update_target_network(self):
    '''update target network by copying from behavior network'''
    # copy behavior network's parameters as target network's
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

Figure 9 implementation of updating behavior/target network in DQN

Figure 9 為 DQN 中更新 behavior network 和 target network 的實作過程。首先在更新 behavior network 中，先從 replay memory 中隨機抽樣一個 mini-batch (裡面包含M個 transition，每個 transition 都記錄之前觀察過的 state, action, reward, next state, done 組合)，這種方法被稱作 experience replay。
給定第 i 個 transition observation (s_i, a_i, r_i, s_{i+1})，target value 可以用以下公式計算：

$$t_i = \begin{cases} r_i & ,if\ episode\ done\ at\ step\ (i+1) \\ r_i + \gamma \max_{a'} Q_t(s_{i+1}, a'; \theta_t) & ,otherwise \end{cases} \quad (1)$$

公式(1)中，γ代表 discount factor，Q_t表示參數為θ_t的 target network，a'則是 action space 中可能的 next action。由於模型的訓練目的是去最小化 target value t_i和 behavior network 計算出來的 estimate value，因此可以將 loss function 定義如下：

$$\mathrm{L} = \frac{1}{M}\sum_{i=1}^{M}\left(t_i - Q(s_i, a_i; \theta)\right)^2 \quad (2)$$

公式(2)為 loss function L的計算方法，採用 Mean Square Error (MSE)，Q表示參數為θ的 behavior network。根據實驗預先定義好的更新頻率，behavior network 每 4 個 iteration 更新一次參數，target network 則每 1000 個 iteration 更新一次，複製當下 behavior network 的參數，以避免更新得太頻繁讓 target value 計算結果不夠穩定。

- **Your implementation of actor and critic network in DDPG**

由於 DDPG 中 actor 和 critic 的 behavior network 和 target network 更新是寫在一起的，因此本部分會先一起介紹 actor, critic network 的架構、behavior network 和 target network 的更新過程，後面再分開介紹 gradient of actor updating 和 gradient of critic updating 的算法。

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(in_features=state_dim, out_features=hidden_dim[0])
        self.fc2 = nn.Linear(in_features=hidden_dim[0], out_features=hidden_dim[1])
        self.fc3 = nn.Linear(in_features=hidden_dim[1], out_features=action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

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

Figure 10 implementation of actor network in DDPG

Figure 10 為 actor network 的實作程式碼，network 由三個全連接層(fully-connected layer)組成，前兩層的 activation function 為 ReLU，最後一層的 activation function 是 tanh。第一個 hidden representation 的 dimension 為 400，第二個 hidden representation 的 dimension 為 300。Actor network 根據當下的 input state，從 LunarLanderContinuous-v2 的 action space 中預測出一個 action。再藉由 critic network (如下一部分介紹)，來根據 state 和預測 action 估計出 Q-value (代表當下與未來 reward 的總和)。總結來說，Actor network 的輸入為 dimension=8 的 state 向量，通過三層全連接層把 state 轉換到維度更高的 hidden representation，最後再輸出 dimension=2 的向量，用來代表預測的 action (分別為 Main engine, Left-Right 的連續控制值)。

```python
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, h1),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, 1),
        )

    def forward(self, x, action):
        x = self.critic_head(torch.cat([x, action], dim=1))
        return self.critic(x)
```

Figure 11 implementation of critic network in DDPG

Figure 11 為 critic network 的實作程式碼，network 由三個全連接層(fully-connected layer)組成，前兩層的 activation function 為 ReLU。第一個 hidden representation 的 dimension 為 400，第二個 hidden representation 的 dimension 為 300。Critic network 將當下 state 和 actor network (如上一部分描述)預測出來的 action 兩個當作輸入，估計出一個一維的 Q-value (代表當下與未來 reward 的總和)。

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        input_state = torch.from_numpy(state).to(self.device)
        if noise:
            action_noise = torch.from_numpy(self._action_noise.sample()).to(self.device)
            action = self._actor_net(input_state) + action_noise
        else:
            action = self._actor_net(input_state)
    return action
```

Figure 12 implementation of action selection in DDPG

Figure 12 為 DDPG 實驗中，action selection 的實作程式碼。由於在 DDPG 實驗中，要處理的是連續(continuous)的 action space，因此沒辦法使用像 DQN 中的 greedy action selection 方法。如同 Figure 10 中的描述，必須使用 action network 來選擇 action，另外為了鼓勵 agent 探索更多可能的動作決策，因此在 actor network 預測的動作(連續數值)上添加了從高斯分布中抽樣的 noise 作為 exploration noise，可以用公式表達如下：

$$a_t = \mu(s_t|\theta_\mu) + N_t \tag{3}$$

公式(3)中，a_t是加上 exploration noise 的動作決策，$\mu(s_t|\theta_\mu)$是 actor network(參數為θ_μ)根據當下 state s_t預測出的動作，N_t則表示從高斯分布中抽樣出來的 exploration noise。

```python
def _update_behavior_network(self, gamma):
    actor_net, critic_net, target_actor_net, target_critic_net = self._actor_net, self._critic_net, self._target_actor_net, self._target_critic_net
    actor_opt, critic_opt = self._actor_opt, self._critic_opt

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

    ## update critic ##
    # critic loss
    ## TODO ##

    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)

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

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

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

Figure 13 implementation of updating behavior network in DDPG

Figure 13 為 DDPG 中，更新 behavior network 的實作過程。模型的 gradient 推導以及 loss function 的定義在後面會詳細描述。

```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)
```

Figure 14 implementation of updating target network in DDPG

Figure 14 為 DDPG 中，更新 target network 的實作過程。跟 DQN 不同，在 DDPG 中使用了 soft target updating 的方法來更新 target network 的參數。在這裡，針對每一個 behavior network 和 target network pairs (即 actor 和 critic 兩個的 network pairs)，裡面的參數更新如下：

$$\theta_t = \tau\theta + (1-\tau)\theta_t \tag{4}$$

公式(4)中，$\tau \ll 1$是一個 hyper-parameter，用來決定想要保留原始參數的比例，θ和θ_t則分別代表 behavior network 和 target network 的參數集合。

- **The gradient of actor updating in DDPG**

將 actor network 定義為μ，其參數為θ_μ；將 critic network 定義為Q，其參數為θ_Q。actor network 應該朝著 critic network 預測的建議方向(policy)來更新參數。首先，從 replay memory buffer 中隨機抽樣一個 mini-batch，裡面包含M個 transition，每個 transition 都記錄之前觀察過的 state, action, reward, next state 組合，即(s_i, a_i, r_i, s_{i+1})。在給定當前狀態s_i的情況下，可從 actor network 獲得 action $\mu(s_i|\theta_\mu)$。接著，通過 critic network 計算 Q-value $Q(s_i, \mu(s_i|\theta_\mu); \theta_Q)$。Action network 的目的是選擇一個可以最大化 critic network 計算出來的 Q-value 的 action 當作決策，因此可以將 loss function 定義如下：

$$L_{\mu} = -\frac{1}{M}\sum_{i} Q\left(s_i, \mu\left(s_i|\theta_{\mu}\right); \theta_Q\right) \tag{5}$$

公式(5)中，對 M 個樣本計算 Q-value 平均，並加總作為 actor network 的 loss function。
可以進一步推導 actor network 的 gradient 如下：

$$\begin{aligned}\nabla_{\theta_{\mu}} L_{\mu} &= -\frac{1}{M}\sum_{i} \nabla_{\theta_{\mu}} Q\left(s_i, \mu\left(s_i|\theta_{\mu}\right); \theta_Q\right) \\ &= -\frac{1}{M}\sum_{i} \nabla_a Q\left(s, a; \theta_Q\right)|_{s=s_i, a=\mu\left(s_i|\theta_{\mu}\right)} \nabla_{\theta_{\mu}} \mu\left(s|\theta_{\mu}\right)|_{s=s_i}\end{aligned} \tag{6}$$

公式(6)為 gradient of actor updating 公式，其程式碼實作如 Figure 13 中所示。

- **The gradient of critic updating in DDPG**

將 critic 的 target network 定義為Q_t，其參數為θ_{Q_t}；並將 actor 的 target network 定義為μ_t，其參數為θ_{μ_t}。對於上一部分中提到的 mini-batch 抽樣中，第 i 個 transition 的 target Q-value (從 critic network 中估計得出)，可以表示如下：

$$t_i = r_i + \gamma Q_t\left(s_{i+1}, \mu_t\left(s_{i+1}|\theta_{\mu_t}\right); \theta_{Q_t}\right) \tag{7}$$

公式(7)表示 target Q-value 的計算方法，其中γ是 discount factor。Critic network 的目標是在全部總共 M個 transition 中，去最小化 Q-value $Q\left(s_i, a_i; \theta_Q\right)$和 target value t_i之間的差異。因此 critic network 的 loss function 可以表達如下：

$$L_Q = \frac{1}{M}\sum_{i} \left(t_i - Q\left(s_i, a_i; \theta_Q\right)\right)^2 \tag{8}$$

公式(8)為 critic network loss function L_Q的計算方法，採用 Mean Square Error (MSE)。
可以進一步推導 critic network 的 gradient 如下：

$$\begin{aligned}\nabla_{\theta_Q} L_Q &= \frac{1}{M}\sum_{i} \nabla_{\theta_Q} \left(t_i - Q\left(s_i, a_i; \theta_Q\right)\right)^2 \\ &\approx \left(r_i + \gamma Q_t\left(s_{i+1}, \mu_t\left(s_{i+1}|\theta_{\mu_t}\right); \theta_{Q_t}\right)\right) - Q\left(s_i, a_i; \theta_Q\right) \nabla_{\theta_Q} Q\left(s_i, a_i; \theta_Q\right)\end{aligned} \tag{9}$$

公式(9)為 gradient of critic updating 公式，其程式碼實作如 Figure 13 中所示。

Q2: Explain effects of the discount factor

一個 agent 的 discounted reward G_t的 general form 可以表示如下：

$$\begin{aligned}G_t &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots \\ &= R_t + \gamma\left(R_{t+1} + \gamma R_{t+2} + \cdots\right)\end{aligned} \tag{10}$$

公式(10)中，$\gamma < 1$是 discount factor，用以決定 estimated future rewards 的權重。Discount factor 如果較小，可以鼓勵 agent 更關注當前 reward，而較大的 discount factor 則會減少 long-term reward 的影

響力。在 DQN 和 DDPG 中，estimated future rewards 由 deep neural network 計算出的 Q-value 表示。例如 DDPG 中 critic network 的 target Q-value (公式(7)所示)，其中的$Q_t(s_{i+1}, \mu_t(s_{i+1}|\theta_{\mu_t}); \theta_{Q_t})$就表示 estimated future rewards。

Q3: Explain benefits of epsilon-greedy in comparison to greedy action selection

ε-greedy 是一種平衡 exploration 跟 exploitation 的方法。在每一個 timestep，agent 會以預先定義好的探索機率 ε 隨機從 action space 選擇一個動作當作決策。另外(1-ε)的機率，agent 會根據 behavior network 估計出來的 value，選擇按照過往經驗最好的 action 當作決策。

ε-greedy 做 exploration 可以讓 agent 有機會去探索沒有嘗試過的決策，而不是一直完全照著過往經驗做決定，以避免一直選擇最佳動作會錯過其他真正更好的選擇。

Q4: Explain the necessity of the target network

在訓練 DQN 和 DDPG 時，每個 state 對應的 Q-value 是通過一個 Q network 或 critic network 估計得到。跟 Q-learning 不同，exact value function 被 deep neural network 組成的 function approximator 取代。在這種設計下，即使在 Q network 上只進行一次 optimization 過程，同樣 state 對應的 Q-value 也會有所改變，這可能會導致 agent 的訓練過程不夠穩定。為了解決這個問題，會需要多引入一個穩定的 network 來解決這個問題，即採用 target network，只在固定的 iteration 間隔後更新一次參數，其他時候 network 中的參數固定，讓使用 target network 預測出來的 Q-value 也會比較穩定。

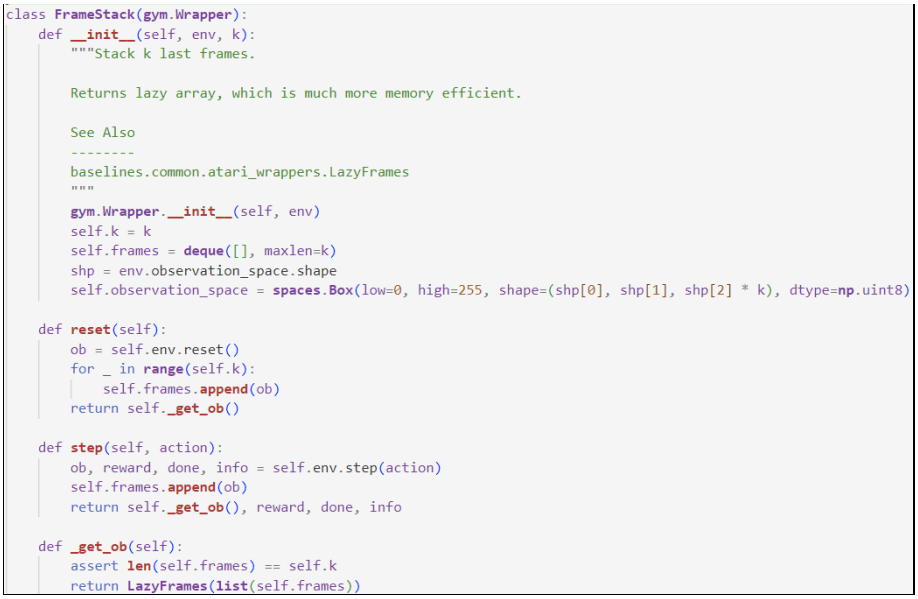
Q5: Describe the tricks you used in Breakout and their effects, and how they differ from those used in LunarLander

```
class Net(nn.Module):
    def __init__(self, num_classes=4, init_weights=True):
        super(Net, self).__init__()

        self.cnn = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4),
                                 nn.ReLU(True),
                                 nn.Conv2d(32, 64, kernel_size=4, stride=2),
                                 nn.ReLU(True),
                                 nn.Conv2d(64, 64, kernel_size=3, stride=1),
                                 nn.ReLU(True)
                                 )
        self.classifier = nn.Sequential(nn.Linear(7*7*64, 512),
                                        nn.ReLU(True),
                                        nn.Linear(512, num_classes)
                                        )
        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        x = x.float() / 255.
        x = self.cnn(x)
        x = torch.flatten(x, start_dim=1)
        x = self.classifier(x)
        return x
```

Figure 15 implementation of value network in Breakout



```python
class FrameStack(gym.Wrapper):
    def __init__(self, env, k):
        """Stack k last frames.

        Returns lazy array, which is much more memory efficient.

        See Also
        --------
        baselines.common.atari_wrappers.LazyFrames
        """
        gym.Wrapper.__init__(self, env)
        self.k = k
        self.frames = deque([], maxlen=k)
        shp = env.observation_space.shape
        self.observation_space = spaces.Box(low=0, high=255, shape=(shp[0], shp[1], shp[2] * k), dtype=np.uint8)

    def reset(self):
        ob = self.env.reset()
        for _ in range(self.k):
            self.frames.append(ob)
        return self._get_ob()

    def step(self, action):
        ob, reward, done, info = self.env.step(action)
        self.frames.append(ob)
        return self._get_ob(), reward, done, info

    def _get_ob(self):
        assert len(self.frames) == self.k
        return LazyFrames(list(self.frames))
```

Figure 16 frame stack in Breakout

Figure 15 為 Breakout 裡面的 value network 架構，跟 DQN 實驗不同，Breakout 接受的 input state 是整張圖像，而不是 dimension=8 的 state vector。因此架構中引入 CNN 提取圖片資訊。並且根據這次 lab 的提示，以 atari_wrappers.py 將 4 個 frame 組合成一個 state 再輸入 value network (如 Figure 16)，以避免單一 frame 作為 state 輸入，會讓模型無法學習到 frame 前後的狀態。

```python
def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            if m.bias is not None:
                nn.init.constant_(m.bias, 0.0)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1.0)
            nn.init.constant_(m.bias, 0.0)
        elif isinstance(m, nn.Linear):
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            nn.init.constant_(m.bias, 0.0)
```

Figure 17 initialize value network in Breakout

Figure 17 為初始化 Breakout value network 的程式碼，用來將模型各層的初始參數調整到某個分布內，保證初始參數不會過於偏離，讓訓練過程可以更加穩定。

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

    ## TODO ##
    # get minibatch q_value prediction for their corresponding action
    q_value = torch.gather(input=self._behavior_net(state), index=action.long(), dim=1)
    with torch.no_grad():
        # get highest action value of minibatch with next state as input of target net
        q_next = torch.max(self._target_net(next_state), dim=1)[0]
        # *(1-done), if done=True=1, q_target should be reward only
    #  TODO
        q_target = reward + gamma * torch.unsqueeze(q_next, dim=1) * (1-done) - done
```

Figure 18 trick for calculating target Q-value in Breakout

Figure 18 為訓練 Breakout 的一個 trick。一開始我使用與 DQN 相同的方法來更新 behavior network 和其他部分，只更改了 input state 從 vector 改為 frame stack。但實驗結果發現 sequence length 很短，每

次 iteration 遊戲都很快就結束，也因此 total reward 較低無法上升至理想結果。
經過推測，我認為是因為在 target Q-value 中，沒有加上對遊戲結束的懲罰。因為在 LunarLander 遊戲中，遊戲結束快慢對最後的 total reward 不會有特殊的影響。但在 Breakout 遊戲中，理論上會希望遊戲可以持續越久越好，盡量不要太快結束，也就有機會取得更多 reward。
因此我在 target Q-value 的計算中，加上一項$(-\text{done})$。如果遊戲結束則$\text{done} = 1$，target Q-value 會因此降低，鼓勵遊戲往不會結束的方向去學習。通過觀察實驗結果也可以發現，多加了此項懲罰後，有助於提升遊戲的 sequence length，total reward 也可以順利上升到理想結果。