Deep Learning

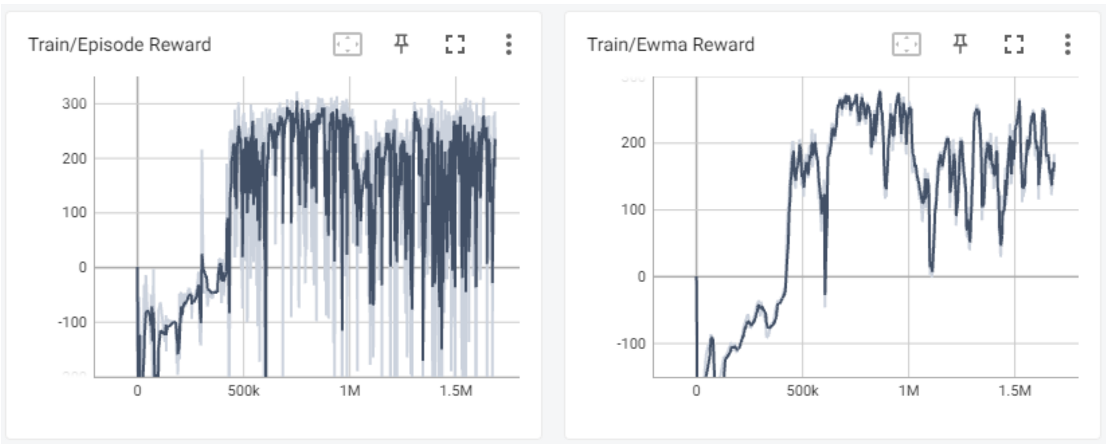
Lab6: Deep Q-Network and Deep Deterministic Policy Gradient

311512064 鄧書桓

■ Experimental Results

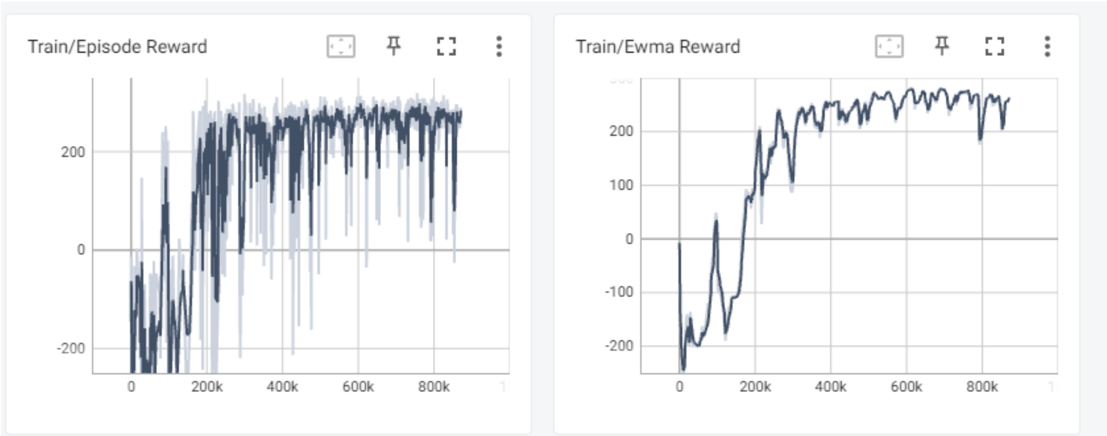
Your screenshot of tensorboard and testing results on LunarLander-v2

```
(base) pp037@ec037:~/DL/lab6$ python dqn-other.py --test_only
/home/pp037/anaconda3/lib/python3.7/site-packages/gym/logger.py
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yello
Start Testing
episode 0: 292.2429947464426
episode 1: 230.37844147099975
episode 2: -239.88354586915597
episode 3: 246.96302934684286
episode 4: 256.0875614597556
episode 5: 229.26159004917918
episode 6: 16.14972087132658
episode 7: 265.3298578100195
episode 8: 226.48973173414527
episode 9: 283.1850867255862
Average Reward 180.62044683451413
```



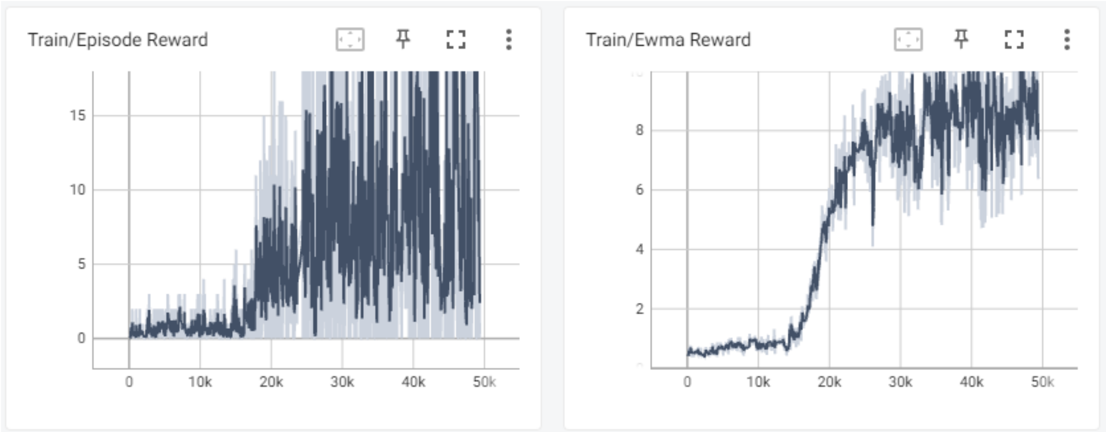
Your screenshot of tensorboard and testing results on LunarLanderContinuous-v2.

```
(base) pp037@ec037:~/DL/lab6$ python ddpg-other.py --test_only
/home/pp037/anaconda3/lib/python3.7/site-packages/gym/logger.py:30:
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
Start Testing
episode 0: 237.05909460680311
episode 1: 274.1095697198743
episode 2: 262.8711878636186
episode 3: 265.6663328991155
episode 4: 303.3701502198901
episode 5: 256.1952609635732
episode 6: 295.82962624140697
episode 7: 289.33367115103505
episode 8: 301.86299594914306
episode 9: 294.55221484220544
Average Reward 278.08501044566657
```



Your screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4

```
(base) pp037@ec037:~/DL/lab6$ python dqn_breakout_other.py --test_only
Start Testing
episode 1: 266.00
episode 2: 161.00
episode 3: 223.00
episode 4: 225.00
episode 5: 155.00
episode 6: 249.00
episode 7: 216.00
episode 8: 241.00
episode 9: 291.00
episode 10: 197.00
Average Reward: 222.40
```



■ Questions

- ■ Describe your major implementation of both DQN and DDPG in detail

1. Your implementation of Q network updating in DQN.

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super(Net, self).__init__()
        ## TODO ##
        self.layer1 = nn.Linear(state_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.layer3 = nn.Linear(hidden_dim, action_dim)


    def forward(self, x):
        ## TODO ##
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

上圖為 Q network 的架構，此網路主要功能為找尋該 state 下各個 action 的 Q-value，因此輸入為 state_dim 輸出為 action_dim。

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

    rnd = random.random()
    if rnd < epsilon:
        return np.random.randint(action_space.n)
    else:
        state = torch.from_numpy(state).float().unsqueeze(0).cuda()
        with torch.no_grad():
            # actions_value = self._behavior_net.forward(state)#state as input out put is action
            action_test = self._behavior_net(state).max(1)[1].view(1, 1).item()
        # action = np.argmax(actions_value.cpu().data.numpy()) #take max q action as action
        action = action_test

    return action
```

此處為選擇動作，因為是使用 epsilon-greedy policy 來選擇，

故有一定機率是隨機行動，一定機率是選擇當下最大 Q 值得

action。

```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 ##


    q_value = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
        q_next= self._target_net(next_state).max(1)[0]
        # view let q_target be same dim with q_value
        q_target = (q_next.view(self.batch_size, 1) * gamma) + reward

    # Compute Huber loss
    # criterion = nn.SmoothL1Loss()
    criterion = nn.MSELoss()
    # .unsqueeze(1)將向量的維度從一維擴展到二維
    loss = criterion(q_value, q_target)

    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    # 將 policy_net 的參數梯度值限制在 -5 到 5 之間。這個函式可以避免梯度爆炸的問題
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

這邊的 update 概念為先讓 behavior net(每一步皆會更新)更

新一千次後，再將參數複製給 target net，這樣更新的策略

是為了避免 bootstraping 的問題，此問題會導致一種正反饋的效應，即小的誤差會被放大並積累，進一步影響 Q-network 的學習和收斂。

2&3. Your implementation and the gradient of actor & critic updating in DDPG

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        h1, h2 = hidden_dim
        self.layer1 = nn.Linear(state_dim, h1)
        self.layer2 = nn.Linear(h1, h2)
        self.layer3 = nn.Linear(h2, action_dim)

    def forward(self, x):
        ## TODO ##
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = torch.tanh(self.layer3(x))
        return x
```

此為 actor 的網路架構，用來生成當下最大 Q-value 的 action，由於 DDPG 可以處理連續且多個 aciton，這邊會輸出多的 action。

```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, action_dim),
        )

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

此為 critic 的架構，用來生成該 action 的 q-value，因此輸入為 action 輸出為 q-value(dim = 1)。

```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 ##
    ## TODO ##
    # 先更新critic_net再更新actor_net
    # critic loss
    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 + (self.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)
    # 梯度上升，這會讓Q value變大
    actor_loss = - critic_net(state, action).mean()
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

上述為 actor 與 critic 的更新部分，這裡的更新策略為先更新 critic 再更新 actor，原因是因為 critic 本身包含 actor 的輸出，因此從最後面更新回來。在更新 actor 時要比較注意，這邊是使用反向梯度來做 update，用來使輸出的 Q-value 會最大。

- Explain effects of the discount factor

Discount factor 的功能是讓離線在 state 越遠的 TD error 對現在

的影響越小。其數學式如下:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0} \gamma^k R_{t+k+1}$$

但由於本 lab 只使用 one step 的 TD error，因此此作用並不明顯。

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

epsilon-greedy 的主要功能是為了讓 agent 可以去探索環境，若每次都是根據最大 Q-value 的 action 去做動作，會讓 agent 僅做一開始認為最優的策略，雖然乍聽之下很合理，但怎麼知道他所選擇的 action 是真的最好的，以整體來看，有些較佳的結果在一開始的得分並不一定是最高的，若沒使用 epsilon-greedy 來探索更多可能的話，會導致僅會走一開始 Q-value 最高的策略，而使整體分數降低。

- Explain the necessity of the target network.

主要功能是為了避免一直更新 behavior network，這樣更新的策略是為了避免 bootstraping 的問題，此問題會導致一種正反饋的效應，即小的誤差會被放大並積累，進一步影響 Q-network 的學習和收斂。

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

這邊由於改成是根據整張遊戲的圖片去訓練生成 action、next

state、reward 和 done，所以會需要多一些步驟。首先是在 buffer 的定義與宣告，要多加以下步驟:

```python
def __init__(self, capacity):

    self.position = 0
    self.size = 0

    # 初始化buffer
    # 一次包含五禎(前四個是當下的state，後4禎是下個state)
    # 84*84是該圖的解析度
    c,h,w = 5, 84, 84
    self.capacity = capacity
    self.m_states = torch.zeros((capacity, c, h, w), dtype=torch.uint8)
    # 根據前四禎決定action
    self.m_actions = torch.zeros((capacity, 1), dtype=torch.long)
    # 根據action決定的reward
    self.m_rewards = torch.zeros((capacity, 1), dtype=torch.int8)
    self.m_dones = torch.zeros((capacity, 1), dtype=torch.bool)
```

這邊的設計想法是一次儲存 5 個 frame，前四張為當下的 state(這邊被 atari_wrapper 包裝成 DeepMind 的型態，一個 state 一次會輸出 4 禎，後三禎為一樣的 action)，後四禎為 next state，而 action 與 reward 設計為由第四禎與第五禎的關係所得。另外，為了讓一開始擁有足夠的 frame(至少 5 禎)，這邊對一開始在發射後有跑 action0(no-op)多次，使其符合所設計的 deque 大小。程式設計如下:

```python
# 先開火然後不動9個state，已獲得足夠的deque
for i in range(10): # no-op
    if i == 0:
        state, _, _, _ = env.step(1)
        n_frame = torch.from_numpy(state)
        h = n_frame.shape[-2]
        n_frame = n_frame.view(1,h,h)
        frame_10.append(n_frame)
    else:
        state, _, _, _ = env.step(0)
        n_frame = torch.from_numpy(state)
        h = n_frame.shape[-2]
        n_frame = n_frame.view(1,h,h)
        frame_10.append(n_frame)
```