DLP HW6

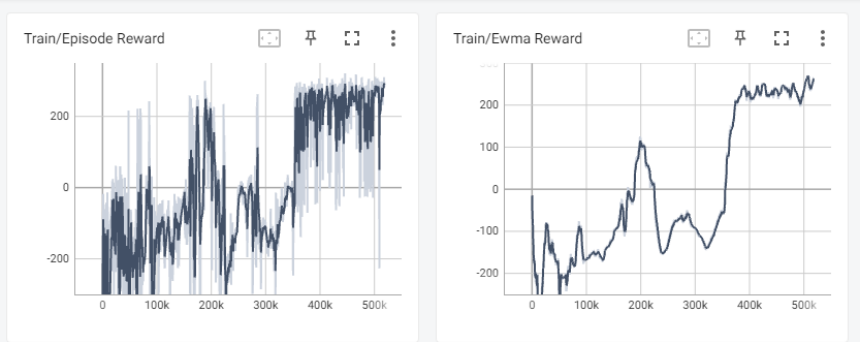
數據所 311554019 宋沛潔

1. The screenshot of tensorboard and testing results on LunarLander-v2:

- DDPG:
  1. Testing results:

```
Start Testing
Episode: 0      Total reward: 246.25
Episode: 1      Total reward: 277.18
Episode: 2      Total reward: 276.51
Episode: 3      Total reward: 269.20
Episode: 4      Total reward: 294.21
Episode: 5      Total reward: 272.64
Episode: 6      Total reward: 289.73
Episode: 7      Total reward: 171.66
Episode: 8      Total reward: 188.48
Episode: 9      Total reward: 254.63
Average Reward 254.04842568562708
```

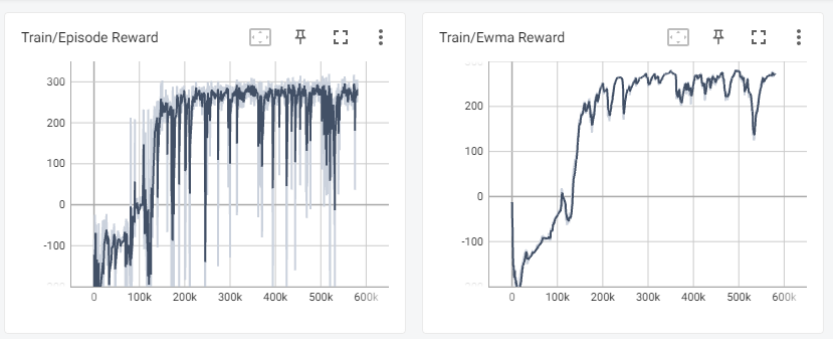
  2. Tensorboard



- DQN
  1. Testing results:

```
Start Testing
Episode: 0      Total reward: 284.48
Episode: 1      Total reward: 294.43
Episode: 2      Total reward: 280.39
Episode: 3      Total reward: 304.09
Episode: 4      Total reward: 243.91
Episode: 5      Total reward: 282.68
Episode: 6      Total reward: 283.87
Episode: 7      Total reward: 229.52
Episode: 8      Total reward: 294.35
Episode: 9      Total reward: 279.68
Average Reward 277.739862397647
```

2. Tensorboard:

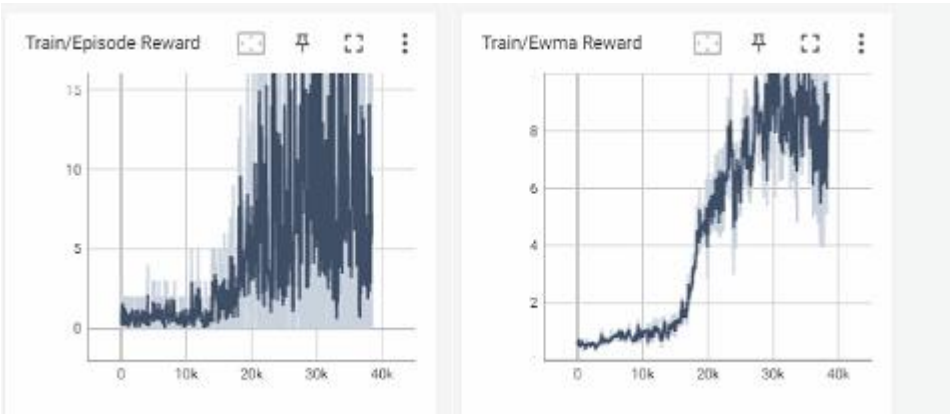


2. The screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4:

1. Testing results:

```
Start Testing
episode 1: 396.00
episode 2: 396.00
episode 3: 421.00
episode 4: 417.00
episode 5: 355.00
episode 6: 379.00
episode 7: 422.00
episode 8: 424.00
episode 9: 421.00
episode 10: 393.00
Average Reward: 402.40
```

2. Tensorboard:



3. Questions:

3.1. Describe both DQN and DDPG in detail

- **DQN**:
  Behavior network 會根據當前 state 選擇 action，而 target network 則是

用來計算目標值。其中，Behavior network 目標是使用 replay memory 來存儲過去 agent 在環境的經驗，透過在訓練過程中進行隨機抽樣。並用下方公式計算目標值，在以 MSE 更新網路。

$$Y_t^Q = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'|\theta)$$

再來是 target network，他主要是每幾個 steps 去複製 behavior network 權重，為的是穩定訓練過程，不會在每個 steps 都直接使用 behavior network 的目標值來更新網絡的權重

```
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 = torch.gather(self._behavior_net(state), dim=1, index=action.long())
    with torch.no_grad():
        q_next = torch.max(self._target_net(next_state), dim=1)[0].unsqueeze(dim=1)
        q_target = reward + gamma * (1 - done)* q_next
    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'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

如果隨機數小於 epsilon，隨機抽樣一個動作，如果大於 epsilon 會從 behavior network 來選擇一個最有可能是最大 Q 值的 action。這個設計可以同時實施 exploration 及 exploitation。在訓練過程中，希望能夠對動作空間進行 exploration，以發現更好的策略。

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if random.random() <= epsilon:
        action = action_space.sample()
    else:
        with torch.no_grad():
            input_state = torch.from_numpy(state).to(self.device)
            actions_vec = self._behavior_net(input_state)
            action = torch.argmax(actions_vec).item()
    return action
```

- **DDPG**:

  ActorNet 是去學習特徵，幫助網絡更好選擇 action。這邊設計兩個全連接層接著 ReLU activation function，最後一層接 Tanh activation function。

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.layer1 = nn.Linear(state_dim, hidden_dim[0])
        self.layer2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.layer3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

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

CriticNet 會根據當前 state 和由 ActorNet 選擇的 action，來估計 Q 值，也就是預測在該狀態下執行該動作的預期 Q 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)
```

Behavior network 會根據當前 state 選擇 action，而 target network 則是用來計算目標值。

首先，使用 critic network 計算當前 State 下的 action 的 Q 值。再根據 target critic network 計算下一個 State 及 target actor network 選擇的動作的 Q 值。接著使用 MSE 作為 loss function 來更新 behavior critic network，最後用 ActorLoss 來更新 behavior actor network。

```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 = 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)
    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 = self._actor_net(state)
    actor_loss = -self._critic_net(state, action).mean()
    # optimize actor
    actor_net.zero_grad()
    critic_net.zero_grad()
    actor_loss.backward()
    actor_opt.step()
```

在 target network 中直接複製權重可能會導致模型更新不穩定，所以使用 soft copy 來更新 target network。將 behavior network 的權重以一定的比例 τ 融合到 target network 的權重中。

```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 ##
        with torch.no_grad():
            target.copy_(tau * behavior + (1 - tau) * target)
```

3.2. Explain the necessity of the target network

Target network 他主要是每幾個 steps 去複製 behavior network 權重，為的是穩定訓練過程，不會在每個 steps 都直接使用 behavior network 的目標值來更新網絡的權重。這可以解決 overestimation，因為 behavior network 的估計值可能有時會過高，導致對於目標值的估計也會偏高，造成訓練結果不穩定。因此加了 target network 可以增加穩定性，使得模型能夠更好地學習到最佳策略。

3.3. Explain effects of the discount factor

discount factor 用來調整未來 reward 的價值。範圍在 0~1，在這份程式 gamma。如果 discount factor 接近 1，代表未來的 reward 對當前影響越大，來關注未來的 reward，幫助學到更長期的策略，更有機會得到更好的結果；而 discount factor 越接近 0，代表未來的獎勵對當前影響越小，agent 較在意當下即時策略。

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

epsilon-greedy 的概念是，如果隨機數小於 epsilon，隨機抽樣一個動作，如果大於 epsilon 會從 behavior network 來選擇一個最有可能是最大 Q 值的 action。這個設計可以同時實施 exploration 及 exploitation。在訓練過程中，希望能夠對動作空間進行 exploration，以發現更好的策略。

因此跟 greedy action selection 比較下，epsilon-greedy 更可以探索不同的策略來避免只有找到局部最佳解，能夠獲得更好的去學習最佳策略。

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

有使用助教 PPT 上提供的 frame stacking 的技術。讓模型看到遊戲的動態，而不是只有單一張。模型會將 4 個連續的 frame 堆疊在一起，形成一個具有時間序列的 state。

agent 在每一輪遊戲的開始時執行 10 次 Atari 遊戲中的 action 0。也就是給遊戲從初始 state 有一些隨機性，避免 agent 只有學會一種固定的遊戲開始策略，增加 exploration。

```python
def train(args, agent, writer):
    print('Start Training')
    env_raw = make_atari('BreakoutNoFrameskip-v4')
    env = wrap_deepmind(env_raw, episode_life=True, clip_rewards=True)
    action_space = env.action_space
    total_steps, epsilon = 0, 1.
    ewma_reward = 0
    queue = deque(maxlen=5)

    for episode in range(args.episode):
        total_reward = 0
        state = env.reset()
        state, reward, done, _ = env.step(1) # fire first !!!
        for _ in range(10):
            result = env.step(0)
            frames = frame(result[0])
            queue.append(frames)
```

```python
    for t in itertools.count(start=1):
        state = torch.cat(list(queue))[1:].unsqueeze(0).to(device)
        if total_steps < args.warmup:
            action = action_space.sample()
        else:
            # select action
            action = agent.select_action(state, epsilon, action_space)
            # decay epsilon
            epsilon -= (1 - args.eps_min) / args.eps_decay
            epsilon = max(epsilon, args.eps_min)
        # execute action
        frames, reward, done, _ = env.step(action)

        frames = frame(frames)
        queue.append(frames)

        ## TODO ##
        # store transition
        agent.append(torch.cat(list(queue)).unsqueeze(0), action, reward, done)

        total_reward += reward
        if total_steps >= args.warmup:
            agent.update(total_steps)
```