DL lab5 report 313552042 陳品翰

· Introduction (5%):

這份作業要用強化學習的方式來訓練 cartpole 跟 pong 這兩個遊戲，在這份報告裡面，我會表明這兩個遊戲分別是用哪些策略來訓練的，並且會比較不同的策略之間訓練結果的差異，我會嘗試分析為何會有這種差異存在，另外，我在訓練 pong 這個遊戲的時候，發現可以把 action space 縮小成(0,2,3)，因為其他的動作其實在遊玩的過程中並沒有作用，其他的動作只在開始遊戲的時候會有用，所以我把 action space 縮小後可以讓模型專注在學習有用的動作上，而不會浪費時間在無用的動作。

· Your implementation (20%):

Task1:

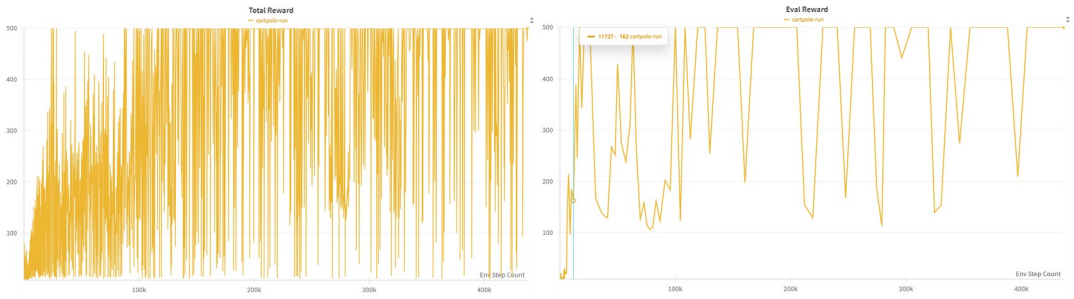
How do you obtain the Bellman error for DQN?

```
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
with torch.no_grad():
    target_q_values = self.target_net(next_states).max(1)[0]
    targets = rewards + self.gamma * target_q_values * (1 - dones)

loss = nn.MSELoss()(q_values, targets)
```

DQN 會有兩個 network，分別是 q_net 跟 target_net，我圖片中的 code 就是實作 Bellman error 的計算，會先依照當前的 state 跟動作得到 q value，之後用 target_net 來去計算 next_state 的 target q value，再依照 reward+(gamma*target_q_value)的公式來計算，後面的 1-done 是要判斷遊戲是否已經結束，loss function 我是採用 MSE。

Explain how you use Weight & Bias to track the model performance.
在網站中會有許多圖表，其中我會用 total reward/env_step 和 Eval reward/env_step 這兩個圖表來看 model 有沒有成功訓練。

Total reward 是指一個 episode 訓練完畢後的分數。



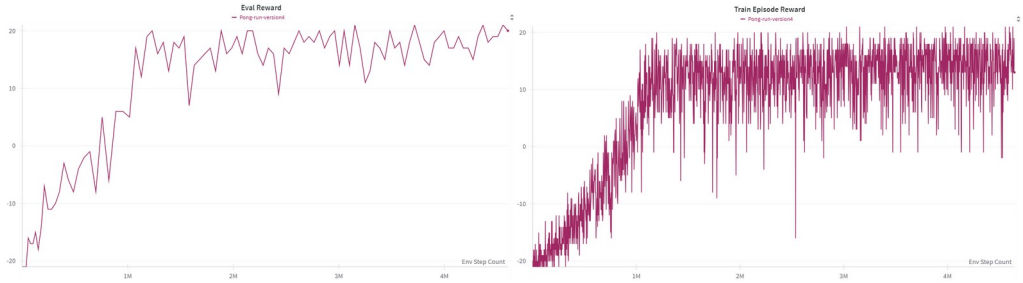
Task2:

How do you obtain the Bellman error for DQN?

```
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
with torch.no_grad():
    target_q_values = self.target_net(next_states).max(1)[0]
    targets = rewards + self.gamma * target_q_values * (1 - dones)

loss = nn.SmoothL1Loss()(q_values, targets)
self.optimizer.zero_grad()
loss.backward()
```

和 task1 類似，差別在於我這邊的 loss 是選用 smoothL1Loss。

Explain how you use Weight & Bias to track the model performance.



我是看 Eval Rewrad/env step 和 Train Episode Reward/env step 來判斷模型有沒有順利學習。

Task3:

How do you obtain the Bellman error for DQN?

```
states = torch.from_numpy(np.array(states).astype(np.float32)).to(self.device) / 255.0
next_states = torch.from_numpy(np.array(next_states).astype(np.float32)).to(self.device) / 255.0
actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
dones = torch.tensor(dones, dtype=torch.float32).to(self.device)
weights = weights.to(self.device)

q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
targets = compute_double_dqn_targets(self.q_net, self.target_net, next_states, rewards, dones, self.gamma)
td_errors = (q_values - targets).abs().detach().cpu().numpy()

loss = (weights * (q_values - targets) ** 2).mean()
self.optimizer.zero_grad()
loss.backward()

torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), max_norm=10.0)
self.optimizer.step()
```

計算當前的 q value 跟 target 的值，算兩個數之間的絕對值當作我的 Bellman error。

How do you modify DQN to Double DQN?

```
def compute_double_dqn_targets(q_net, target_net, next_states, rewards, dones, gamma):
    with torch.no_grad():
        next_q_online = q_net(next_states)
        next_q_target = target_net(next_states)
        best_actions = next_q_online.argmax(dim=1)
        selected_q = next_q_target.gather(1, best_actions.unsqueeze(1)).squeeze(1)
        targets = rewards + gamma * selected_q * (1 - dones)
    return targets
```

使用 q_online 來選擇動作，使用 target_net 來評估所選動作的值，在 target 的計算中使用這個經過雙重評估的 Q 值。

How do you implement the memory buffer for PER?

```
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta=0.4, n_step=3, gamma=0.99):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.n_step = n_step
        self.gamma = gamma
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.pos = 0
        self.n_step_buffer = deque(maxlen=n_step)
```

```
def sample(self, batch_size):
    if len(self.buffer) == self.capacity:
        prios = self.priorities
    else:
        prios = self.priorities[:self.pos]
    probs = prios / prios.sum()
    indices = np.random.choice(len(self.buffer), batch_size, p=probs)
    samples = [self.buffer[i] for i in indices]
    total = len(self.buffer)
    weights = (total * probs[indices]) ** (-self.beta)
    weights /= weights.max()
    states, actions, rewards, next_states, dones = zip(*samples)
    return states, actions, rewards, next_states, dones, indices, torch.tensor(weights, dtype=torch.float32)
```

```python
def update_priorities(self, indices, errors):
    for idx, error in zip(indices, errors):
        self.priorities[idx] = (error + 1e-5) ** self.alpha
```

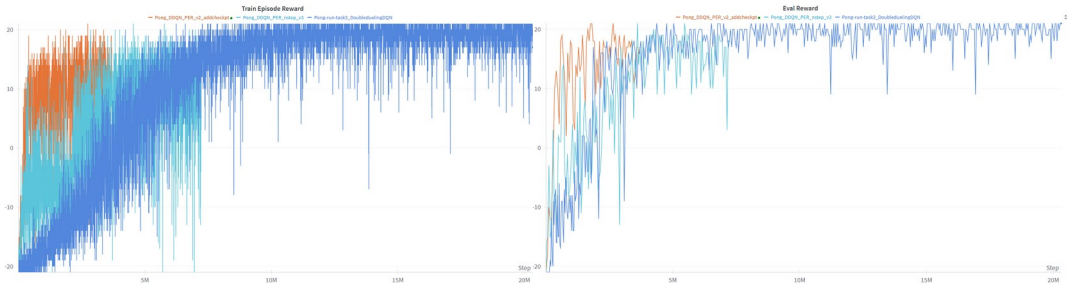
使用 TD error 來更新優先級，sample 的時候會依照優先級來選擇資料。

How do you modify the 1-step return to multi-step return?

```python
def add(self, transition, error):
    self.n_step_buffer.append(transition)
    if len(self.n_step_buffer) < self.n_step:
        return

    state, action = self.n_step_buffer[0].state, self.n_step_buffer[0].action
    reward, next_state, done = 0.0, self.n_step_buffer[-1].next_state, self.n_step_buffer[-1].done

    for idx, transition in enumerate(self.n_step_buffer):
        reward += (self.gamma ** idx) * transition.reward
        if transition.done:
            break

    n_step_transition = Transition(state, action, reward, next_state, done)
    priority = (error + 1e-5) ** self.alpha
    if len(self.buffer) < self.capacity:
        self.buffer.append(n_step_transition)
    else:
        self.buffer[self.pos] = n_step_transition
    self.priorities[self.pos] = priority
    self.pos = (self.pos + 1) % self.capacity
```

累積 n step 內的折扣獎勵，並創建 n step 的轉換，如果中間轉換有 done，則提前終止，最後把這個 n step 的轉換存回 buffer。

Explain how you use Weight & Bias to track the model performance.

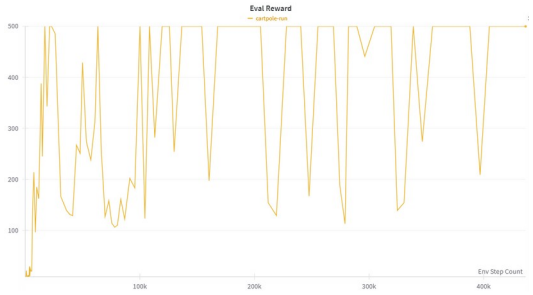


我會看 Eval reward/env step 和 train episode reward /env step 來看我的模型有沒有順利訓練。

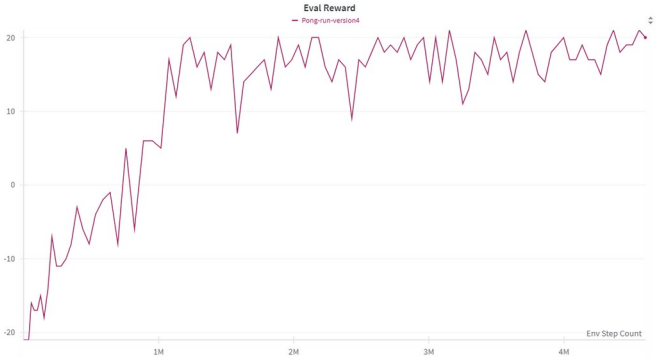
Analysis and discussions (25%):

Plot the training curves (evaluation score versus environment steps) for Task 1, Task 2, and Task 3 separately (10%).

Task1:



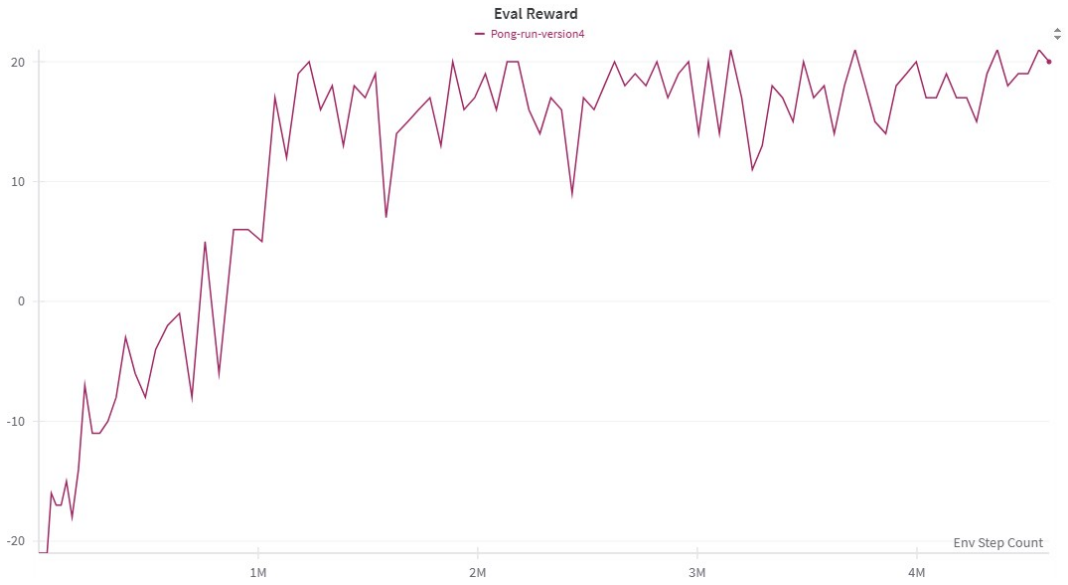
Task2:



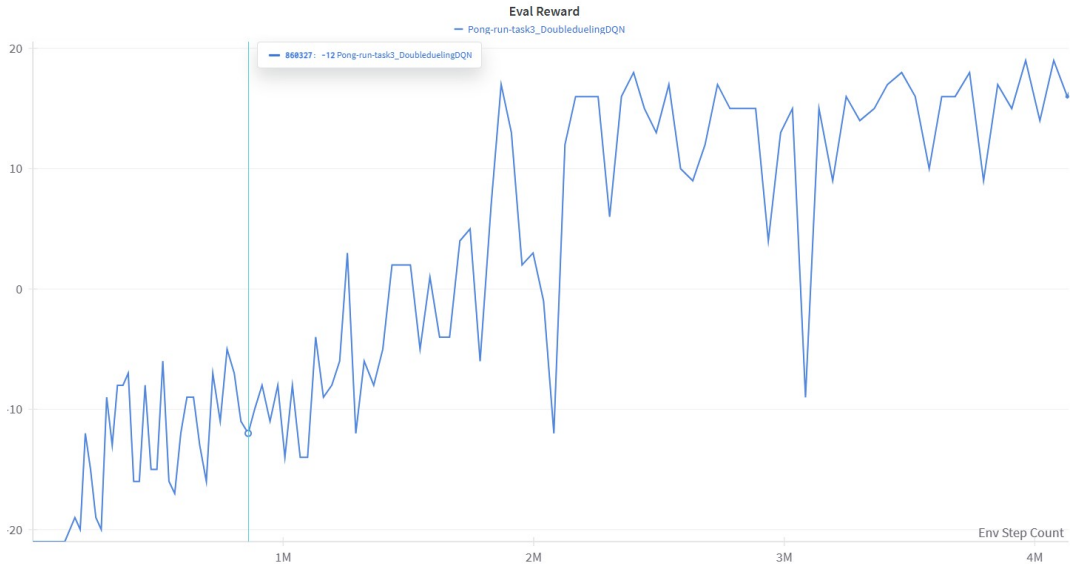
Task3:

Analyze the sample efficiency with and without the DQN enhancements. If possible, perform an ablation study on each technique separately (15%).

DQN:

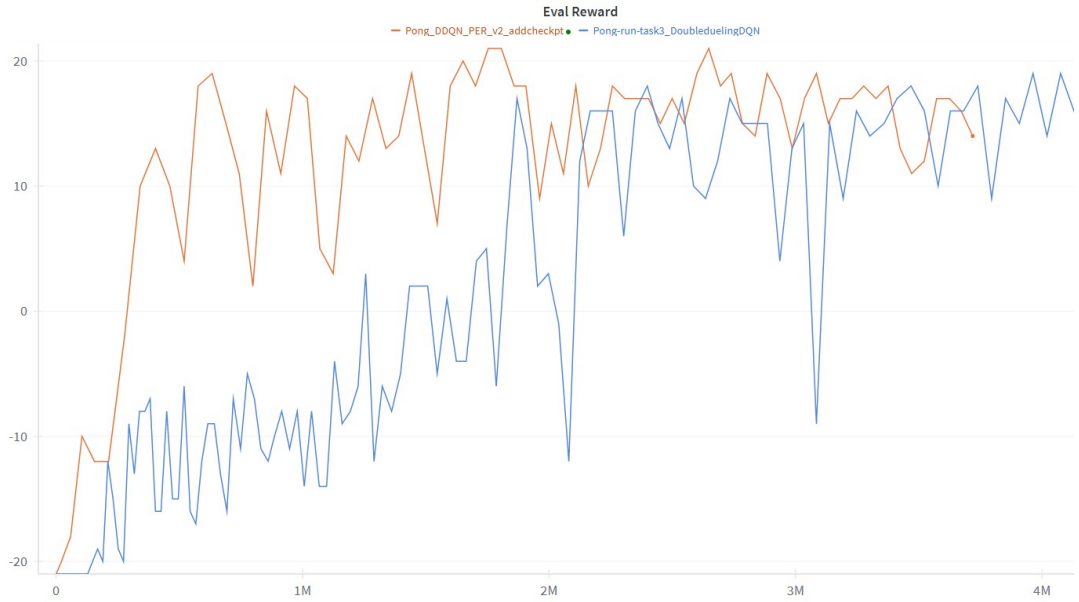


DDQN:



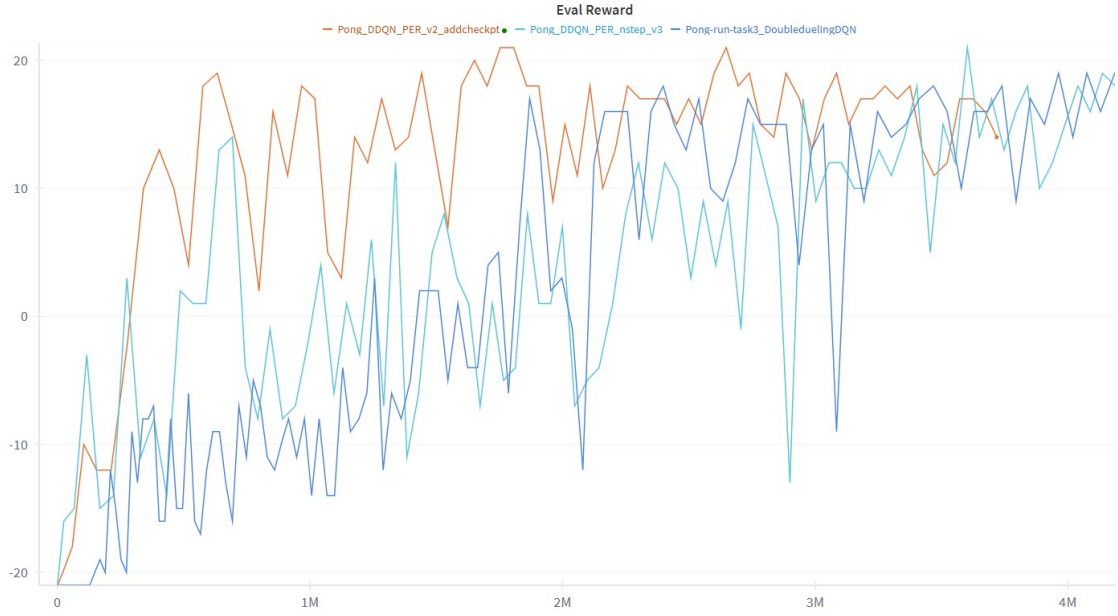
在我的實作中，DQN 的表現並沒有比 DDQN 還要差，相反的結果反而會比較好，依照我的推測可能是因為模型的架構需要依照環境來調整，在 pong 這種簡單的環境裡面，過度的保守估計可能反而會限制了學習的效率。

DDQN+PER:



PER 確實顯著提升了 DDQN 的樣本效率，特別是在訓練早期和中期。這證明了 PER 在稀疏獎勵環境中的價值，能夠有效地找出並重複學習重要的經驗。對於 Pong 這類得分事件稀少的遊戲，PER 是一個非常有效的增強技術。

DQN+PER+Nstep: n step=3



PER 確實顯著提升了樣本效率，但加入 N-step learning 後效果並不理想。這可能是因為 N-step 和 PER 的組合需要更細緻的參數調整。在 Pong 這種簡單的環境中，純 PER 的改進已經足夠有效，N-step 的額外複雜度可能帶來的收益有限。

Additional analysis on other training strategies (Bonus up to 10%)

```
if self.best_reward > 10:
    grad_norm = torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), max_norm=50.0)
else:
    grad_norm = 0.0  # no clipping
```

我發現加上適當的梯度裁剪可以讓模型有更穩定的表現，可能的原因為，遊戲中的獎勵非常稀疏，當模型遇到罕見的得分情況時，Q 值的預測可能會有巨大的誤差，如果沒有做適當的裁剪，那麼大的誤差會導致極大的梯度更新，這會破壞已經學習的參數。

另外，我在訓練時的 loss function 是選擇使用 smoothL1Loss 而不是 MSE，選擇的理由也是因為要避免在更新參數時會有這種大的梯度更新而破壞已經學習的參數的情況。

在我繳交的 zip file 裡面，code 裡面 dqn.py 是 task1 的 code，我也會附上 evaluate checkpoint 的.py 檔案，檔名都會標示是要 evaluate 哪個 task，至於 task2 和 task3，我會附上各自的.py 檔案，且檔名會清楚標注。