ORGANIZATION

1. Introduction
2. Implementation
   A. Bellman error for DQN
   B. Modify DQN to Double DQN
   C. Implement the memory buffer for PER
   D. Modify the 1-step return to multi-step return
   E. Weight & Bias
3. Analysis and discussions
   A. Training curves
      a. Task 1
      b. Task 2
      c. Task 3
   B. efficiency with enhancements
4. Additional analysis and training strategies

1. Introduction (to report)

本實驗實作 DQN，先以狀態較為簡單的 CartPole 環境作為測試，並透過 Bellman Error 來更新模型的梯度。接著將相同的架構應用在 Atari 遊戲 Pong 上，並進一步加入 Prioritized Experience Replay 以及 Multi-step Learning 機制，以更有效率地挑選訓練樣本，並透過一次傳遞多步驟的 reward 來加速模型的收斂和效能提升。

2. Implementation

A. Bellman error for DQN

```
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
```

```
with torch.no_grad():
    target_q_values = rewards + self.gamma * self.target_net(next_states).max(1)[0] * (1 - dones)
```

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

Bellman error 的計算公式是希望 q_values 輸出的值盡可能的靠近 target_q_values，而 target_q_values 是由目前狀態與環境互動所得到的 reward 加上 gamma 乘以下一個狀態執行 action 所能得到的最大 Q 值所得來的。然後將 q_values 和 target_q_values 計算 MSE 就可以得到這個 batch size 的 Bellman error。

B. Modify DQN to Double DQN

```python
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
with torch.no_grad():
    best_action = self.q_net(next_states).argmax(dim=1)
    target_q_values = rewards + (self.gamma ** self.multi_step) * self.target_net(next_states).gather(1, best_action.unsqueeze(1)).squeeze(1) * (1 - dones)
weights = torch.tensor(weights, dtype=torch.float32).to(self.device)
loss = nn.MSELoss(reduction='none')(q_values, target_q_values)
self.memory.update_priorities(indices, loss.detach().cpu().numpy())
loss = (weights * loss).mean()
```

與 DQN 不同的是，target_q_values 中不再採用 target_net 網路對於 next_states 的最大 Q 值，而是將其更改為 next_states 經由 q_net 網路所決定 Q 值的最大動作，交給 target_net 去得出的 Q 值。由此更改就能夠讓網路不會高估 Q 值造成成效不好。

C. Implement the memory buffer for PER

```python
class PrioritizedReplayBuffer:
    """
        Prioritizing the samples in the replay memory by the Bellman error
        See the paper (Schaul et al., 2016) at https://arxiv.org/abs/1511.05952
    """
    def __init__(self, capacity, alpha=0.6, beta=0.4):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.pos = 0

    def add(self, transition, error):
        ########## YOUR CODE HERE (for Task 3) ##########
        if len(self.buffer) < self.capacity:
            self.buffer.append(transition)
            pos = len(self.buffer) - 1
        else:
            pos = np.argmin(self.priorities)
            self.buffer[pos] = transition
        priority = (abs(error) + 1e-5) ** self.alpha
        self.priorities[pos] = priority
        ########## END OF YOUR CODE (for Task 3) ##########
        return
    def sample(self, batch_size):
        ########## YOUR CODE HERE (for Task 3) ##########
        if len(self.buffer) == 0:
            return None
        priority = self.priorities[:len(self.buffer)]
        prob = priority / priority.sum()
        indices = np.random.choice(len(self.buffer), batch_size, p=prob)
        samples = [self.buffer[idx] for idx in indices]
        weights = (len(self.buffer) * prob[indices]) ** (-self.beta)
        weights /= weights.max()
        self.beta = min(1.0, self.beta + 1e-4)
        return samples, indices, weights
        ########## END OF YOUR CODE (for Task 3) ##########
        return
    def update_priorities(self, indices, errors):
        ########## YOUR CODE HERE (for Task 3) ##########
        for ind, error in zip(indices, errors):
            self.priorities[ind] = (abs(error) + 1e-5) ** self.alpha
        ########## END OF YOUR CODE (for Task 3) ##########
        return

    def __len__(self):
        return len(self.buffer)
```

a. Add

在新增當中，如果資料大小尚未達到 capacity，就會直接加進去，而如果大小已滿，則會將最小 error 的那筆資料做替換(和一般的 PER 實作不同)。

b. Sample

根據 priority(已做 alpha 次方)的比例挑選資料，並計算其 weights 作為計算 loss 時候所需要的資訊。

c. Update

將 indices 的資料中更新 error，為了避免 error 為 0 導致 sample 不到該筆資料，因此會加上一個 eps 確保該資料能夠有小機率被選取到。

## D. Modify the 1-step return to multi-step return

```
next_state = self.preprocessor.step(next_obs)
states.append(next_state)
actions_history.append(action)
rewards_history.append(reward)
if len(rewards_history) >= self.multi_step:
    pre_state = states[-self.multi_step-1]
    pre_action = actions_history[-self.multi_step]
    get_reward = sum(self.gamma ** k * rewards_history[-self.multi_step + k] for k in range(self.multi_step))
    now_state = states[-1]
    self.memory.add((pre_state, pre_action, get_reward, now_state, done), np.max(self.memory.priorities))
```

```
with torch.no_grad():
    best_action = self.q_net(next_states).argmax(dim=1)
    target_q_values = rewards + (self.gamma ** self.multi_step) * self.target_net(next_states).gather(1, best_action.unsqueeze(1)).squeeze(1) * (1 - dones)
weights = torch.tensor(weights, dtype=torch.float32).to(self.device)
```

在 n-step 當中，不同於 1-step，實作上會儲存過去的狀態、動作和 reward，再將前 n 步的所有 discount reward 加總，作為(s,a,r,s)的資料，計算

loss 的時候，就會同時考慮第 1~n 的 reward 以及 n+1 的估計 Q 值。

E. Weight & Bias

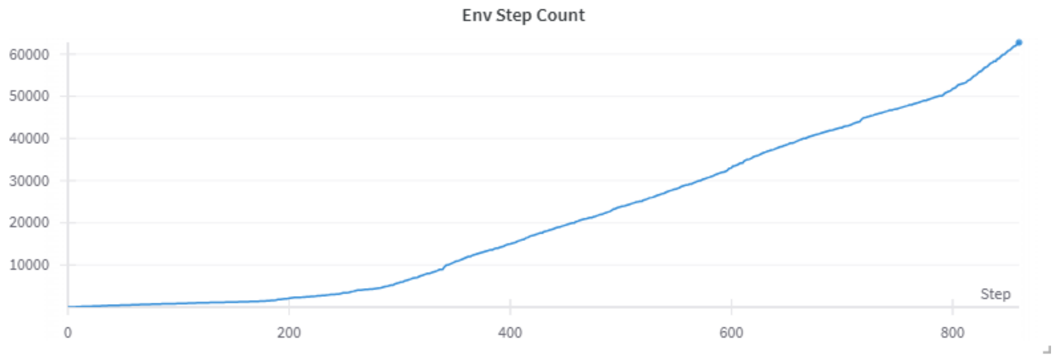
```
wandb.log({
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Eval Reward": eval_reward,
    "Average Reward": np.mean(rewards[-20:]),
    "Best Reward": self.best_reward
})
```

在使用 wandb 當中，除了範例 code 所記錄的各項數值，本次實作還額外新增的 Average Reward 和 Best Reward 資料，這些是在 evaluation 階段紀錄的，根據 evaluate 的結果紀錄 rewards，並且將最新的 20 場平均 reward 顯示出來，以符合本次 lab 的評估標準。

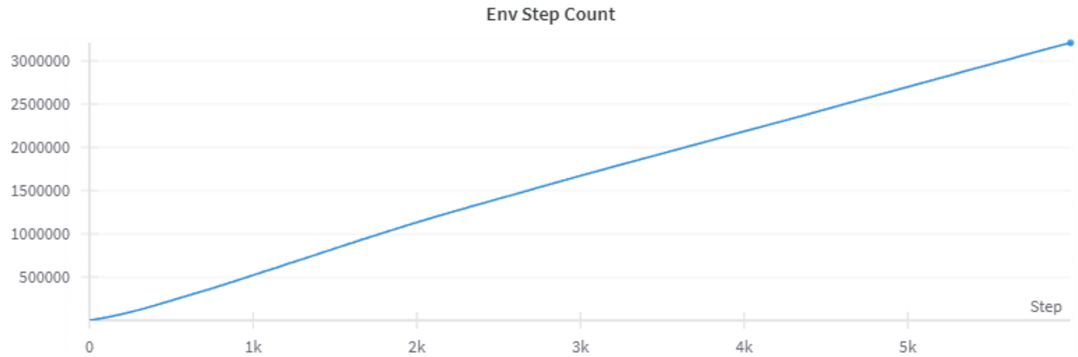
3. Analysis and discussions

A. Training curves

a. Task 1

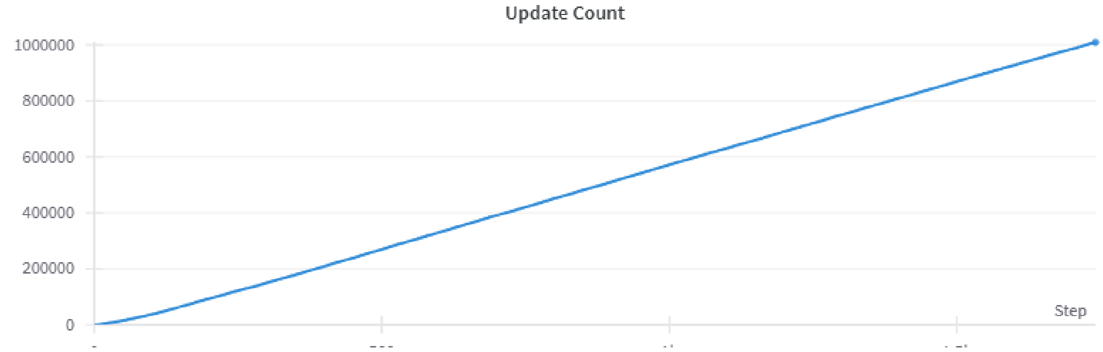



b. Task 2





c. Task 3





B. efficiency with enhancements

從 Task2 到 Task3 之間的比較看出使用 PER 的技巧對於模型訓練速度有顯著的提升。對於 multi-step 而言，本次 Task3 使用的是 multi-step=2 做訓練，然而 multi-step 設置太大反而可能導致模型性能下降。

Multi-step = 7


Multi-step = 11


## C. Additional analysis and training strategies

### a. Regularization

```python
class AtariPreprocessor:
    """
        Preprocesing the state input of DQN for Atari
    """
    def __init__(self, frame_stack=4):
        self.frame_stack = frame_stack
        self.frames = deque(maxlen=frame_stack)

    def preprocess(self, obs):
        gray = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
        resized = cv2.resize(gray, (84, 84), interpolation=cv2.INTER_AREA)
        return resized / 255
        # return resized

    def reset(self, obs):
        frame = self.preprocess(obs)
        self.frames = deque([frame for _ in range(self.frame_stack)], maxlen=self.frame_stack)
        return np.stack(self.frames, axis=0)

    def step(self, obs):
        frame = self.preprocess(obs)
        self.frames.append(frame)
        return np.stack(self.frames, axis=0)
```

將 0~255 的資料所放到 0~1。

### b. smooth L1 loss

其公式如下，smooth L1 loss 在 x = -1~1 之間採用的是 L2，而在-1~1 之外採用的是 L1，這樣的能夠使模型在 training 的時候不會因為 loss 太大而導致梯度爆炸，又能保證在 loss 很小的時候

可以細微的調整梯度，相比 L1 和 L2，smooth L1 loss 算是將他們各自的優點結合，在模型訓練上也較為穩定。

$$\mathrm{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

```
loss = nn.functional.smooth_l1_loss(q_values, target_q_values, reduction='none')
```

4. Execute code

A. Training

python ./ LAB5_313551176_王駿睿_Code/dqn-v1.py

python ./ LAB5_313551176_王駿睿_Code/dqn-v2.py

python ./ LAB5_313551176_王駿睿_Code/dqn-v3.py

B. Evaluate

python ./ LAB5_313551176_王駿睿_Code/ test_model_v1.py

python ./ LAB5_313551176_王駿睿_Code/ test_model_v2.py

python ./ LAB5_313551176_王駿睿_Code/ test_model_v3.py