Lab5: Value-Based Reinforcement Learning

謝鎧駿 B103040021

一、 Introduction

此 Lab 主要實作了 vanilla DQN 在 Cartpole-v1、Pong-v5 上的表現，並去進一步實作多個改良策略，如：DDQN、Prioritized replay buffer、Multi-step return，以比較不同策略對學習穩定與效能的影響。

二、 Implementation

1. Task 1: Vanilla DQN solve cartpole-v1

i. Q network：

使用 3 層 Linear + ReLU 的 Fully connected network，其中有 1024 個 neuron。

```
self.network = nn.Sequential(
    nn.Linear(state_dim,1024),
    nn.ReLU(),
    nn.Linear(1024, 1024),
    nn.ReLU(),
    nn.Linear(1024, num_actions)
)
```

ii. Optimizer and Loss function：

使用 Adam optimizer 和 MSE loss function。

iii. Strategies：

- Epsilon-greedy policy
- Experience replay：replay buffer 直接用 list 存入所有 transitions。
- Two networks：Q network 與 target Q network。

iv. Hyperparameters：

- Batch size : 32
- Learning rate : 0.0005
- Discount factor : 0.99
- Epsilon decay = 0.999、Epsilon min = 0.02
- Target update frequency : 1000
- Replay start size : 500
- Max episodes steps : 10000
- Episodes : 1000

- Train per step : 1

2. Task 2: Vanilla DQN solve Pong-v5

i. Q network：

使用和 Deep mind 論文相同的 Network 架構，3 層 Convolution layers 接 Flatten 並通過兩層 Linear layers。

此三層 Convolution layers 可以快速的 down sampling 並擴大 receptive field，Flatten 並通過兩層 Linear layer 使 Network 可以學習 Q-value。

```
self.network = nn.Sequential(
    nn.Conv2d(input_channels, 32, kernel_size=8, stride=4),
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=4, stride=2),
    nn.ReLU(),
    nn.Conv2d(64, 64, kernel_size=3, stride=1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * 7 * 7, 512),
    nn.ReLU(),
    nn.Linear(512, num_actions)
)
```

ii. Optimizer and Loss function：

使用 Adam optimizer 和 MSE loss function。

iii. Strategies：

- Epsilon-greedy policy
- Experience replay
- Two networks：Q network 與 target Q network。

iv. Hyperparameters：

- Batch size : 32
- Learning rate : 0.0001
- Discount factor : 0.99
- Epsilon decay = 0.999999、Epsilon min = 0.05
- Target update frequency : 1000
- Memory size : 100000
- Replay start size : 50000
- Max episodes steps : 10000
- Episodes : 10000
- Train per step : 1

3. Task 3:Enhanced DQN solve Pong-v5

這邊只說明與 Task2 不同的地方。

i. Enhanced DQN：DDQN、Priority replay buffer、Multi-step return 此三個技巧說明於第 5,6,7 點。

ii. Linear epsilon decay：

原本 Task2 是每次乘上一個 epsilon decay，但這樣在 Task3 比較難控制什麼時候遞減到 minimum，所以改成 Linear epsilon decay，讓我可以控制用多少 evn steps 去遞減到 minimum。

```
decay_progress = min(1.0, self.env_count / self.epsilon_decay_steps)
self.epsilon = self.epsilon_start - decay_progress * (self.epsilon_start - self.epsilon_min)
self.train_count += 1
```

iii. Reward clipping

原本 Task2 是直接用環境給的 reward，但這樣 reward 範圍從 -21～+21，有時候會使 reward 變動太大，導致計算出來的 TD target 不太穩定，因此我將 reward clip 到-1～+1。

```
reward = np.clip(reward, -1, 1)
```

iv. RMSProp

因為在 pong 這個環境中，初期的 reward 通常都是 0，因為機器還沒學會擊球，這時用上 Adam 會比較容易 gradient explode，而 RMSProp 相對 Adam 來說更穩定，能幫助機器在前期探索的穩定性。

```
self.optimizer = optim.RMSprop(self.q_net.parameters(), lr=args.lr, alpha=0.95, eps=1e-2)
```

v. Hyperparameters：

- Batch size : 32
- Learning rate : 0.00025
- Discount factor : 0.99
- Epsilon decay steps = 250000
- Target update frequency : 5000
- Memory size : 500000
- Replay start size : 30000
- Max episodes steps : 10000
- Episodes : 10000
- Train per step : 3
- N steps : 4

4. How do you obtain the Bellman error for DQN

我是直接按照講義的定義跟助教作業說明的公式去做計算(如下圖)，

$$\left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \bar{\theta}) - Q(s, a; \theta)\right)$$

對照我的程式步驟是：

i. 計算 Q(s,a)：

把從 replay buffer sample 出來的 states 丟到 Q network 中，並把對應的 actions 的 Q values 取出。

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

ii. 計算 target：

把那些 sample 出來的 data 的 next state 丟到 target Q network 中，取出每個輸出的最大 Q value，並乘上 gamma 值，最後加上當前此 action 得到的 reward，即為 target value 了，最後乘上 1-dones 是因為當遊戲結束就不計算了。

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

iii. 計算 Bellman error：

用自己定義的 criterion 去計算 Q(s,a)與 target 值得 loss。

```
loss = criterion(q_values, q_targets)
```

5. How do you modify DQN to Double DQN?

我將選 action 與計算 Q value 兩個動作拆給 Q network 跟 target Q network。

```
next_actions = self.q_net(next_states).argmax(1, keepdim=True)
q_nexts      = self.target_net(next_states).gather(1, next_actions).squeeze(1)
q_targets    = rewards + self.gamma * q_nexts * (1 - dones)
```

6. How do you implement the memory buffer for PER?

i. Add

將 transition 與其對應的 priority 儲存起來。

```
def add(self, transition, error):
    ########## YOUR CODE HERE (for Task 3) ##########
    priority = (abs(error) + 1e-6) ** self.alpha

    if len(self.buffer) < self.capacity:
        self.buffer.append(transition)
    else:
        self.buffer[self.pos] = transition

    self.priorities[self.pos] = priority
    self.pos = (self.pos + 1) % self.capacity
    ########## END OF YOUR CODE (for Task 3) ##########
    return
```

ii. Sample

首先將 priority 正規化成機率分布，接著根據計算出來的機率分布去抽出資料的 index，最後計算 IS weight，並把 IS weight 正規化，以便之後修正偏差值。

```
def sample(self, batch_size):
    ########## YOUR CODE HERE (for Task 3) ##########
    if len(self.buffer) == self.capacity:
        prios = self.priorities
    else:
        prios = self.priorities[: self.pos] # valid part only

    probs = prios / prios.sum()
    indices = np.random.choice(len(probs), batch_size, p=probs)

    # Importance-sampling weights
    N = len(self.buffer)
    weights = (N * probs[indices]) ** (-self.beta)
    weights /= weights.max()                    # normalize to 1

    transitions = [self.buffer[idx] for idx in indices]
    ########## END OF YOUR CODE (for Task 3) ##########
    return transitions, indices, torch.tensor(weights, dtype=torch.float32)
```

iii. Update priorities

更新 replay buffer 中指定的 transition 的 priority，以便下次能更有效率的 sample。

```
def update_priorities(self, indices, errors):
    ########## YOUR CODE HERE (for Task 3) ##########
    for idx, err in zip(indices, errors):
        self.priorities[idx] = (abs(err) + 1e-6) ** self.alpha
    ########## END OF YOUR CODE (for Task 3) ##########
    return
```

iv. 當一筆新資料放入 Prioritized Replay Buffer 時，給它高 priority（最大的現有 priority 或預設 1.0），以便在早期被抽中訓練，當他被抽中後，就會被 update priority。

```
if len(self.memory) > 0:
    valid_prios = self.memory.priorities[: len(self.memory)]
    max_prio    = valid_prios.max()
else:
    max_prio = 1.0
self.memory.add((state, action, reward, next_state, done), error=max_prio)
```

v. 最後就是計算 Loss function 的時候要乘上 weight，避免失真，然後 update 完參數後要 update sample 到的那些 transitions 的 priorities。

```
td_errors = q_targets - q_values
loss = (is_w * td_errors.pow(2)).mean()   # MSE

self.optimizer.zero_grad()
loss.backward()
nn.utils.clip_grad_norm_(self.q_net.parameters(), 5)
self.optimizer.step()

self.memory.update_priorities(indices, td_errors.detach().cpu().numpy())
```

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

原本是直接將 transition append 到 memory 裡面，使用 multi-step return 後，先將 transition append 到一個 buffer 裡面，當 buffer 收滿 n 個 transition 的時候，就可以開始反向迭代去計算 R，計算完就 append 到 memory 裡面。

```
self.nstep_buffer.append( (state, action, reward, next_state, done) )

# Multi-step return
if len(self.nstep_buffer) == self.n_steps:
    R, s_n, done_n = 0.0, next_state, done
    for i, (_,_,r, s_i, d_i) in enumerate(reversed(self.nstep_buffer)):
        R = r + (self.gamma * R * (1 - d_i))
        if d_i:
            s_n, done_n = s_i, True
            break
    s_0, a_0, *_ = self.nstep_buffer[0]
    self.memory.append( (s_0, a_0, R, s_n, done_n) )
```

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

我主要利用 TA 寫好的資訊去 track model performance，像是追蹤 epsilon 讓我知道用 linear epsilon decay 更能控制 decay 的速度與時間；而 Eval reward 主要用來評估 model 的好壞，也能知道花多少 evn steps 才能穩定收斂在高分；最後，透過 Total reward and Eval reward，讓我很好的去評估一組超參數的好壞，讓我更快地找出適合該 model 的超參數。

三、 Analysis and Discussions

1. Training curves：

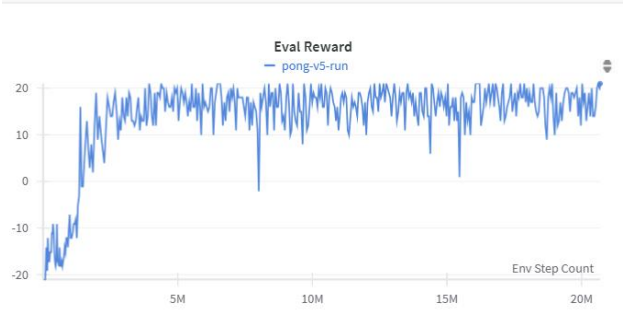
- Task1:

  最初 reward 緩慢上升，中期出現明顯波動，花一段時間緩慢學習探索後，約在 80k 時上升接近 500，之後收斂穩定在 500。

- Task2:

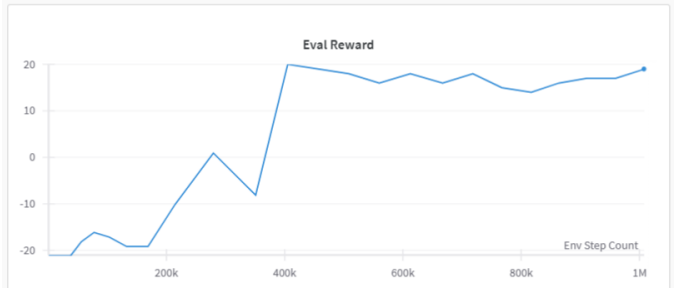
  初期(約 0-1M evn steps)agent 的分數都偏低，約在 1.4M evn steps 的時候，reward 明顯提升，推測是開始學會擊球，接著 3M-20M evn steps 都維持在 15-21 分，但還是有些許不穩定。此結果花了

約 3M env steps 才收斂分數，因此推薦可以優先加入 Multi-step Return，加快收斂速度。



- **Task3:**

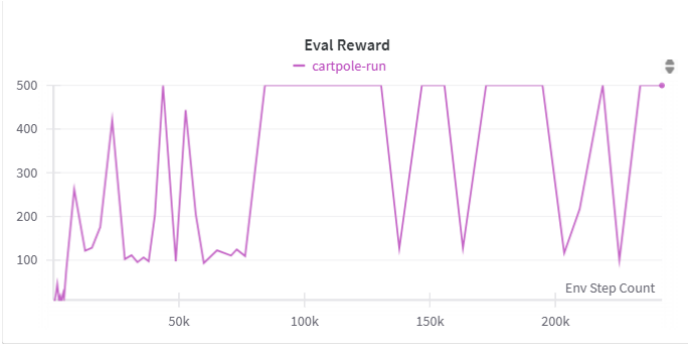
  我們可以發現在 400k evn steps 的時候就已經收斂了，與 Task2 相比，真的快了超級多，所以推測用上 Enhanced DQN，會增加學習力並加快收斂的速度。



2. **Analyze the sample efficiency with and without the DQN enhancements:**

   - Task1 with DDQN:

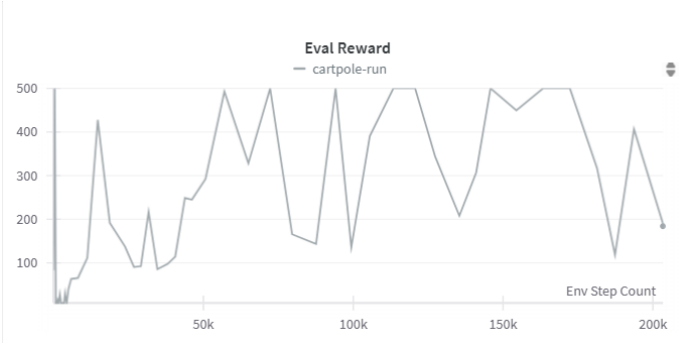
     與原本的 task1 相比，因為 DDQN 抑制了過高 Q-value 的偏差，初期幫助大，使 reward 較早上升，且較早碰到最高分，但中期和後期有些許震盪，雖然能到達 500 分，但不太穩定。



   - Task1 with priority replay buffer:

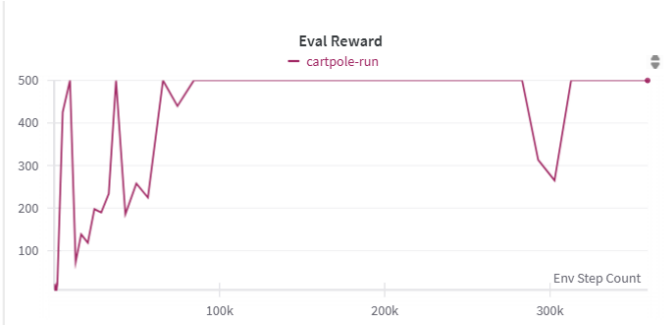
     雖然相較 DDQN 可以更早到達 500 分，推測是因為優先取樣使

agent 可以快速的學習到關鍵的 transition，但是可能過度依賴在高 TD error 的資料，反而無法穩定 500 分。



- Task1 with multi-step return:

  可以觀察到 multi-step return 的學習效率跟收斂速度是最好的，果然多看未來幾步的結果，使 agent 更有遠見，既能使 agent 早點學習到關鍵，又能穩定在高分。



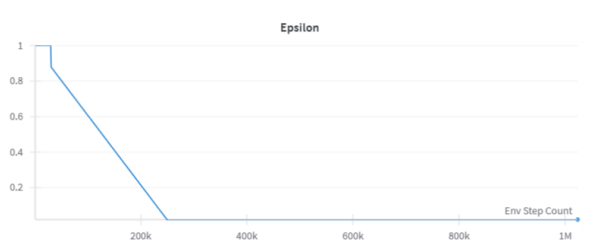
- Comparison between task2 & task3:

  我們可以在第 1 點看到 task2 跟 task3 的 Eval reward 訓練圖，在 task2 的 vanilla DQN，花了大約 3M 的 evn steps 才收斂分數；然而在 task3，用上了 Enhanced DQN 後，只花了 400k 就成功收斂分數了，增加了 agent 很多的學習力與收斂速度。

四、 Additional analysis on other training strategies

i. Linear epsilon decay

控制 exploration 在 evn steps 250k 的時候遞減到 epsilon minimum，避免太快或太久收斂。



ii. Reward clipping

控制 TD target 的變動，使 Q learning 更穩定。

iii. RMSProp

相較 Adam 更適合用在 Pong 環境中，較為穩定。

iv. Gradient clipping

在 cartpole 跟 Pong 環境中，有時會出現 gradient explode 的現象，因此使用 Gradient clipping 來限制 gradient。