

# DL\_Lab5\_Report

## A.Introduction (5%)

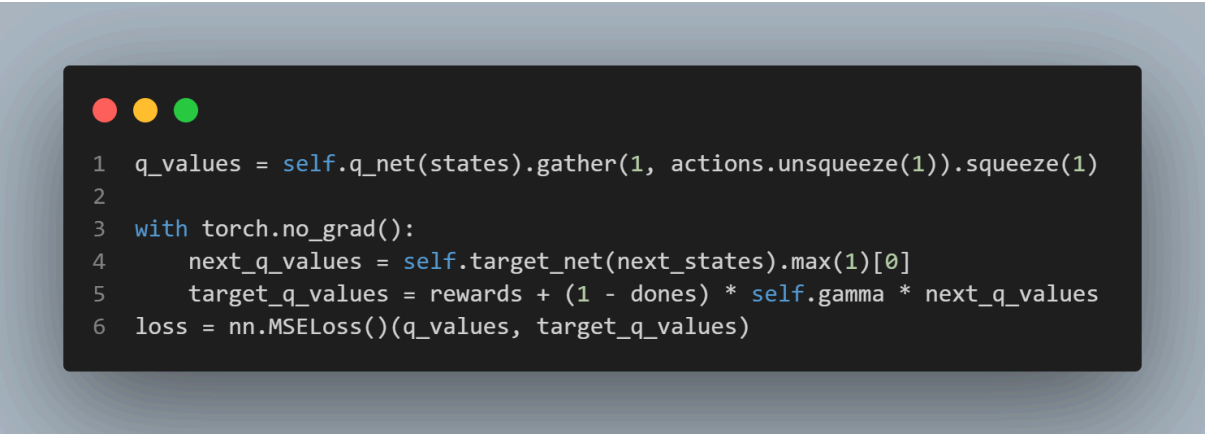
This report presents our implementation of Deep Q-Network (DQN) and its variants applied to the Pong&cart pole game environment. I learned how to implement standard DQN, Double DQN, prioritized experience replay (PER), and multi-step returns, while tracking model performance using the Weights & Biases (wandb) platform. The report is organized into sections that cover our code implementations for Tasks 1–3.

---

## B.Your Implementation (20%)

### 1. Bellman Error for DQN

I computed the Bellman error using the temporal difference (TD) loss between predicted Q-values and target Q-values. This is the vanilla DQN update using MSE loss, where `target_net` is a periodically updated copy of `q_net`.



```
1 q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
2
3 with torch.no_grad():
4     next_q_values = self.target_net(next_states).max(1)[0]
5     target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
6 loss = nn.MSELoss()(q_values, target_q_values)
```

### 2.modify DQN to Double DQN

To implement DDQN, Double DQN decouples action selection and evaluation by using the online network to choose the best action and the target network to evaluate its value:

```

1 elif args.task == 3: # DDQN
2     # 1) 用線上網路選出最優動作
3     next_actions = self.q_net(next_states).argmax(dim=1)
4     # 2) 用 target 網路評估該動作的價值
5     next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
6     target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
7

```

### 3.How do you implement the memory buffer for PER

The PrioritizedReplayBuffer class .It use self.buffer to store transitions in the format (state, action, reward, next\_state, done).Besides the buffer , I also used a parallel array self.priorities to hold the priority value for each transition. While self.pos tracks the current insertion index, supporting a circular buffer when capacity is full.

```

1 class PrioritizedReplayBuffer:
2     """
3         Prioritizing the samples in the replay memory by the Bellman error
4         See the paper (Schaul et al., 2016) at https://arxiv.org/abs/1511.05952
5     """
6     def __init__(self, capacity, alpha=0.6, beta=0.4):
7         self.capacity = capacity
8         self.alpha = alpha
9         self.beta = beta
10        self.buffer = []
11        self.priorities = np.zeros((capacity,), dtype=np.float32)
12        self.pos = 0
13        self.device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")

```

Below are the main three functions of the class .

#### Adding Transitions (add method):

Each time a transition is added, compute its TD error and assign a priority value as:  $\text{priority} = |TD\ error|^\alpha$ , where  $\alpha$  controls how much prioritization is used . I didn't change the default  $\alpha = 0.6$ .And If the buffer is full, the oldest transition is overwritten.

#### Sampling Mechanism (sample method):

Sampling is done according to a probability distribution defined by:

$$P(i) = \text{priority}_i^\alpha / \sum \text{priority}_j^\alpha$$

Transitions are drawn with probabilities proportional to their priorities.

Importance-sampling (IS) weights are computed to correct for bias:

$w_i = (1 / N * 1 / P(i))^\beta$  normalized by the maximum weight to avoid instability.

### Updating Priorities (update\_priorities method):

After training on a batch, it will update the priorities of sampled transitions using their latest TD errors:  $new\_priority = |TD\ error|^\alpha$ .

```
1 def add(self, transition, error):
2     ##### YOUR CODE HERE (for Task 3) #####
3     # 插入新經驗與其 priority
4     p = (abs(error) + 1e-6) ** self.alpha
5     if len(self.buffer) < self.capacity:
6         self.buffer.append(transition)
7     else:
8         self.buffer[self.pos] = transition
9         self.priorities[self.pos] = p
10        self.pos = (self.pos + 1) % self.capacity
11        ##### END OF YOUR CODE (for Task 3) #####
12    return
13 def sample(self, batch_size):
14     ##### YOUR CODE HERE (for Task 3) #####
15     # 根據 priority 抽樣
16     scaled_p = self.priorities[:len(self.buffer)] ** self.alpha
17     probs = scaled_p / scaled_p.sum()
18     indices = np.random.choice(len(self.buffer), batch_size, p=probs)
19     samples = [self.buffer[i] for i in indices]
20     # 計算重要性取樣權重
21     N = len(self.buffer)
22     weights = (N * probs[indices]) ** (-self.beta)
23     weights = weights / weights.max() # normalize
24     return indices, samples, torch.tensor(weights, dtype=torch.float32, device=self.device)
25
26     ##### END OF YOUR CODE (for Task 3) #####
27    return
28 def update_priorities(self, indices, errors):
29     ##### YOUR CODE HERE (for Task 3) #####
30     # 更新被抽樣 transitions 的 priority
31     for idx, err in zip(indices, errors.detach().cpu().numpy()):
32         self.priorities[idx] = (abs(err) + 1e-6) ** self.alpha
33     ##### END OF YOUR CODE (for Task 3) #####
34    return
```

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

Below is my code about step return issue . For 1-step return ( else statement)

just simply append to the memory . As for the multi-step return,the buffer's add method holds raw transitions until it has  $n_{\text{step}}$  of them, then collapses them into one  $n$ -step tuple . The resulting training target becomes

$$R_t^{(n)} = r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a Q(s_{t+n}, a).$$

```

1  if self.args.task == 3:
2      # Add multi-step transition to PER buffer
3      transition = (state, action, reward, next_state, done)
4      state_t = torch.from_numpy(np.array(state, dtype=np.float32)).unsqueeze(0).to(self.device)
5      next_state_t = torch.from_numpy(np.array(next_state, dtype=np.float32)).unsqueeze(0).to(self.device)
6      action_t = torch.tensor([action], dtype=torch.int64).to(self.device)
7      reward_t = torch.tensor([reward], dtype=torch.float32).to(self.device)
8      done_t = torch.tensor([done], dtype=torch.float32).to(self.device)
9      with torch.no_grad():
10         q_val = self.q_net(state_t).gather(1, action_t.unsqueeze(1)).squeeze(1)
11         next_q = self.target_net(next_state_t).max(1)[0]
12         target = reward_t + (1 - done_t) * self.gamma * next_q
13         init_error = (target - q_val).item()
14         self.memory.add(transition, init_error)
15     else:
16         # 非 PER 任務就維持原本 append
17         self.memory.append((state, action, reward, next_state, done))
18

```

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

I initialize a wandb run at the start of training. During training and evaluation,I logged metrics such as:

- Episode total reward
- Training loss, Q-value mean and std every 1000 updates
- Epsilon value and step counts
- Evaluation reward

And also updated wandb.config with hyperparameters for reproducibility.

## C. Analysis and discussions (25%)

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

The x axis which is the step should all be multiplied by 20 since I only evaluate once every 20 steps .

Task1 :

x-axis should be 24k

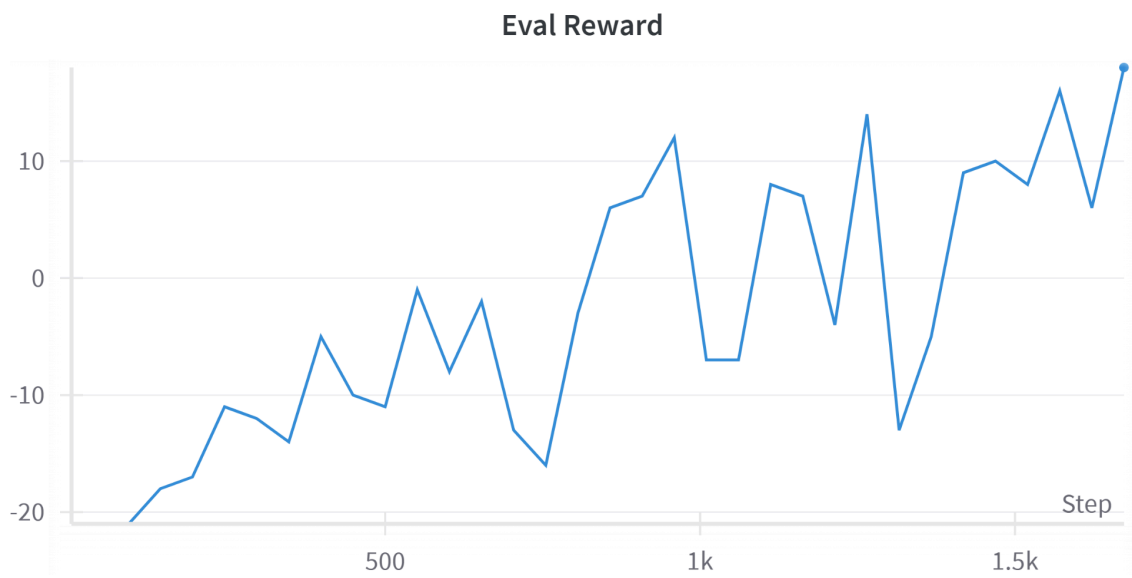
Final Eval Reward : 500



Task 2 :

Final Eval Reward : 18

x-axis should be 30k



Task 3 :

I have tried several hyperparameters , but the result turned out to be quite tragic, below are two of the result.

x-axis should be 30k



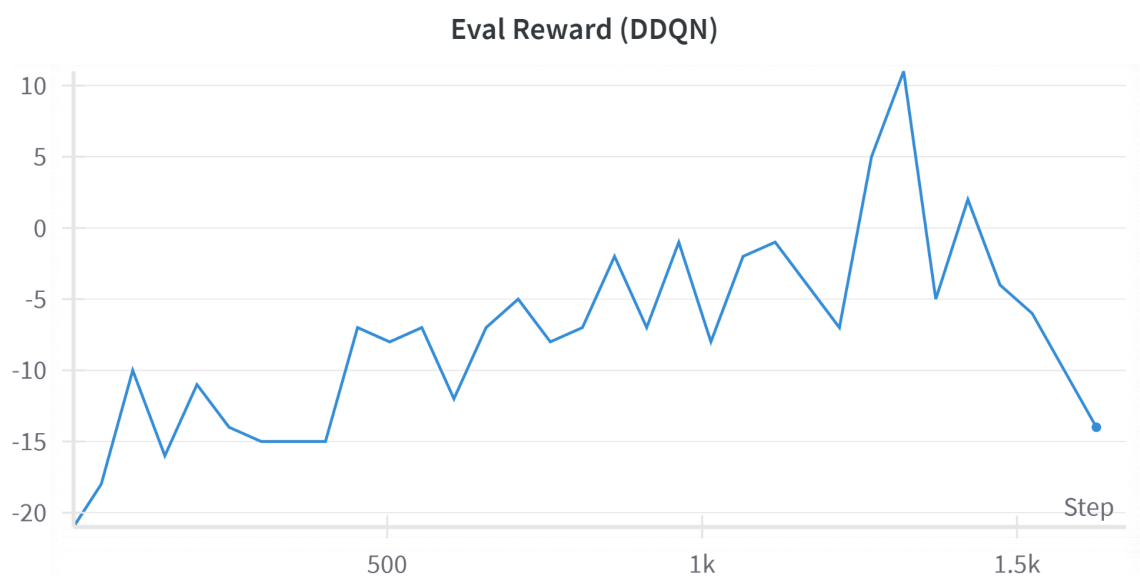
– Analyze the sample efficiency with and without the DQN enhancements. If possible, perform an ablation study on each technique separately (15%).

Below are the result of the ablation study .

- DDQN:

In the original vanilla DQN the max operator is used to both select and evaluate the next action, which often leads to overestimated Q-values.

DDQN solves it by decoupling action selection and evaluation — the action is selected using the online network (q\_net), but its value is estimated by the target network (target\_net).

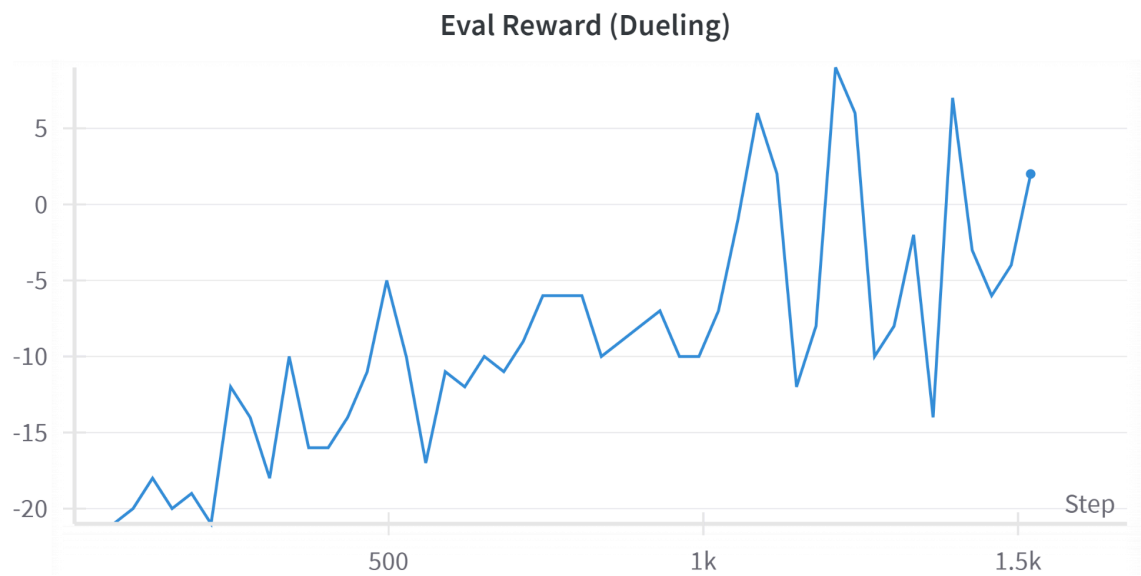


- Dueling Network Architecture:

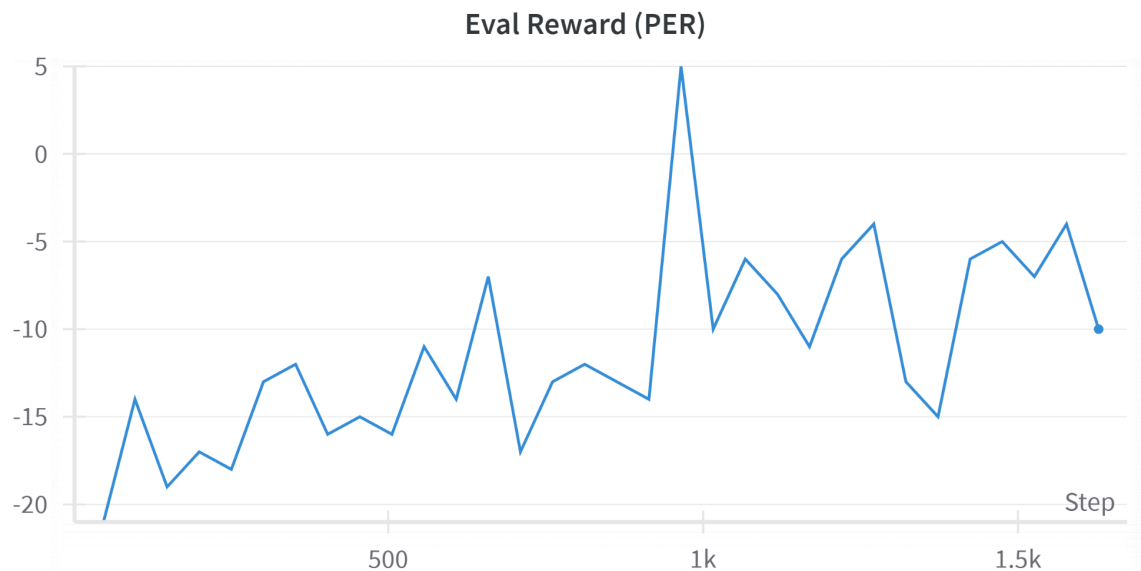
In the original vanilla DQN, the choice of action doesn't matter much, but DQN still spends capacity estimating each action's Q-value. Dueling splits the network into two streams: One estimates the state value  $V(s)$  The other estimates the advantage  $A(s,a)$



then it combines them into the Q-value.



- **PER:**  
Problem with uniform replay: Transitions are sampled equally, even if some are already well-learned or unimportant. And PER It prioritizes transitions with larger TD errors, which represent areas where the model is still uncertain or wrong.



By reducing overestimation (DDQN), improving value estimation (Dueling), and focusing learning on important experiences (PER), these enhancements help the agent make more effective use of each environment interaction, thus improving sample efficiency.

By applying these three techniques together , I reached eval score 18 in the end

