1.introduction:

The theme of this lab is reinforcement learning. We implemented two cases: starting with the simpler CartPole environment to understand the basic operations of DQN and the overall reinforcement learning workflow. We then moved on to the more complex case, Pong, where we enhanced the DQN with a CNN to enable learning directly from visual input. Additional architectural improvements were also applied to further strengthen the DQN.

2. Code implementation:

Q. How do you obtain the Bellman error for DQN?

$$L_{\text{DQN}}(\theta) := \frac{1}{2} \sum_{(s,a,r,s') \in D} \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \bar{\theta}) - Q(s, a; \theta) \right)^2$$

We calculate the Q value from the Q-network and get the expected Q value by calculating the Q value from the target network and get the maximum one and using the bellman equation. The TD error, which we want to minimize, is calculated by subtracting the current Q-value from the expected (target) Q-value.

```
with torch.no_grad():
    next_q_values = self.target_net(next_states).max(1)[0]

target_q_values = rewards + (1 - dones) * self.gamma * next_q_values

loss = F.mse_loss(q_values, target_q_values)
```

Q. How do you modify DQN to Double DQN?

$$L_{\text{DDQN}}(\theta) := \frac{1}{2} \sum_{(s,a,r,s') \sim D} \left(r + \gamma Q\left(s', \arg\max_{a' \in A} Q(s, a; \theta); \bar{\theta}\right) - Q(s, a; \theta) \right)^2$$

Unlike the original method that directly feeds next states into the target network and takes the maximum Q-value as the estimate of future rewards, Double DQN first uses the online network (q_net) to compute and select the action with the highest Q-value for each next state. These selected actions are then passed to the target network (target_net) to retrieve the corresponding Q-values, which are used as part of the TD target. By decoupling action selection from Q-value estimation, this approach effectively mitigates overestimation and improves the

stability and performance of learning.

```
if self.DDQN:
    next_actions = self.q_net(next_states).argmax(dim=1)
    next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
```

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

I implement the functions of the Prioritize Replay Buffer to control the priority of the batch we choose instead of randomly select one:

Add():

We append each new transition (batch) to the buffer. If the buffer is full, we overwrite the oldest entry in a circular manner. For each new transition, we also compute its priority using the TD (Temporal Difference) error and the hyperparameter α (alpha) based on the formula below, which controls how much prioritization is used:

$$\text{Priority: } p_i = |\delta_i| + \epsilon$$

Sample():

At first, we select the indices of the batch to sample. The sampling is based on each transition's priority — the higher the priority, the more likely it is to be selected.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Then, we compute the importance-sampling weights (being used for loss computation) for the selected batches using the given formula and normalize them by dividing by the maximum weight.

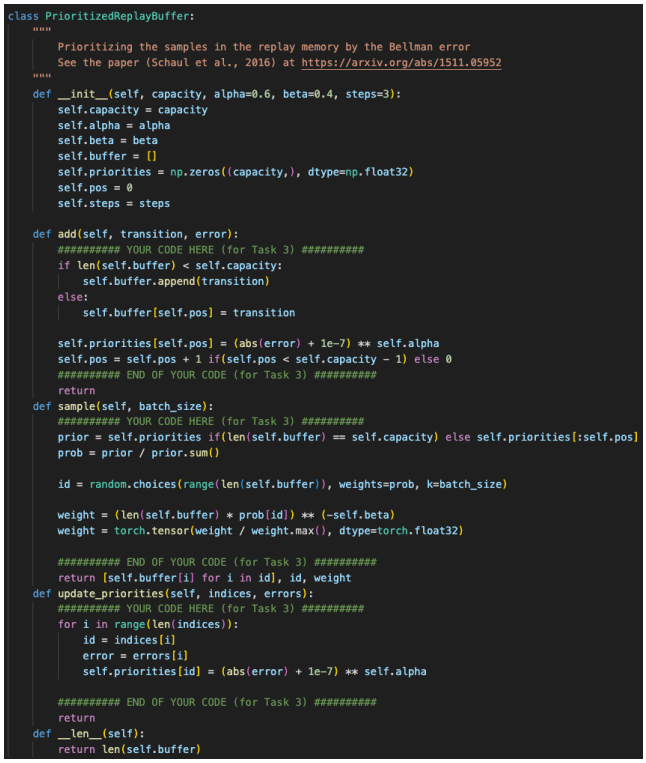
$$w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta$$

Finally, we return the sampled batch, the corresponding weights, and their

indices.

update_priorities():

After training with the selected batch and obtaining the corresponding TD errors, we update the priorities of the specified indices based on their respective TD errors.



```
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, steps=3):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.pos = 0
        self.steps = steps

    def add(self, transition, error):
        ########## YOUR CODE HERE (for Task 3) ##########
        if len(self.buffer) < self.capacity:
            self.buffer.append(transition)
        else:
            self.buffer[self.pos] = transition

        self.priorities[self.pos] = (abs(error) + 1e-7) ** self.alpha
        self.pos = self.pos + 1 if(self.pos < self.capacity - 1) else 0
        ########## END OF YOUR CODE (for Task 3) ##########
        return
    def sample(self, batch_size):
        ########## YOUR CODE HERE (for Task 3) ##########
        prior = self.priorities if(len(self.buffer) == self.capacity) else self.priorities[:self.pos]
        prob = prior / prior.sum()

        id = random.choices(range(len(self.buffer)), weights=prob, k=batch_size)

        weight = (len(self.buffer) * prob[id]) ** (-self.beta)
        weight = torch.tensor(weight / weight.max(), dtype=torch.float32)

        ########## END OF YOUR CODE (for Task 3) ##########
        return [self.buffer[i] for i in id], id, weight
    def update_priorities(self, indices, errors):
        ########## YOUR CODE HERE (for Task 3) ##########
        for i in range(len(indices)):
            id = indices[i]
            error = errors[i]
            self.priorities[id] = (abs(error) + 1e-7) ** self.alpha

        ########## END OF YOUR CODE (for Task 3) ##########
        return
    def __len__(self):
        return len(self.buffer)
```

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

I implemented a MultiStepBuffer to compute the n-step return. Before the buffer reaches the required size (n_step), each new transition is simply appended. Once the buffer is full, we iterate through the stored transitions to compute the prefix of the n-step return by multiplying rewards and gamma , stopping early if a done signal is encountered.

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q(s_{t+n}, a')$$

We then retrieve the initial state and action from the oldest transition, pop it from the buffer to make space for new entries, and if a terminal state (done) was

reached during the accumulation, we clear the entire buffer. Finally, the computed n-step transition is returned for storage in the main replay buffer.

```python
class MultiStepBuffer:
    def __init__(self, n_step, gamma):
        self.n_step = n_step
        self.gamma = gamma
        self.buffer = []

    def add(self, transition):
        self.buffer.append(transition)
        if len(self.buffer) < self.n_step:
            return None

        total_reward, next_state, finish = 0, None, False
        for id, batch in enumerate(self.buffer):
            state, action, reward, next, done = batch
            total_reward += (self.gamma ** id) * reward
            next_state = next
            if done:
                finish = True
                break

        state, action, _, _, _ = self.buffer[0]
        self.buffer.pop(0)

        if finish:
            self.buffer.clear()

        return (state, action, total_reward, next_state, finish)

    def reset(self):
        self.buffer.clear()
```

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

At different stages, we use the wandb.log function to send training information such as step_count and env_count to Weights & Biases. This information is then visualized as charts on the W&B website, allowing us to observe the trends and changes of various metrics over time.

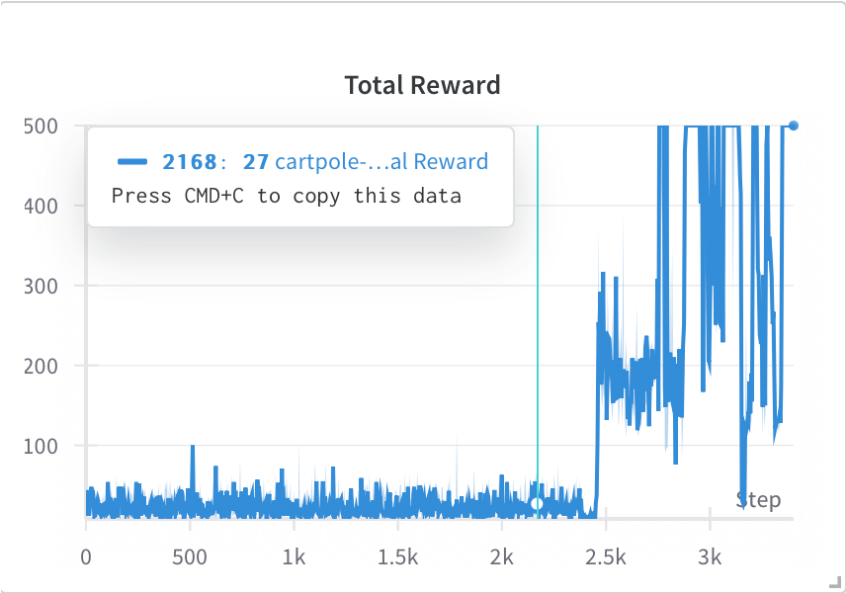
```
    if self.env_count % 1000 == 0:
        print(f"[Collect] Ep: {ep} Step: {step_count} SC: {self.env_count} UC: {self.train_count} Eps: {self.epsilon:.4f}")
        wandb.log({
            "Episode": ep,
            "Step Count": step_count,
            "Env Step Count": self.env_count,
            "Update Count": self.train_count,
            "Epsilon": self.epsilon
        })
        ########## YOUR CODE HERE  ##########
        # Add additional wandb logs for debugging if needed

        ########## END OF YOUR CODE ##########
print(f"[Eval] Ep: {ep} Total Reward: {total_reward} SC: {self.env_count} UC: {self.train_count} Eps: {self.epsilon:.4f}")
wandb.log({
    "Episode": ep,
    "Total Reward": total_reward,
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Epsilon": self.epsilon
})
########## YOUR CODE HERE  ##########
# Add additional wandb logs for debugging if needed

########## END OF YOUR CODE ##########
if ep % 100 == 0:
    model_path = os.path.join(self.save_dir, f"model_ep{ep}.pt")
    torch.save(self.q_net.state_dict(), model_path)
    print(f"Saved model checkpoint to {model_path}")

if ep % 20 == 0:
    eval_reward = self.evaluate()
    if eval_reward >= self.best_reward:
        self.best_reward = eval_reward
        model_path = os.path.join(self.save_dir, "best_model.pt")
        torch.save(self.q_net.state_dict(), model_path)
        print(f"Saved new best model to {model_path} with reward {eval_reward}")
    print(f"[TrueEval] Ep: {ep} Eval Reward: {eval_reward:.2f} SC: {self.env_count} UC: {self.train_count}")
    wandb.log({
        "Env Step Count": self.env_count,
        "Update Count": self.train_count,
        "Eval Reward": eval_reward
    })
```

Task3:

3. Experiments:

1) Screenshot:

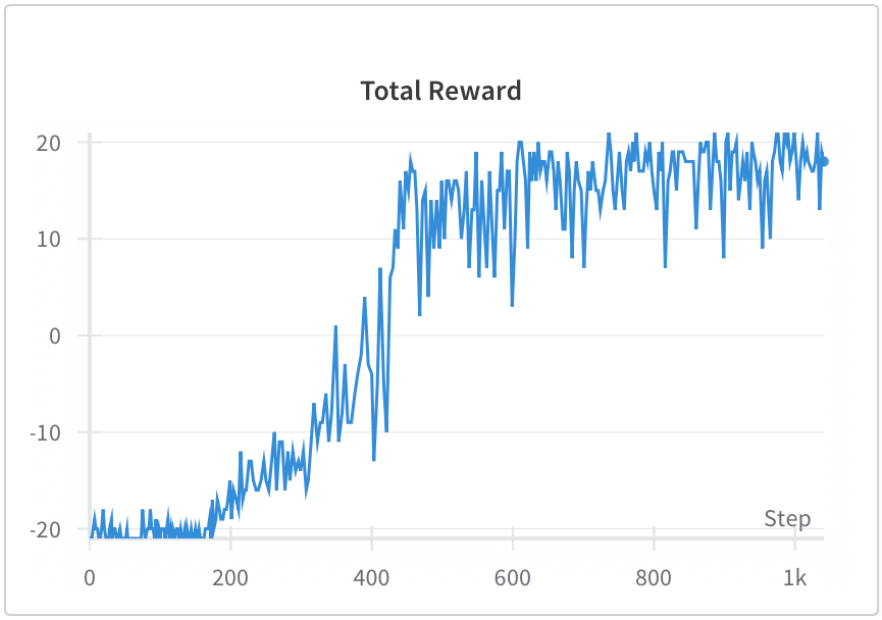
1. Cartpole:



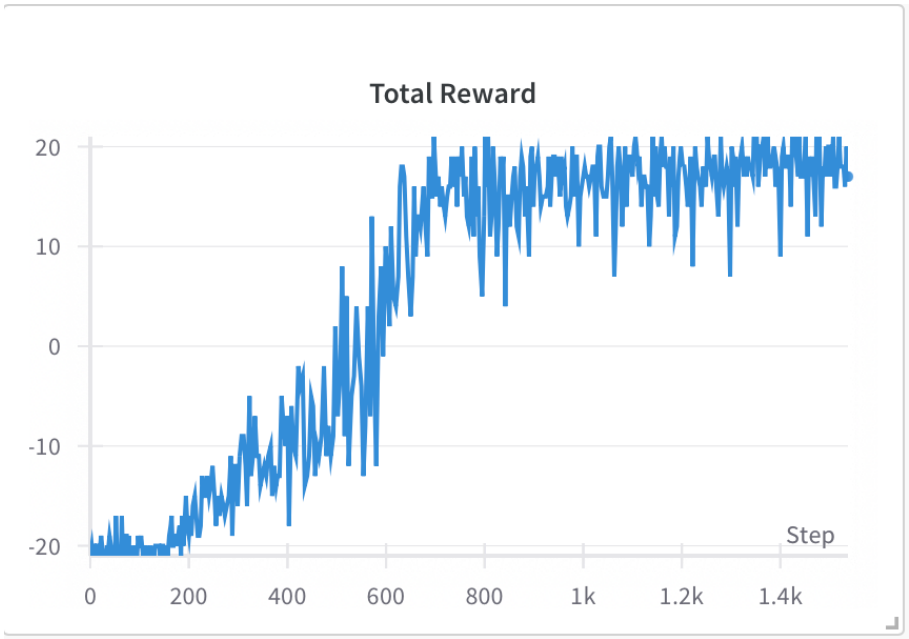
You can observe that during the initial phase before around 2.5k steps, the model wasn't training due to an insufficient replay buffer size, resulting in scores mostly below 100. However, once training began, the score steadily increased and

reached 30 around epoch 2756. Although there were still frequent fluctuations afterward, the model managed to achieve scores as high as 500 multiple times!

2. Pong (Normal)

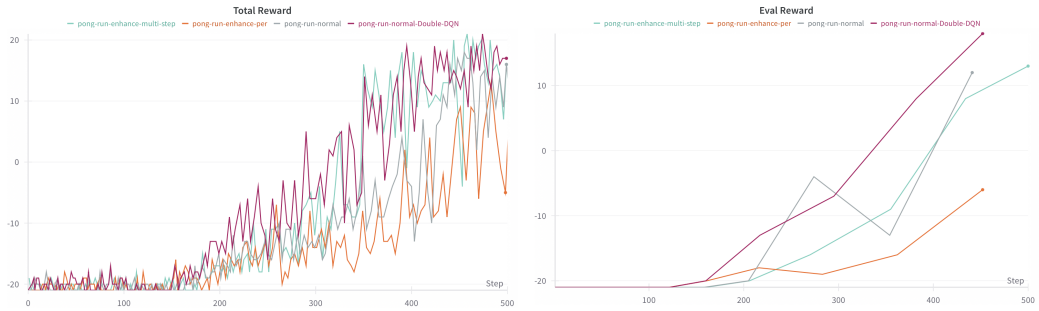


3.Pong (Enhancement)



2.Experiments:

a) enhancement method comparison:

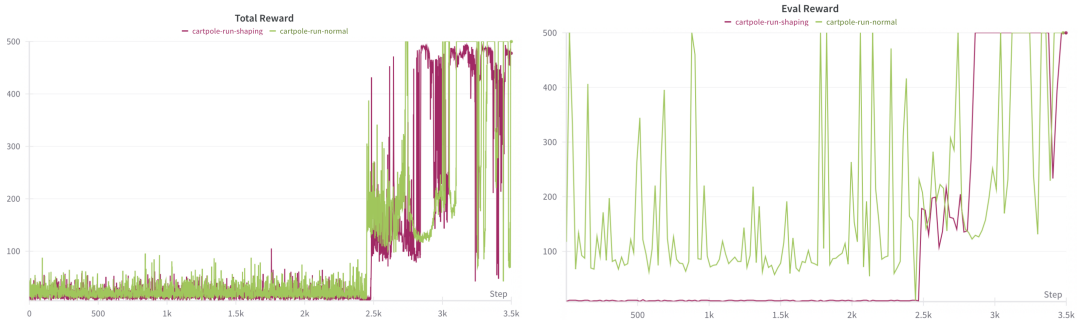


We compared the performance of the standard DQN with versions incorporating various enhancements. It is evident that the combination of DDQN and multi-step returns significantly improves performance. However, the performance of the PER-enhanced model is actually worse than the baseline. This may be due to the fact that the complexity of the Pong game does not require such a sophisticated model. Additionally, PER may introduce extra bias into the training process, making learning less stable and ultimately degrading overall performance.

b) Other training strategy: Reward_Shaping:

```
if self.shape:
    x, _, theta, _ = next_state
    x_norm = 1 - abs(x) / self.env.x_threshold
    theta_norm = 1 - abs(theta) / self.env.theta_threshold_radians
    reward = 0.5 * x_norm + 0.5 * theta_norm
```

To encourage the model to learn more strategically, we design the reward based on positional information. The goal is to prevent the agent from drifting away from its original position, which helps in keeping the pole upright effectively. We use the distance from the origin and the angle relative to the origin as evaluation criteria — the greater the deviation, the lower the score.



We can observe from the total rewards that, due to reward shaping being based on positional accuracy, the agent may not achieve the maximum possible score

because small position errors lead to reduced rewards. At first glance, this may make reward shaping seem less effective than the original reward scheme. However, its true impact is revealed during evaluation: not only does the learning curve become more stable with less fluctuation, but the agent is also able to consistently keep the pole upright in later episodes. Compared to the original approach, which remains unstable even in the later stages, this shows significant improvement — indicating that reward shaping is indeed an effective strategy.

Train command:

Cartpole:

Normal: python dqn1.py

Reward Shaping: python dqn1.py --shape

Pong:

Normal: python dqn2.py

DDQN: python dqn2.py --DDQN

N_steps: python dqn2.py --STEP

PER: python dqn2.py --PER

Test:

Task 1: python test_model1.py --model-path ./results/LAB5_110704071_task1_cartpole.pt

Task 2: python test_model2.py --model-path ./results/LAB5_110704071_task2_pong.pt

Task 3: python test_model2.py --model-path ./results/LAB5_110704071_task3_pong800000.pt