DLP
Lab 5: Value-Based Reinforcement Learning
Report

林睿騰(Rui-Teng Lin)
313551105

April 29, 2025

Contents

1 Introduction

In this lab, we will implement a simple value-based reinforcement learning algorithm, Q-learning, to solve the Cartpole and ALE Pong environment. Not only Q-learning, we will also apply some tricks to see if they can improve the performance of the algorithm.

1.1 Q-learning

Q-learning is a model-free off-policy algorithm that learns the optimal policy by maximizing the expected sum of future rewards.

The update rule can be written as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{1}$$

1.2 Double Q-learning

Since for Q-learning, the action-value function is estimated by maximizing the expected sum of future rewards, it is possible that the action-value function is overestimated.

To alleviate this problem, we can use double Q-learning, which uses two Q-networks to estimate the action-value function, its update rule can be written as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q'(s', \arg max_{a'} Q(s',a')) - Q(s,a)] \tag{2}$$

1.3 Prioritized Experience Replay

In Q-learning, we use replay memory to sample the transition (s,a,r,s') uniformly. However, some transitions are more informative than others, so we can use prioritized experience replay to sample the transitions with higher priority. Also, among all the transitions, some transitions are more likely to be sampled, so we can use importance sampling to correct the bias.

The priority and the weight of the transition can be calculated by:

$$p_i = \frac{1}{N} + \epsilon \tag{3}$$

$$w_i = \left(\frac{1}{N p_i}\right)^{\beta} \tag{4}$$

where N is the number of transitions in the replay memory, ϵ is a small constant to avoid the priority being zero, β is the exponent of the weight.

1.4 N-step Return

In Q-learning, we use the next-step return to update the action-value function, but in N-step Q-learning, we use the N-step return to update the action-value function.

The N-step return can be written as:

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n Q(s_{t+n}, a_{t+n}) \tag{5}$$

2 Implementation Details

2.1 Bellman Error

Calculation of the Bellman error in the original DQN algorithm is straightforward, as shown in the following code snippet:

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

# No gradient calculation for the target network
next_q_values = self.target_net(next_states).max(dim=1)[0]
next_q_values = next_q_values * (1 - dones)
target_q_values = rewards + (self.gamma ** self.n_step_return) *
next_q_values

loss = torch.nn.functional.mse_loss(q_values, target_q_values)
```

2.2 Double Q-learning

To convert the original DQN algorithm to double Q-learning, we need to modify the target network to estimate the action-value function.

```
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)
target_q_values = rewards + (self.gamma ** self.n_step_return) *
next_q_values
```

and the other part is the same as the original DQN algorithm.

2.3 Prioritized Experience Replay

To convert a normal replay buffer to a prioritized replay buffer, we need to modify the add method to calculate the priority of the transition. By default, the priority of newly added transition is set to infinity to let the agent pick it up in the next iteration and evaluate the state-action value.

```
def add(self, transition, error=np.inf):

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

    self.priorities[self.pos] = abs(error)
    self.pos = (self.pos + 1) % self.capacity

    return
```

And the sample method is modified to sample the transition with the probability proportional to the priority, also calculate the weights for the transitions. In the implementation, I normalized the weights of the importance sampling with the maximum weight to stabilize the training.

```
def sample(self, batch_size):

```

```
    probs = self.priorities[:len(self.buffer)]

    if np.any(probs):
        probs = np.where(probs == np.inf, 10, 1).astype(np.float32)
    else:
        probs = probs ** self.alpha

    probs /= probs.sum()

    indices = np.random.choice(len(self.buffer), size=batch_size, p=probs)
    samples = [self.buffer[i] for i in indices]
    weights = (len(self.buffer) * probs[indices]) ** (-self.beta)
    weights /= weights.sum()

    return samples, weights, indices
```

2.4 N-step Return

To calculate the n-step return, I use sliding window to store the n-step memory and calculate the n-step return. For implementation, I use deque to store the n-step memory and use the relation between the n-step return at each time step to calculate the n-step return of current and next state.

The relation between n-step return at time T and T + 1 can be derived as follows:

$$G_T = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-1} r_{t+n-1}$$
$$G_{T+1} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-2} r_{t+n-1} + \gamma^{n-1} r_{t+n}$$

From these equations, we can see that:

$$G_{T+1} = \frac{G_T - r_t}{\gamma} + \gamma^{n-1} r_{t+n}$$

This relation allows us to efficiently calculate the n-step return by maintaining a running sum and updating it incrementally, rather than recomputing the entire sum at each step. This is reflected in the implementation where we subtract the oldest reward and divide by gamma to get the next n-step return:

```
if len(self.n_step_memory) == self.n_step_return:
    ret = self.n_step_returns
    s, a, r, s_, d = self.n_step_memory.popleft()
    self.memory.add((s, a, ret, next_state, done))

    self.n_step_returns -= r
    self.n_step_returns /= self.gamma

self.n_step_returns += reward * (self.gamma ** len(self.n_step_memory))
self.n_step_memory.append((state, action, reward, next_state, done))
```

2.5 WandB Logging

For logging, I records the loss, reward, step, and other metrics during the training.

```
wandb.log(
    {
        "train/episode": ep,
        "train/total_reward": total_reward,
        "train/env_step_count": self.env_steps,
        "train/update_count": self.train_steps,
        "train/epsilon": self.epsilon,
    }
)

wandb.log(
    {
        "eval/env_step_count": self.env_steps,
        "eval/update_count": self.train_steps,
        "eval/reward": eval_reward,
    }
)

wandb.log(
    {
        "update/loss": loss.item(),
        "update/q_values": q_values.mean().item(),
        "update/target_q_values": target_q_values.mean().item(),
        "update/step": self.train_steps,
    }
)
```

And other than the mettices, I also record the training config for better analysis.

```
wandb.init(
    project=f"DLP-Lab5-DQN-{args.env_name.replace('/', '-')}",
    name=f"...",
    save_code=False,
    config=args,
)
```

3 Analysis and Discussion

3.1 Training Curves

Task 1:

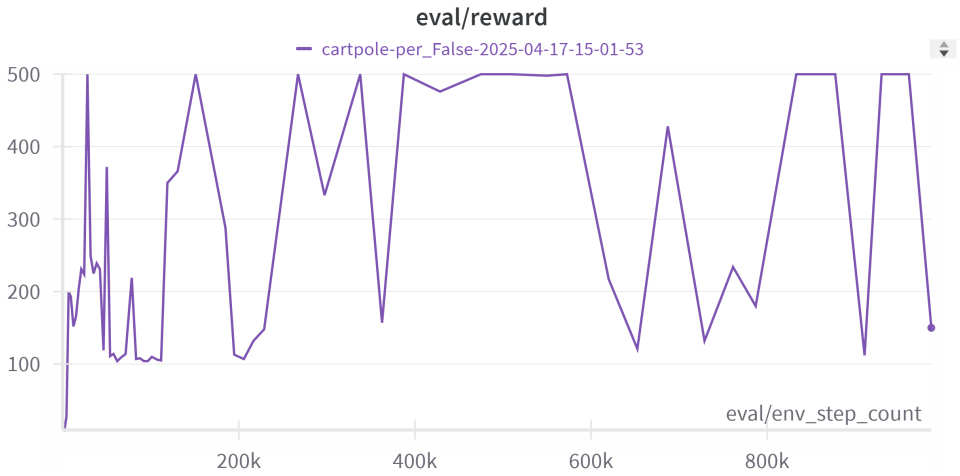

Figure 1: Training curves for Task 1

Task 2:

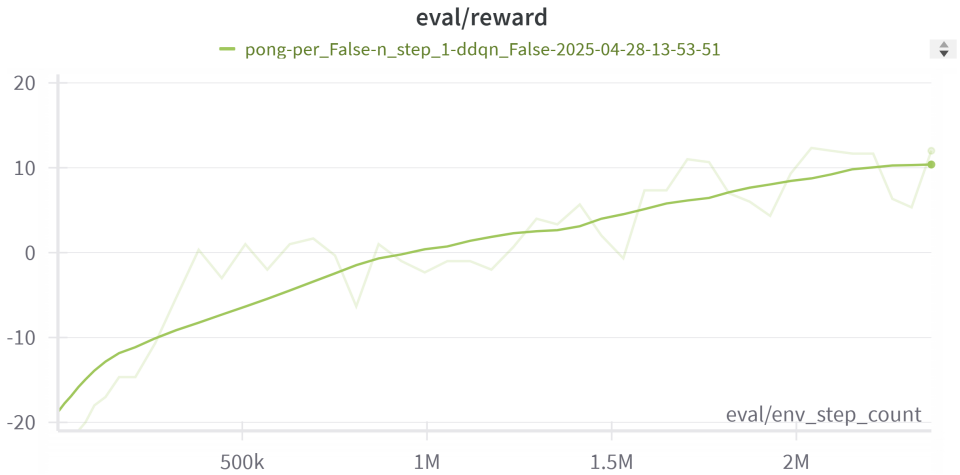

Figure 2: Training curves for Task 2

Task 3:

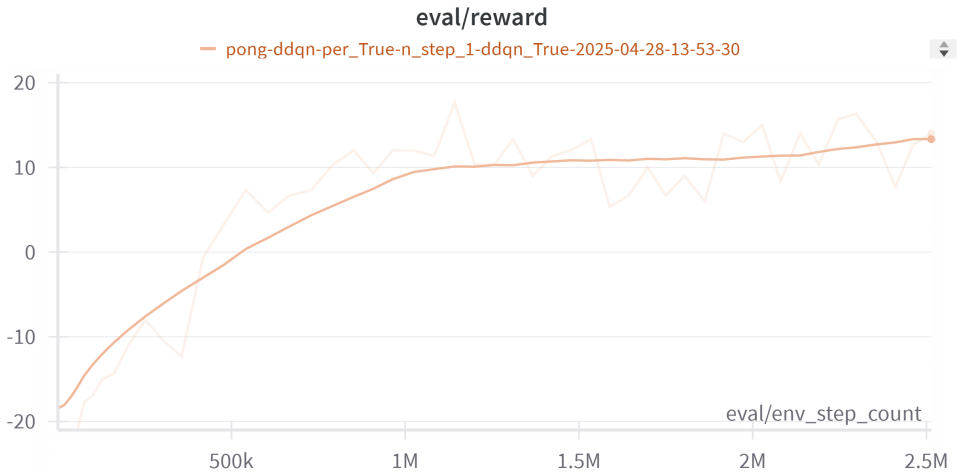

Figure 3: Training curves for Task 3

3.2 Sample Efficiency

It is obvious that the orange line (PER) is more sample efficient than the green line (without PER).

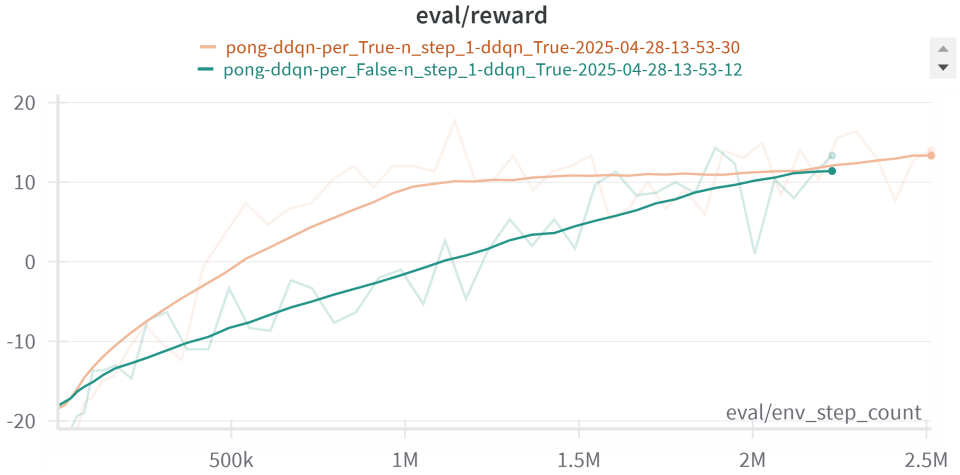

Figure 4: Sample efficiency comparison