Lab5 313605019 方敏

A. Introduction

This lab explores value-based reinforcement learning using Deep Q-Networks through three tasks. First begin with implementing a basic DQN in the simple CartPole environment. Second, extend it to handle high-dimensional visual input in Atari Pong, and at last, integrate advanced techniques such as Double DQN, Prioritized Experience Replay, and multi-step return to improve learning efficiency and overall performance.

The models are implemented using PyTorch and use Weights & Biases for training progress and monitoring. This report presents the briefly implementation details, training results, and comparative analysis of each task, highlighting the effects of various enhancements on learning effectiveness and sample efficiency. A detailed explanation of the code and training logic will be presented in the demo video.

B. Implementation

(1) Task1:

Task 1 does not require the use of AtariPreprocessor or PrioritizedReplayBuffer.

```
########## YOUR CODE HERE (5~10 lines) ##########

# MLP network (CartPole)
self.network = nn.Sequential(
    nn.Linear(4, 128),
    nn.ReLU(),
    nn.Linear(128, 128),
    nn.ReLU(),
    nn.Linear(128, num_actions)
)

########## END OF YOUR CODE ##########
```

This block defines a three-layer fully connected MLP used to approximate the Q-function in DQN. The input dimension is 4, maps to the CartPole state space: cart position, cart velocity, pole angle, and pole angular velocity. The output dimension is num_actions, corresponds to the 2 actions (moving left or right). The two hidden layers use ReLU with 128 units.

```
########## YOUR CODE HERE  ##########
# Add additional wandb logs for debugging if needed
wandb.log({
    "Step Total Reward": total_reward,
    "Step Epsilon": self.epsilon
})
########## END OF YOUR CODE ##########
```

This block records the total reward of each episode and the current epsilon to Weights & Biases platform for visualization and analysis.

```
########## YOUR CODE HERE (<5 lines) ##########
# Sample a mini-batch of (s,a,r,s',done) from the replay buffer
batch = random.sample(self.memory, self.batch_size)
states, actions, rewards, next_states, dones = zip(*batch)


########## END OF YOUR CODE ##########
```

This block implements the experience replay mechanism in DQN. It randomly samples (number = batch_size) transitions from self.memory. Each transition is a tuple of (state, action, reward, next_state, done), represents the agent experienced at one time step.

```
########## YOUR CODE HERE (~10 lines) ##########
# Implement the loss function of DQN and the gradient updates
states = torch.tensor(states, dtype=torch.float32).to(self.device)
next_states = torch.tensor(next_states, dtype=torch.float32).to(self.device)
actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
dones = torch.tensor(dones, dtype=torch.float32).to(self.device)

q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
next_q_values = self.target_net(next_states).max(1)[0]
target_q = rewards + self.gamma * next_q_values * (1 - dones)

loss = nn.MSELoss()(q_values, target_q.detach())
self.optimizer.zero_grad(); loss.backward(); self.optimizer.step()


########## END OF YOUR CODE ##########
```

This block is the core of the DQN training algorithm. First, the sampled data is converted to tensors and moved to GPU. Then, q_net is used to calculate the Q-values for the current states and actions, and target_net predicts the maximum Q-value of the next states. Based on the Bellman equation, the target Q-values are calculated.

Mean squared error computes the temporal difference (TD) error. Finally, clear gradients, do backpropagation, and update the network parameters.

(2) Task2:

```
########## YOUR CODE HERE (5~10 lines) ##########

self.network = nn.Sequential(
    nn.Conv2d(4, 32, kernel_size=8, stride=4),   # (N, 32, 20, 20)
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=4, stride=2),  # (N, 64, 9, 9)
    nn.ReLU(),
    nn.Conv2d(64, 64, kernel_size=3, stride=1),  # (N, 64, 7, 7)
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64*7*7, 512),
    nn.ReLU(),
    nn.Linear(512, num_actions)
)
########## END OF YOUR CODE ##########
```

This block modifies the network into a convolutional architecture of DQN, it is designed to process visual inputs from Atari games. The input has a shape of (4, 84, 84), which representing a stack of four grayscale frames over time. The image passes through three convolutional layers and then flattened and fed into two fully connected layers. The final output is a vector of Q-values for each possible action. In Pong, num_actions is 6..

```
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.buffer = deque(maxlen=capacity)

    def add(self, transition):
        ########## YOUR CODE HERE (for Task 3) ##########
        self.buffer.append(transition)
        ########## END OF YOUR CODE (for Task 3) ##########

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

    def sample(self, batch_size):
        ########## YOUR CODE HERE (for Task 3) ##########
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (
            np.stack(states),
            np.array(actions),
            np.array(rewards, dtype=np.float32),
            np.stack(next_states),
            np.array(dones, dtype=np.float32)
        )
        ########## END OF YOUR CODE (for Task 3) ##########
    def update_priorities(self, indices, errors):
        ########## YOUR CODE HERE (for Task 3) ##########

        ########## END OF YOUR CODE (for Task 3) ##########
        return
```

Here, I added a new __len__ function to return the current number of samples in the replay buffer. Because if without this function, calling len(self.memory) would raise an error because memory is a custom class, not

a native Python list.

In sample function, I randomly samples a batch of experience tuples from the replay buffer and unpacks them into five elements: (state, action, reward, next_state, done), each representing the result of the agent's interaction at a single time step. The update_priorities function is not implemented because Prioritized Experience Replay (PER) is only used in Task 3.

```
########## YOUR CODE HERE (<5 lines) ##########
# Sample a mini-batch of (s,a,r,s',done) from the replay buffer
states, actions, rewards, next_states, dones = self.memory.sample(self.batch_size)

########## END OF YOUR CODE ##########
```

This block extracts a batch of training samples from the replay buffer as a mini-batch.

```
########## YOUR CODE HERE (~10 lines) ##########
# Implement the loss function of DQN and the gradient updates
# Q(s, a)
states = torch.from_numpy(states).float().to(self.device) / 255.0
next_states = torch.from_numpy(next_states).float().to(self.device) / 255.0
actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
dones = torch.tensor(dones, dtype=torch.float32).to(self.device)
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
# max_a' Q_target(s', a')
with torch.no_grad():
    next_q_values = self.target_net(next_states).max(1)[0]
    target_q = rewards + self.gamma * next_q_values * (1 - dones)
loss = nn.MSELoss()(q_values, target_q)

self.optimizer.zero_grad()
loss.backward()
nn.utils.clip_grad_norm_(self.q_net.parameters(), 10.0)
self.optimizer.step()
########## END OF YOUR CODE ##########
```

This block implements the complete training procedure of DQN, is similar to Task 1. Some differences are: 1. The image pixel values are divided by 255 to normalize them into the [0, 1] range, it can improves training stability and efficiency. 2. The clip_grad_norm_ function is added to prevent gradient explosion.

(3) Task3:

- How do you obtain the Bellman error for DQN?

  in train()

```
with torch.no_grad():
    next_actions = self.q_net(next_states).argmax(1)
    next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
    target_q = rewards + self.gamma * next_q_values * (1 - dones)

current_q = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
td_errors = current_q - target_q
```

$$\delta = Q(s,a) - \left[r + \gamma \max_{a'} Q(s',a')\right]$$

Bellman error is TD error.

Use the Double DQN approach, the next action is selected by the main Q-network (q_net), and its Q-value is evaluated using the target network (target_net). The TD target is computed as the immediate reward plus the discounted Q-value of the next state, unless the episode has ended. The predicted Q-value is obtained from the main network, and the Bellman error is the difference between the predicted value and the TD target. This error will use for training and in the PER mechanism.

- How do you modify DQN to Double DQN?

  in train()

```
with torch.no_grad():
    next_actions = self.q_net(next_states).argmax(1)
    next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
    target_q = rewards + self.gamma * next_q_values * (1 - dones)

current_q = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
td_errors = current_q - target_q
```

Double DQN separates action selection and evaluation to reduce overestimation . The action is selected using the online network (q_net), but the value is evaluated using the target network (target_net).

This modification avoids overestimating Q-values and can improve stability.

- How do you implement the memory buffer for PER?

  in PrioritizedReplayBuffer.add()

  Adding experience with priority.

```
priority = (abs(error) + 1e-6) ** self.alpha
```

In PrioritizedReplayBuffer.sample()

Sampling based on priority.

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

In PrioritizedReplayBuffer.update_priorities()

Updating priorities

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

Prioritized Experience Replay (PER) samples transitions with higher TD errors more frequently. It can improve sample efficiency.

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

In DQNAgent.__init__()

Maintain n-step buffer.

```
self.n_step = args.n_step
self.n_step_buffer = deque(maxlen=self.n_step)
```

In DQNAgent.run()

Adding to buffer.

```
# Store transition in n-step buffer
self.n_step_buffer.append((state, action, reward, next_state, done))
```

In DQNAgent.compute_n_step()

Compute n-step return.

```
def compute_n_step(self):
    reward, next_state, done = 0, self.n_step_buffer[-1][3], self.n_step_buffer[-1][4]
    for idx in reversed(range(len(self.n_step_buffer))):
        r, n_s, d = self.n_step_buffer[idx][2:]
        reward = r + self.gamma * reward * (1 - d)
        if d:
            next_state, done = n_s, d
    return reward, next_state, done
```

In multi-step return, the agent considers cumulative rewards over multiple steps instead of only the next step.

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

Initialize

```
wandb.init(project="DLP-Lab5-DQN-Pong", name=args.wandb_run_name, save_code=True)
```

Logging during training

```
wandb.log({
    "Episode": ep,
    "Total Reward": total_reward,
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Epsilon": self.epsilon
})
```

Logging during evaluation

```
wandb.log({
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Eval Reward": eval_reward
})
```

C. Analysis and discussions

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

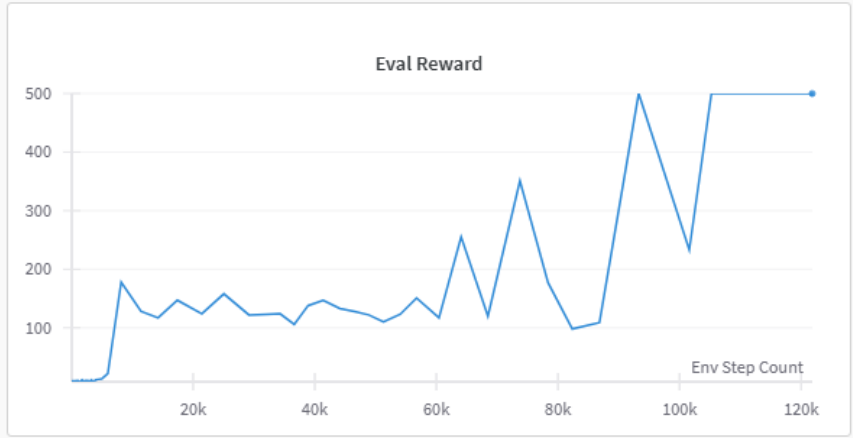
Task1:

Training command:

`python dqn_task1.py --wandb-run-name task1_cartpole --save-dir ./results_task1_cartpole --epsilon-decay 0.995 --replay-start-size 1000 --lr 0.001`

Testing command:

`python test_model_cartpole.py --model-path ./results_task1_cartpole/best_model.pt --output-dir demo_task1_cartpole --episodes 5 --seed 42`




```
Saved CartPole Episode 0 → Reward: 500.0 → demo_task1_cartpole\cartpole_ep0_r500.mp4
Saved CartPole Episode 1 → Reward: 500.0 → demo_task1_cartpole\cartpole_ep1_r500.mp4
Saved CartPole Episode 2 → Reward: 500.0 → demo_task1_cartpole\cartpole_ep2_r500.mp4
Saved CartPole Episode 3 → Reward: 500.0 → demo_task1_cartpole\cartpole_ep3_r500.mp4
Saved CartPole Episode 4 → Reward: 500.0 → demo_task1_cartpole\cartpole_ep4_r500.mp4
```

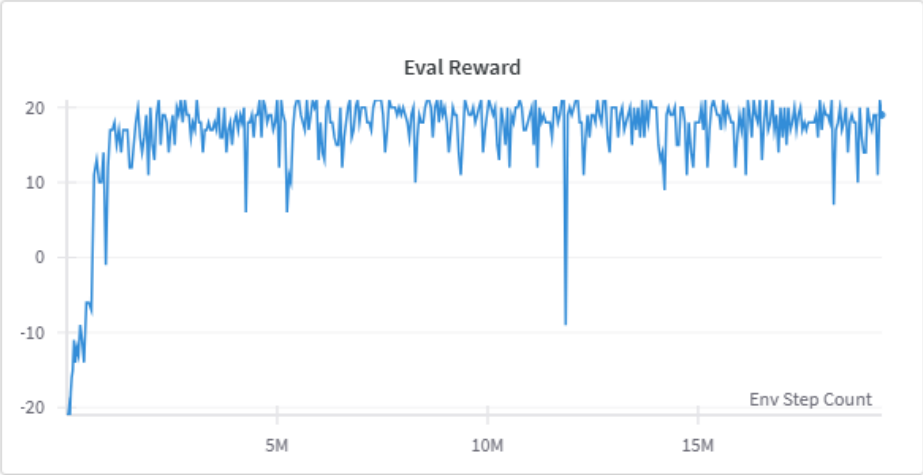
Task2:

Training command:

```
python dqn_task2.py --save-dir ./results_task2_pong --wandb-run-name task2_pong --batch-size 32 --memory-size 100000 --lr 0.0001 --discount-factor 0.99 --epsilon-start 1.0 --epsilon-decay 0.999995 --epsilon-min 0.05 --target-update-frequency 1000 --replay-start-size 50000 --max-episode-steps 10000 --train-per-step 1
```

Testing command:

```
python test_model.py --model-path results_task2_pong/best_model.pt --output-dir ./demo_task2_pong --episodes 20 --seed 42
```




```
Saved episode 0 with total reward 20.0 → ./eval_videos/ep2500\eval_ep0.mp4
Saved episode 1 with total reward 18.0 → ./eval_videos/ep2500\eval_ep1.mp4
Saved episode 2 with total reward 21.0 → ./eval_videos/ep2500\eval_ep2.mp4
Saved episode 3 with total reward 18.0 → ./eval_videos/ep2500\eval_ep3.mp4
Saved episode 4 with total reward 18.0 → ./eval_videos/ep2500\eval_ep4.mp4
Saved episode 5 with total reward 21.0 → ./eval_videos/ep2500\eval_ep5.mp4
Saved episode 6 with total reward 19.0 → ./eval_videos/ep2500\eval_ep6.mp4
Saved episode 7 with total reward 21.0 → ./eval_videos/ep2500\eval_ep7.mp4
Saved episode 8 with total reward 18.0 → ./eval_videos/ep2500\eval_ep8.mp4
Saved episode 9 with total reward 20.0 → ./eval_videos/ep2500\eval_ep9.mp4
Saved episode 10 with total reward 19.0 → ./eval_videos/ep2500\eval_ep10.mp4
Saved episode 11 with total reward 19.0 → ./eval_videos/ep2500\eval_ep11.mp4
Saved episode 12 with total reward 19.0 → ./eval_videos/ep2500\eval_ep12.mp4
Saved episode 13 with total reward 20.0 → ./eval_videos/ep2500\eval_ep13.mp4
Saved episode 14 with total reward 19.0 → ./eval_videos/ep2500\eval_ep14.mp4
Saved episode 15 with total reward 20.0 → ./eval_videos/ep2500\eval_ep15.mp4
Saved episode 16 with total reward 21.0 → ./eval_videos/ep2500\eval_ep16.mp4
Saved episode 17 with total reward 20.0 → ./eval_videos/ep2500\eval_ep17.mp4
Saved episode 18 with total reward 20.0 → ./eval_videos/ep2500\eval_ep18.mp4
Saved episode 19 with total reward 21.0 → ./eval_videos/ep2500\eval_ep19.mp4
```

Task3:

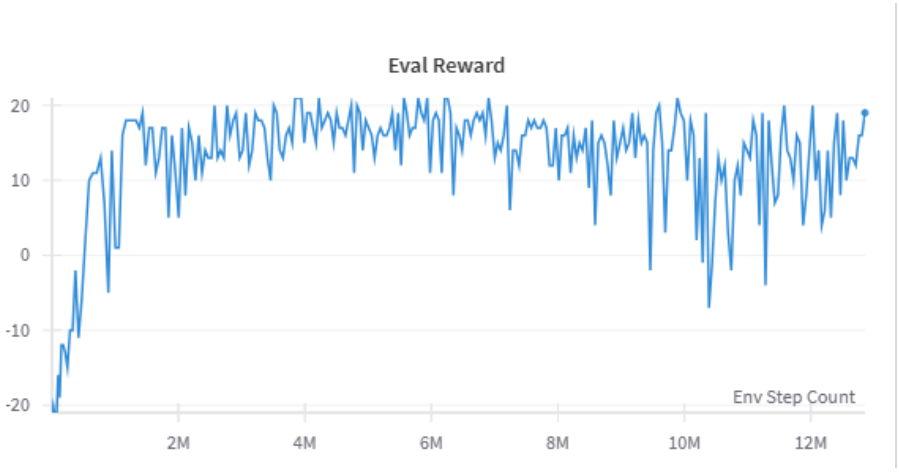
Training command:

```
python dqn_task3.py --save-dir ./results_task3_pong --wandb-run-name task3_pong_epsilonreset --n-step 3 --alpha 0.6 --beta 0.4 --batch-size 32
```

--memory-size 100000 --lr 0.0001 --discount-factor 0.99 --epsilon-start 1.0 --epsilon-decay 0.99999 --epsilon-min 0.05 --target-update-frequency 1000 --replay-start-size 50000 --max-episode-steps 10000 --train-per-step 1

Testing command:

python test_model.py --model-path results_task3_pong/best_model.pt --output-dir ./demo_task2_pong --episodes 20 --seed 42



I am not able to achieve a score above 19 within 1 million environment steps. 2M is limit.

(2) Analyze the sample efficiency with and without the DQN enhancements.

Double DQN separates action selection and evaluation to reduce overestimation . The action is selected using the online network (q_net), but the value is evaluated using the target network (target_net).

I think the performance of Task 3 was not as good as Task 2, because the training did not converge consistently, and rewards fluctuated more than in Task 2. It may because I didn't tune the hyperparameters carefully for each enhancement.

(3) Additional analysis on other training strategies

I applied input normalization by dividing pixel values by 255 and used clip_grad_norm_ to prevent gradient explosion. These techniques reduced my training time by more than half.