Deep Learning Lab5: Value-Based Reinforcement Learning

TAICA Student ID (NCU): 113522118

Name: 韓志鴻

1. Introduction

This report primarily presents my implementation and experimental results of the Deep Q-Network (DQN) algorithm in reinforcement learning tasks. First, in **Task 1**, I implemented the vanilla DQN in the low-dimensional CartPole-v1 environment; next, in **Task 2**, I trained a CNN-based model on the more complex Pong-v5; and finally, in **Task 3**, I introduced enhancements such as Double DQN, Prioritized Experience Replay (PER), and Multi-Step Returns. Final experiments show that, when the trained model is evaluated 100 episodes, its averages score can reach about 19.

The structure of this report is as follows:

- **Your Implementation**: This chapter describes key aspects of my implementations, including DQN, DDQN, and other enhancement techniques.
- **Analysis and Discussions**: An analysis and discussion of the training results for each task.
- **Additional Analysis on Other Training Strategies**: An exploration of improvements beyond the training strategies mentioned above.
- **Reference**: Any materials referenced during the completion of this lab5.

2. Your implementation

a、 How do you obtain the Bellman error for DQN?

i. Sample a batch of transitions from the replay buffer.

```
batch = random.sample(self.memory, self.batch_size)
states, actions, rewards, next_states, dones = zip(*batch)
```

ii. Compute the current Q-values.

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

iii. Compute the target values.

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

iv. Compute the Bellman error = target − q_values; this is calculated by the MSELoss and directly used to compute the DQN loss.

```
loss = nn.MSELoss()(q_values, target)
```

b、 How do you modify DQN to Double DQN?

i. The main difference between DDQN and DQN lies in the red-boxed term of the following formula, and its implementation is as follows:

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

ii. Use the Q-network (q_net) to select the action for each next state (next_states).

```
next_actions = self.q_net(next_states).argmax(1)
```

iii. Use the target network to evaluate the Q-values of those selected actions.

```
next_q = self.target_net(next_states).gather(1,next_actions.unsqueeze(1)).squeeze(1)
```

iv. Combine them to form the Double DQN target values.

```
target = rewards + (self.gamma**self.n_steps) * next_q * (1 - dones)
```

v. Finally, use the target in the loss computation.

```
loss = 0.5 * (weights.to(self.device) * (target.detach() - q_values).pow(2)).mean()
```

c、 How do you implement the memory buffer for PER?

i. First, initialize the class.

```
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta=0.4):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.pos = 0
```

ii. Add the new sample, following the formula below:

$$\boxed{\delta_i = r_i + \gamma \max_{a'} Q(s'_i, a') - Q(s_i, a_i)} \quad \boxed{p_i = |\delta_i| + \epsilon}$$

```
def add(self, transition, error):
    # Compute priority = (|error| + epsilon) ** alpha
    priority = (abs(error) + 1e-5) ** self.alpha
    if len(self.buffer) < self.capacity:
        # Buffer not full yet: append new transition
        self.buffer.append(transition)
    else:
        # Buffer full: overwrite the oldest transition
        self.buffer[self.pos] = transition
    # Update the priority at the current position
    self.priorities[self.pos] = priority
    # Move position pointer, wrap around if needed
    self.pos = (self.pos + 1) % self.capacity
```

iii. Sample from the buffer and update the weights.

$$\boxed{P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}} \quad \boxed{w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta}$$

```
def sample(self, batch_size):
    # Use full priorities if buffer is full;
    # otherwise use priorities up to current pos
    if len(self.buffer) == self.capacity:
        prios = self.priorities
    else:
        prios = self.priorities[:self.pos]
    # Calculate sampling probabilities: P(i) = prio_i / sum(prios)
```

```python
    probs = prios / prios.sum()
    # Randomly sample indices based on probabilities
    indices = np.random.choice(len(self.buffer), batch_size, p=probs)
    # Retrieve sampled transitions
    samples = [self.buffer[i] for i in indices]
    # Compute importance-sampling weights: w_i = (N * P(i))^-beta
    total = len(self.buffer)
    weights = (total * probs[indices]) ** (-self.beta)
    weights /= weights.max()
    return samples, indices, torch.tensor(weights, dtype=torch.float32)
```

iv. Update the priorities.

```python
def update_priorities(self, indices, errors):
    for idx, err in zip(indices, errors):
        # Compute new priority = (|error| + epsilon) ** alpha
        self.priorities[idx] = (abs(err) + 1e-5) ** self.alpha
```

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

i. During Agent initialization, create a deque of length n.

```python
self.n_step_buffer = deque(maxlen=self.n_steps)
```

ii. Call it at every interaction with the environment.

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

iii. Then follow the formula below:

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

The calculation for the red-boxed term is as follows:

```python
R = sum([self.n_step_buffer[i][2] * (self.gamma**i) for i in range(self.n_steps)])
```

iv. Then, package and store the n-step transition by taking the initial state and action from the first element in the buffer, and the next_state and done flag from the last element.

```python
s0, a0 = self.n_step_buffer[0][0], self.n_step_buffer[0][1]
sn, _, _, next_sn, dn = self.n_step_buffer[-1]
```

v. Including the green-boxed term in the formula, compute the preliminary TD-error, and pack it along with the prior experience into the PER buffer. Then use the newly computed TD-error as the priority P_i; these entries will subsequently be used for the DDQN calculations described in Task 2.

```python
# compute initial TD error for priority
with torch.no_grad():
    s0_t = torch.from_numpy(np.array(s0)).float().unsqueeze(0).to(self.device)
    next_sn_t = torch.from_numpy(np.array(next_sn)).float().unsqueeze(0).to(self.device)
    q0 = self.q_net(s0_t)[0, a0]
    next_q = self.target_net(next_sn_t).max(1)[0]
    td_error = (R + (self.gamma**self.n_steps) * next_q * (1 - dn) - q0).abs().item()
self.memory.add((s0, a0, R, next_sn, dn), td_error)
```

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

i. Initialize wandb.

```
wandb.init(project=args.wandb_project, name=args.wandb_run_name, save_code=True)
```

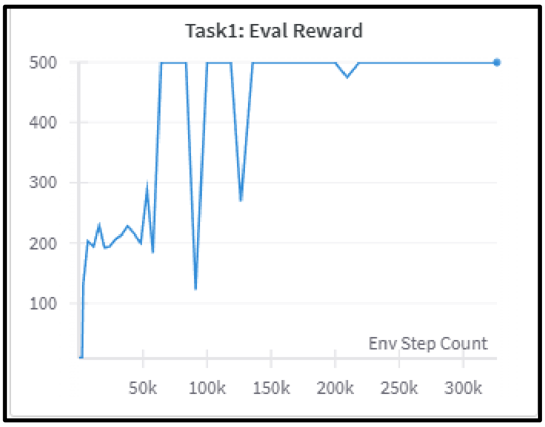
ii. Log the following parameters; the resulting line plots will serve as key reference indicators for whether the model is training in the right direction:

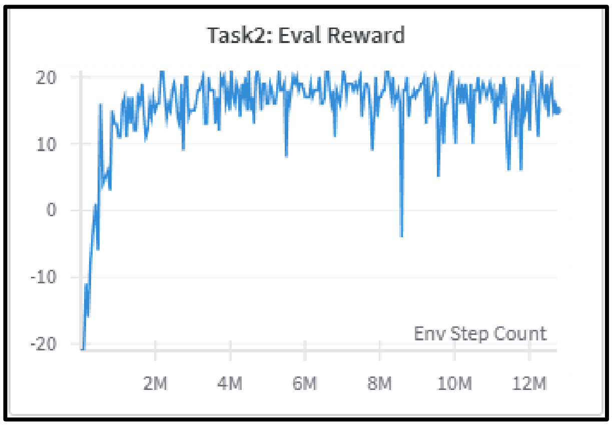
```
wandb.log({
    "Episode": ep,
    "Total Reward": total_reward,
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Epsilon": self.epsilon,
    "Learning Rate": self.scheduler.get_last_lr()[0], # new add
})
wandb.log({
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Eval Reward": eval_reward
})
```

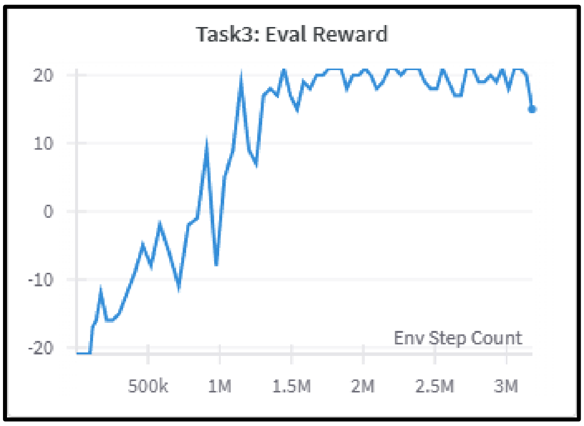
3. Analysis and discussions

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

i. For the three models trained in the above tasks, we computed the average score using test_model_task1.py, test_model_task2.py, and test_model_task3.py, with the number of **episodes** set to **100** and the **random seed** set to **113522118**.







b、 Analyze the sample efficiency with and without the DQN enhancements. If possible, perform an ablation study on each technique separately.

i. <u>With and without the DQN enhancements:</u>

Based on experiments, under the settings without the DQN enhancements it’s clear that the model requires a very large number of environment steps to exceed an average score of 19. As shown in the Task 2 Eval Reward plot above, the reward stabilizes at 19 after around 2 million environment steps; in contrast, with the DQN enhancements applied in Task 3, the same convergence to an Eval Reward of 19 is achieved with far fewer environment steps.

ii. <u>Ablation Study:</u>

According to the ablation experiments in the paper *Rainbow: Combining Improvements in Deep Reinforcement Learning*, as shown in Figure 1 below, the training performance ranks as **Rainbow > no priority (blue dashed line) > no multi-step (yellow dashed line)**. Among the two ablations—no priority and no multi-step—although no multi-step ends up slightly better in the later stages, no priority outperforms it for the majority of the training.

Comparing to the Task 3 DQN enhancements, I ran experiments where I removed Prioritized Experience Replay and Multi-Step Return and observed the corresponding reward curves—smoothed in the same way as in the Rainbow paper. In Figure 2 below, we can see that the training performance ranks as **DDQN Enhanced (black line) > no priority (blue line) > no multi-step (yellow line)**. Although the Task 3 enhancements aren’t a full implementation of Rainbow, **the ablation-study ordering still matches the results reported in the paper.**

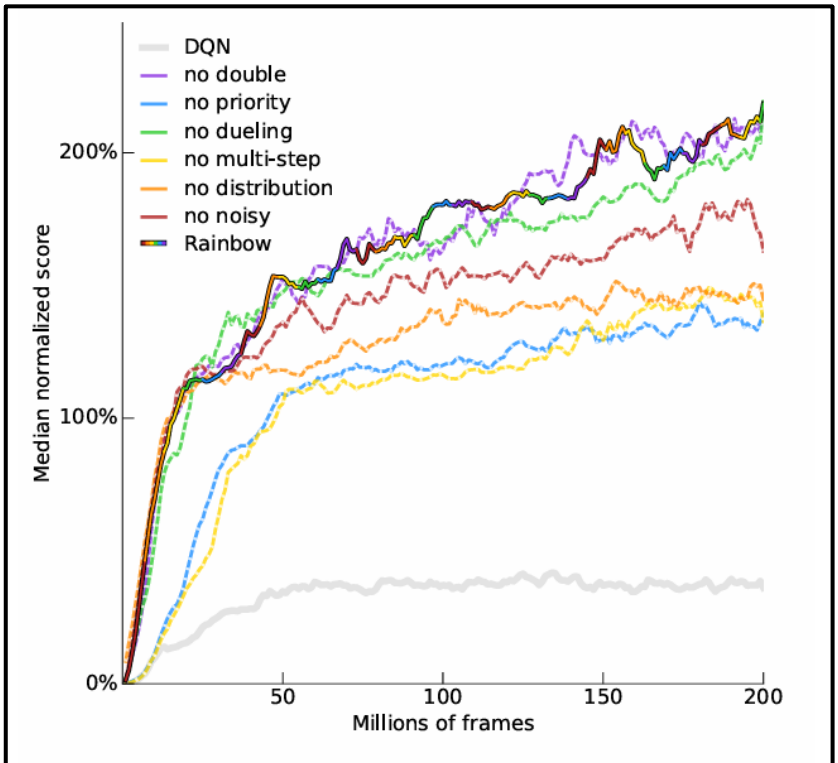


Figure 3: **Median human-normalized performance** across 57 Atari games, as a function of time. We compare our integrated agent (rainbow-colored) to DQN (gray) and to six different ablations (dashed lines). Curves are smoothed with a moving average over 5 points.

Fig1. Line plot of the ablation experiments from the Rainbow: Combining Improvements in

Deep Reinforcement Learning paper.

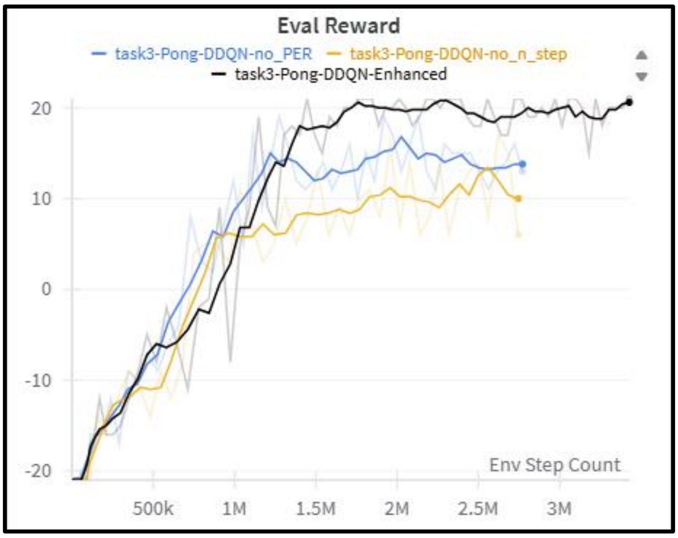


Fig2. Eval Reward – Env Step Count training curves. Curves are smoothed with a Running average over 5 points.

4. <u>Additional analysis on other training strategies</u>

a、 Use Dueling DQN

i. As shown in the architecture diagram below, Dueling DQN modifies the neural network architecture so that, at the output layer, it no longer directly outputs the Q-value, but instead produces two separate values:

- $V(s)$：For each state, there is a single scalar value.
- $A(s,a)$：For each state–action pair, there is a value.
- According to the Dueling DQN paper, the Q-value formula is as follows:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

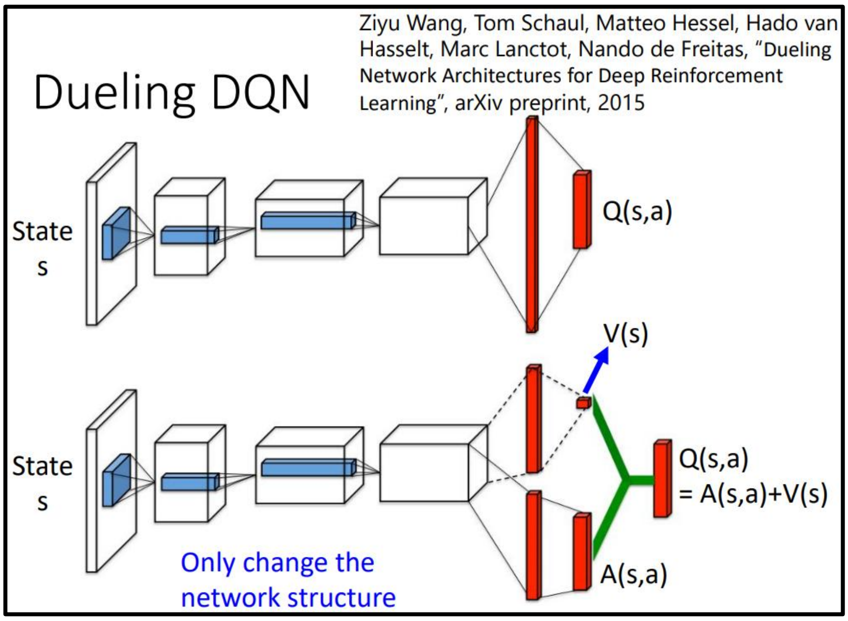


Fig. 3. Dueling DQN architecture (adapted from Professor Hung-yi Lee's lecture materials)

ii. Compared to the original vanilla DQN network, the Dueling DQN architecture offers the following advantages:

- More efficient learning of the state-value function:
  Vanilla DQN cannot distinguish between the inherent quality of a state and the quality of an action. The Dueling DQN architecture separates the state-value $V(s)$ and the advantage function $A(s,a)$ into two streams, allowing the network to focus on learning $V(s)$. When action advantages differ only slightly, this avoids wasting capacity on modify every individual Q-value. As shown below, updating $V(s)$ automatically updates all corresponding $Q(s,a)$ values via their summation.

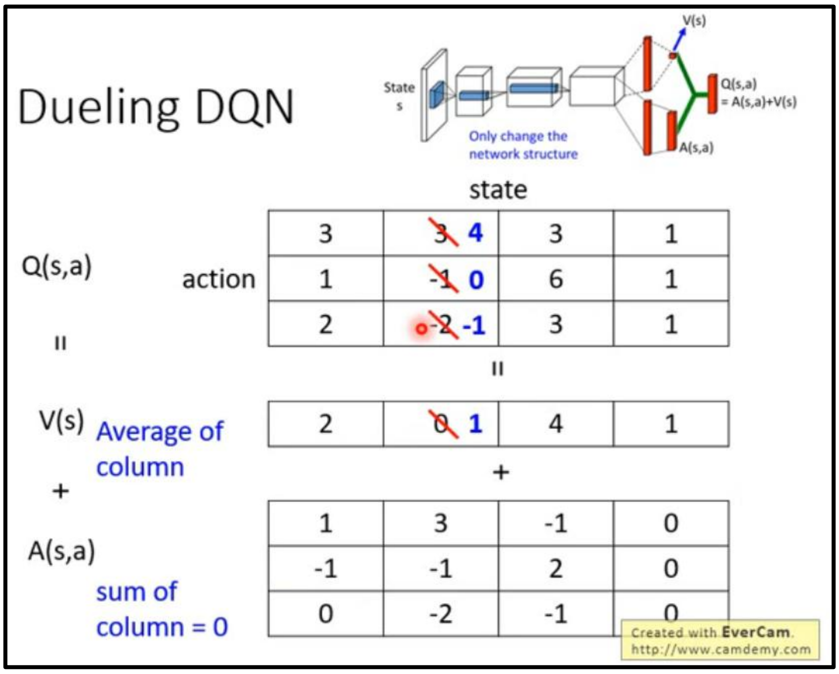


Fig. 4. Dueling DQN computation (adapted from Professor Hung-yi Lee's lecture materials)

- By splitting into two streams, for states where the differences between actions are small (e.g., when the ball is far from either paddle in Pong), the network can quickly learn a single $V(s)$ and only learn small $A(s,a)$ adjustments when necessary—improving sample efficiency compared to vanilla DQN.
- Dueling DQN essentially only modifies the network architecture by splitting the final layer into two streams. It makes no changes to the overall training loop (DDQN target calculation, PER, multi-step returns, target network updates), so it's easy to implement.

iii. The implemented Dueling DQN architecture is as follows:

```
class DuelingDQN(nn.Module):
    def __init__(self, num_actions):
        super(DuelingDQN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(4, 32, 8, 4),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, 4, 2),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, 3, 1),
            nn.ReLU(inplace=True),
```

```python
            nn.Flatten()
        )

        # shared projection
        self.fc = nn.Sequential(
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(inplace=True)
        )

        # Value and Advantage heads for dueling
        self.value = nn.Linear(512, 1)
        self.advantage = nn.Linear(512, num_actions)

    def forward(self, x):
        x = self.features(x.float() / 255.0)
        x = self.fc(x)
        v = self.value(x)         # V(s)
        a = self.advantage(x)     # A(s,a)
        # Q(s,a) = V(s) + (A(s,a) - mean_a A(s,a))
        return v + a - a.mean(dim=1, keepdim=True)
```

b、 Use Dynamic Learning Rate

In the default setup, the learning rate is static; however, toward the end of training this can lead to slow convergence or even gradient explosion. To address this, I incorporated PyTorch's optim.lr_scheduler.StepLR as a dynamic learning-rate strategy. This scheduler multiplies the current learning rate by a factor γ after a fixed number of steps, allowing the learning rate to decrease during later stages and thus stabilizing training.

There are many other dynamic learning-rate strategies—such as ReduceLROnPlateau, CosineAnnealingLR, and OneCycleLR—but due to time and hardware resource constraints, I wasn't able to test all of them. Furthermore, because reinforcement-learning training is unstable, it can be difficult to tune some of these schedulers' hyperparameters effectively. For simplicity, I chose StepLR, setting it to reduce the learning rate by a factor of 0.9 every 20,000 steps, so that the learning rate gradually decreases as training progresses.

5. Reference

[1] M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning," arXiv:1710.02298, 2017.

[2] 李宏毅_ATDL_DRL Lecture 4, https://hackmd.io/@shaoeChen/HyyXreFcB

[3] part4-DuelingDQN-PyTorch-Pong.ipynb, https://colab.research.google.com/drive/1EW7i4Jo_u2VbZAls7CVON_bKfFyKqKIn#sandboxMode=true&scrollTo=OvvBAoQVJsuU