Lab 5 Value-Based Reinforcement Learning

Shyang-En Weng 翁祥恩 413551036

April 29, 2025

1 Introduction

In this lab, we were working on the implementation of the Deep Q-Network (DQN) algorithm, a fundamental method in value-based reinforcement learning. DQN combines Q-learning with deep neural networks. We will go through the details of the equations and implement it into a pratical RL on Atari game CartPole and Pong-v5, making the reinforcement learning to work in both low-dimensional and high-dimensional (visual) environments.

1.1 Highlighted Points

The highlighted points of the implementation are shown below:

- I implemented the vanilla DQN algorithm, incorporating two essential techniques: experience replay and the use of a target network.
- Further enhancements were introduced to improve the performance of DQN, including Double DQN, Prioritized Experience Replay, and Multi-Step Return. Additionally, reward clipping and loss clipping were applied to further stabilize the training process.

2 Implementation Details

2.1 Task 1: Vanilla DQN in a toy environment

DQN [1] improves Q-learning by introducing a deep learning framework, using a neural network as a function approximator for the Q-value. The training of DQN is based on the squared Bellman error, minimizing the difference between the target Q-value (derived from rewards) and the estimated Q-value predicted by the network:

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

where s, a, and r are the current state, action, and reward, respectively, and s' and a' are the next state and action. γ is the discount factor, determining the

importance of future rewards. $\mathcal{A}$ is the action space, representing all possible actions. $Q(s', a'; \bar{\theta})$ is the Q-value for the next state-action pair with target network parameters $\bar{\theta}$, while $Q(s, a; \theta)$ is the Q-value for the current state-action pair with current network parameters θ. The term $r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \bar{\theta})$ is the target Q-value, and the difference $r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \bar{\theta}) - Q(s, a; \theta)$ represents the temporal difference (TD) error, which is minimized during training. Therefore, the target network is used evaluate the model based on bellman error.

Next, experience replay is also used in the DQN algorithm. Instead of learning directly from consecutive samples, experiences are stored in a replay buffer and randomly sampled during training. The sampling is defined as:

$$P(\text{sample } i) = \frac{1}{n}, \quad \forall i \tag{2}$$

This technique breaks the temporal correlations between samples, improves data efficiency, and stabilizes the training process by smoothing out the learning updates. In my implementation, I used a `deque` as the memory structure for the replay buffer, and randomly sampled batches during training.

```
# initialize
self.memory = deque(maxlen=args.memory_size)
# add transition to deque
self.memory.append((state, action, reward, next_state, done))
# randomly sample from deque
batch = random.sample(self.memory, self.batch_size)
```

2.2 Task 2: Vanilla DQN with Visual Observations on Atari

Compared to the CartPole environment, which is relatively simple and involves balancing a pole on a moving cart, Pong is more complex due to its image input instead of just four input features, its fast-paced gameplay, and the need for precise timing and strategy to compete against an opponent. The action space and state transitions in Pong are more challenging to model, requiring the agent to learn complex strategies for paddle control and ball prediction in a continuous, dynamic environment.

In this task, AtariPreprocessor is necessary to utilize for the preprocessing of the input images which it is not used in task 1 for CartPole. The experimence replay and the use of target network remain the same as task 1.

2.3 Task 3: Enhanced DQN

For enhanced DQN, I implemented three different improvement techniques, including Double DQN, Prioritized Experience Replay, and Multi-Step Return.

2.3.1 Double DQN

Double DQN [4] addresses the overestimation bias commonly seen in the standard DQN by decoupling the action selection and action evaluation steps. Instead of directly taking the maximum Q-value from the target network, Double DQN uses the current network to select the action and the target network to evaluate it, resulting in more accurate value estimates. The update rule is modified as follows:

$$L_{\text{DDQN}} := \frac{1}{2} \sum_{(s,a,r,s') \in D} \left[r + \gamma Q \left(s', \arg\max_{a' \in \mathcal{A}} Q(s', a'; \theta); \bar{\theta} \right) - Q(s, a; \theta) \right]^2 \quad (3)$$

2.3.2 Prioritized Experience Replay

Prioritized Experience Replay (PER) [2] improves sampling efficiency by giving higher sampling probability to transitions with larger temporal-difference (TD) errors, defined as:

$$P(\text{sample } i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (4)$$

where p_i is the priority of sample i, typically defined based on the Bellman error (temporal-difference error), and α controls the degree of prioritization.

To correct the bias introduced by non-uniform sampling, importance sampling weights are applied during the loss calculation. Without correction, the agent would overfit to high-priority samples, leading to biased updates. The importance weight for each sampled transition i is computed as:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(\text{sample } i)} \right)^\beta, \quad (5)$$

where N is the total number of transitions in the replay buffer, $P(\text{sample } i)$ is the probability of sampling transition i, and $\beta \in [0, 1]$ controls the strength of the correction. For numerical stability, the IS weights are normalized as $\tilde{w}_i = \frac{w_i}{\max_j w_j}$.

During training, the loss for each sample is multiplied by its normalized importance weight, ensuring that updates remain unbiased and stable even when prioritized sampling is used. Therefore, combining both techniques, PER focuses the learning on more significant experiences that can have a greater impact on the Q-network updates.

2.3.3 Multi-Step Return

Multi-Step Return enhances learning by considering multiple future rewards instead of only the immediate reward. This helps propagate reward signals faster across states, leading to improved convergence, especially in environments with delayed rewards like Pong. The n-step return is calculated as:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n Q(s_{t+n}, a_{t+n}; \bar{\theta}) \tag{6}$$

where n is the number of steps, r_{t+k} is the reward received at time step $t+k$, and s_{t+n} is the state after n steps.

These enhancements together significantly improve the stability and performance of the DQN agent.

2.4 How do you obtain the Bellman error for DQN

The use of Bellman error for DQN is explained in Section 2.1. The Bellman error is used to quantify the difference between the predicted Q-value and the target Q-value for each state-action pair. This error is minimized during training to update the Q-function and improve the agent's decision-making. The implementation is shown as:

Listing 1: Bellman error for DQN.

```
q_values = self.q_net(states).gather(1,
    actions.unsqueeze(1)).squeeze(1)
with torch.no_grad():
    next_q = self.target_net(next_states).max(dim=1)[0]
    target_q = rewards + self.gamma * next_q * (1 - dones)
td_errors = target_q - q_values
loss = (td_errors ** 2).mean()
```

2.5 How do you modify DQN to Double DQN

The algorithm of DDQN [4] is explained in Section 2.3.1. For the implementation, it used the online q network to predict accurate action, then

Listing 2: DDQN.

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

td_errors = target_q - q_values
loss = (weights * td_errors ** 2).mean()
```

2.6 How do you implement the memory buffer for PER?

The algorithm of the memory buffer for PER is explained in Section 2.3.2. For the implementation, I used a `deque` to store both the buffer and the priorities.

When adding a new transition into the buffer, I also add the maximum priority value to the priority list to ensure new samples are sampled with high probability. During sampling, transitions are selected according to their priority, and importance-sampling weights are computed to correct the bias introduced by prioritized sampling. The code of PER and usage are shown bellow:

Listing 3: PER.

```
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, n_multi_step=1, gamma=0.99):
    self.capacity = capacity
    self.alpha = alpha
    self.beta = beta
    self.buffer = deque(maxlen=capacity)
    self.priorities = deque(maxlen=capacity)
    # self.priorities = np.zeros((capacity,), dtype=np.float32)
    # self.pos = 0
    self.n_multi_step = n_multi_step
    self.gamma = gamma

def add(self, transition, error):
    ########## YOUR CODE HERE (for Task 3) ##########
    # if len(self.buffer) > self.capacity:
    #     self.buffer.popleft()
    self.buffer.append(transition)
    priority = (error + 1e-6) ** self.alpha
    self.priorities.append(priority)
    # self.priorities[self.pos] = priority
    # self.pos = (self.pos + 1) % (self.capacity)

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

def sample(self, batch_size):
    ########## YOUR CODE HERE (for Task 3) ##########
    prios = np.array(self.priorities)
    probs = prios / prios.sum()
    indices = np.random.choice(len(self.buffer), batch_size, p=probs)
    samples = [self.buffer[idx] for idx in indices]

    # Compute importance-sampling weights
    total = len(self.buffer)
```

```
        weights = (total * probs[indices]) ** (-self.beta)
        weights /= weights.max()  # Normalize for stability

        adjusted_samples = samples

        ########## END OF YOUR CODE (for Task 3) ##########
        return adjusted_samples, indices, weights
```

Listing 4: PER usage.

```
# initialization
self.memory = PrioritizedReplayBuffer(
        args.memory_size, alpha=0.6, beta=0.4,
        n_multi_step=args.n_steps,
        gamma=args.discount_factor)
# add
max_priority = np.array(
    self.memory.priorities).max()
    if len(self.memory) > 0 else 1.0
self.memory.add(
    (state, action, reward, next_state, done),
    max_priority)

# sample
batch, indices, weights = self.memory.sample(self.batch_size)
## weights for the importance-sampling weights
weights = torch.tensor(weights, dtype=torch.float32).to(self.device)

# update
clip_error = abs(td_errors)
self.memory.update_priorities(
    indices, clip_error.detach().cpu().numpy())
```

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

The algorithm for n-step return [3] is explained in Section 2.3.3. For the implementation, the sampling part of the memory buffer is modified to account for multi-step returns, where the reward is adjusted over multiple steps, with each step's reward multiplied by γ^n if $n > 1$.

Listing 5: Multi-step return.

```
def sample(self, batch_size):
    ########## YOUR CODE HERE (for Task 3) ##########
    prios = np.array(self.priorities)
    probs = prios / prios.sum()
```

```
indices = np.random.choice(len(self.buffer), batch_size, p=probs)
samples = [self.buffer[idx] for idx in indices]

# Compute importance-sampling weights
total = len(self.buffer)
weights = (total * probs[indices]) ** (-self.beta)
weights /= weights.max()  # Normalize for stability

# Adjust rewards for n-step return
if self.n_multi_step > 1:
    adjusted_samples = []
    for i in indices:
        state, action, reward, next_state, done = self.buffer[i]
        cumulative_reward = reward
        for n in range(1, self.n_multi_step):
            # From 1 to n-1
            if i + n < len(self.buffer):
                _, _, next_reward, next_next_state, next_done =
                    self.buffer[i + n]
                cumulative_reward += (self.gamma ** n) * next_reward
                if next_done:
                    break
            else:
                break
        adjusted_samples.append(
            (state, action, cumulative_reward, next_state, done))
else:
    adjusted_samples = samples

########## END OF YOUR CODE (for Task 3) ##########
return adjusted_samples, indices, weights
```

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

For tracking the model performance, I monitored both the total rewards during training and the evaluation rewards. Specifically, during training, the total reward obtained at each environment step was recorded to observe the learning progress. Other important training statuses such as update count, episode number, and epsilon value were also recorded. These metrics allowed us to detect issues such as divergence, overfitting, or insufficient exploration, and made it easier to monitor the training process and optimize the model design and hyperparameters.

3 Analysis & Discussion

3.1 Training Results

3.1.1 Task 1

The training reward curve for Task 1 is shown in Figure 1 and 2, illustrating the learning progress of the DQN agent on the CartPole environment over time. The corresponding epsilon decay during training is shown in Figure 3. We can observe that after epsilon decays to zero, there is a sudden drop in rewards, followed by a continuous increase as the agent adapts and refines its policy. The sudden drop after epsilon decays to zero occurs because the agent loses the benefit of exploration, relying entirely on its current policy, which may still be suboptimal at that stage. Without further exploration, the agent may initially exploit imperfect actions, leading to a temporary decline in performance. However, as training continues, the policy improves through further updates, resulting in a steady increase in rewards.

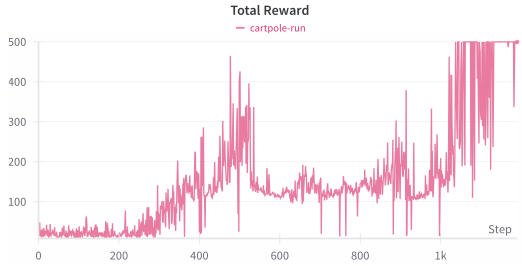

Figure 1: Training reward curve on CartPole for Task 1.

Figure 2: Evaluation reward curve on CartPole for Task 1.

3.1.2 Task 2

The training results for Task 2 are shown in Figures 4, 5, and 6. Compared to Task 1, solving the Pong-v5 environment with DQN becomes more challenging,

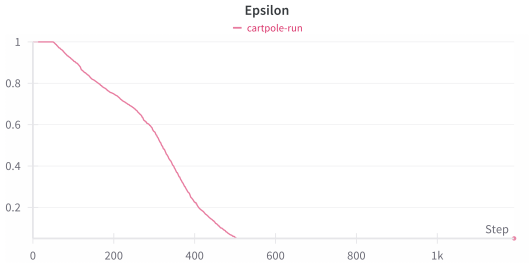

Figure 3: Epsilon decay curve on CartPole for Task 1.

as the agent takes longer to achieve high scores. This is likely due to the increased complexity of the environment, which requires the agent to learn more sophisticated strategies and handle continuous image input instead of simple state-action pairs.

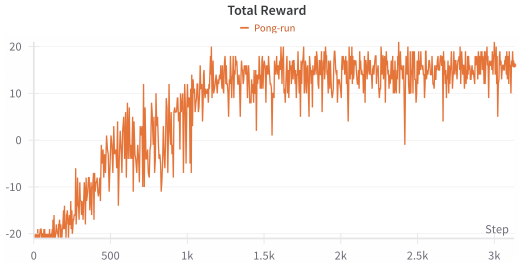

Figure 4: Training reward curve on Pong-v5 for Task 2.

3.1.3 Task 3

The training results for Task 3 are shown in Figures 7, 8, and 9. From these results, we observe that around the 1000 step (approximately 700k environment step), the reward begins to stabilize around a score of 15, and then gradually increases as training progresses.

3.2 Analyze the sample efficiency with and without the DQN enhancements

Compared to the results in Sections 3.1.2 and 3.1.3, we can see that DQN with enhancements enables the model to learn faster and achieve better performance. While the vanilla DQN reaches an evaluation reward of 19 around the 100k environment step (1.5k step), the enhanced DQN achieves the same reward by approximately the 90k environment step (1.3k step). This demonstrates that

Figure 5: Evaluation reward curve on Pong-v5 for Task 2.

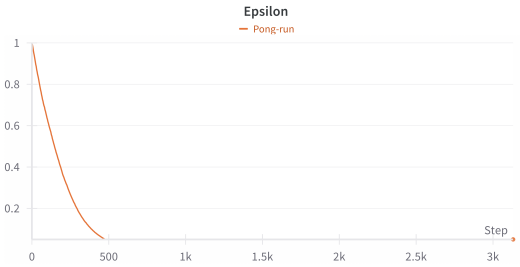

Figure 6: Epsilon decay curve on Pong-v5 for Task 2.

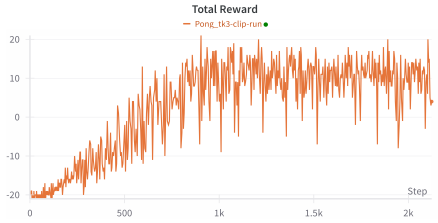

Figure 7: Total reward on Pong-v5 for task 3.

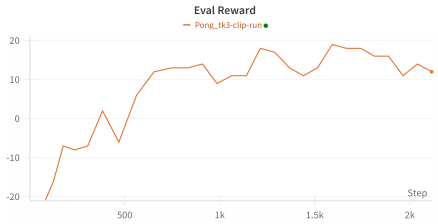

Figure 8: Evaluation reward on Pong-v5 for task 3.

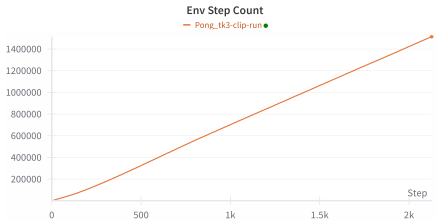

Figure 9: Environment Step Count on Pong-v5 for task 3.

the incorporated improvements significantly accelerate learning and increase training efficiency.

3.3 Additional analysis on other training strategies

I also applied clipping to the rewards and losses to stabilize the training, as introduced in the DDQN [4]. Clipping helps prevent extremely large gradients caused by outlier rewards or Q-value targets, which could otherwise destabilize the learning process. The implementation is shown in Listing 6. The tracked total and evaluation rewards also demonstrate that applying clipping improves the stability of training and leads to slightly better performance, as shown in Figure 10.

Listing 6: Rewards and losses clipping.

```
# Clip rewards for stability reasons
rewards = torch.clamp(rewards, -1, 1)
with torch.no_grad():
        next_q = self.target_net(next_states).max(dim=1)[0]
        target_q = rewards + self.gamma * next_q * (1 - dones)
td_errors = target_q - q_values
# Clip errors for stability reasons
torch.clamp(td_errors, -1, 1)
loss = (weights * td_errors ** 2).mean()
```

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[2] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

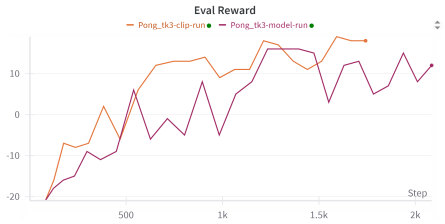

Figure 10: With and without Rewards and losses clipping for task 3. Red: without clipping, orange: with clipping.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[4] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.