Deep Learning Lab 5 - Value-Based Reinforcement Learning

110550088 李杰穎

April 30, 2025

1 Introduction

In this lab, I implement and enhance the Deep Q-Network (DQN) [4] algorithm using PyTorch across two distinct environments: the simple CartPole balancing task and the more complex Atari Pong game. The work progresses in strategic stages, beginning with a foundational implementation that successfully achieves maximum scores in both environments through three key components: uniform sampling for experience replay, 1-step returns for value estimation, and ϵ-greedy exploration.

Building on this solid baseline, based on observations in [2], I incorporate advanced DQN enhancements to dramatically improve sample efficiency:

1. **Huber Loss** to increase robustness against outliers and stabilize training
2. **Prioritized Experience Replay (PER)** with annealing β parameter to focus training on the most informative experiences
3. **Double DQN (DDQN)** to reduce overestimation bias in Q-value targets
4. **Multi-step Return** to propagate rewards more effectively through time
5. **Dueling DQN architecture** to separately estimate state values and action advantages
6. **Noisy Networks** for more effective state-dependent exploration
7. **Enhanced Frame Preprocessing** to increase the convergence speed of neural networks

The combination of these advanced techniques yields remarkable results: achieving a near-perfect score of 19 in Pong within just 200,000 environment steps, a significant improvement over the standard DQN implementation which typically requires millions of steps to reach comparable performance. This substantial efficiency gain demonstrates the power of integrating multiple algorithmic improvements in deep reinforcement learning.

This report is organized as follows: Section 2 details my implementation of the various DQN components and enhancements. Section 3 presents the experimental results and analysis, including training curves and ablation studies. Finally, Section 4 concludes with key findings and potential future work.

2 Implementation

2.1 Bellman Error for DQN

The Bellman error for Q-learning represents the difference between the current Q-value estimate and the target Q-value, which is defined as:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a), \tag{1}$$

where δ is the Bellman error (also called TD error), r is the current reward, γ is the discount factor, $Q(s, a)$ is the Q-value with given state s and action a, and s' is the next state.

In DQN, we approximate the Q-value function using a deep neural network. To train this network, we formulate the loss function as the mean squared error of the Bellman error. Formally, the loss function can be written as:

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

where θ represents the parameters of the main Q-network, $\bar{\theta}$ represents the parameters of the target Q-network, D is the experience replay buffer containing transitions (s, a, r, s'), r is the immediate reward, γ is the discount factor,

$Q(s, a; \theta)$ is the predicted Q-value for state s and action a, and $\max_{a' \in A} Q(s', a'; \bar{\theta})$ is the maximum Q-value for the next state s' across all possible actions.

The PyTorch implementation of this loss function is:

Code 1: **Implementation of the Bellman loss**

```python
# Get Q-values for current states and actions
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

# Calculate target Q-values using the target network
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

# Calculate TD errors for updating priorities
td_errors = target_q_values - q_values

# Calculate weighted MSE loss
loss = (weights * (td_errors ** 2)).mean()
```

2.2 Network Architecture

I implemented two different Q-network architectures: a Multi-Layer Perceptron (MLP) for the CartPole environment and a Convolutional Neural Network (CNN) for the Atari Pong environment. The implementation is flexible and supports various enhancements like dueling architecture and noisy networks:

Code 2: **Implementation of the vanilla DQN**

```python
class VanillaDQN(nn.Module):
    def __init__(self, input_dim, num_actions, conv=False, hidden_dim=512, dueling=False, noisy=False,
    ↪  vanilla=False):
        super(VanillaDQN, self).__init__()
        self.conv = conv
        self.dueling = dueling
        self.noisy = noisy
        self.num_actions = num_actions

        # Choose the appropriate linear layer based on noisy flag
        LinearLayer = NoisyLinear if noisy else nn.Linear
        self.multiplier = 2 if vanilla else 1
        if conv:
            self.network = nn.Sequential(
                nn.Conv2d(input_dim, 16 * self.multiplier, kernel_size=8, stride=4),
                nn.ReLU(),
                nn.Conv2d(16 * self.multiplier, 32 * self.multiplier, kernel_size=4, stride=2),
                nn.ReLU(),
                nn.Conv2d(32 * self.multiplier, 32 * self.multiplier, kernel_size=3, stride=1),
                nn.ReLU(),
                nn.Flatten(),
                LinearLayer(32 * 7 * 7 * self.multiplier, 512),
                nn.ReLU(),
                LinearLayer(512, num_actions)
            )
        else:
            self.network = nn.Sequential(
                LinearLayer(input_dim, hidden_dim),
                nn.ReLU(),
                LinearLayer(hidden_dim, hidden_dim),
                nn.ReLU(),
                LinearLayer(hidden_dim, num_actions)
            )

    def forward(self, x):
        if self.conv:
            x = x / 255.0

        return self.network(x)
```

2.3 Huber Loss

The Huber loss is a robust loss function that combines the benefits of mean squared error (MSE) for small errors and mean absolute error (MAE) for large errors. This makes it less sensitive to outliers than MSE, which is particularly important in RL where rewards and Q-values can sometimes have high variance.

As described in [3] in their seminal DQN paper, using the Huber loss instead of MSE can stabilize training by reducing the impact of large TD errors. The Huber loss is defined as:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta \\ \delta(|y - f(x)| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{3}$$

where δ is a hyperparameter that determines the threshold between the quadratic and linear regions.

In PyTorch, this is implemented using the `smooth_l1_loss` function:

Code 3: **Implementation of Huber loss**

```python
# Get Q-values for current states and actions
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

# Calculate target Q-values using the target network
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

# Calculate loss using Huber loss (smooth_l1_loss) with importance sampling weights
loss = (weights * torch.nn.functional.smooth_l1_loss(q_values, target_q_values.detach(),
↪  reduction='none')).mean()
```

2.4 Prioritized Experience Replay (PER)

Prioritized Experience Replay (PER) was introduced by [5] to address a key limitation of uniform sampling in experience replay: not all transitions are equally valuable for learning. PER prioritizes transitions with higher TD errors, as these represent experiences from which the agent can learn more.

The key components of PER include:

- **Priority Assignment**: Each transition is assigned a priority proportional to its TD error: $p_i = (|\delta_i| + \epsilon)^\alpha$, where ϵ is a small positive constant to ensure non-zero sampling probability and α controls the degree of prioritization.
- **Probability Calculation**: The probability of sampling transition i is: $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$
- **Importance Sampling**: To correct the bias introduced by prioritized sampling, importance sampling weights are applied to the loss: $w_i = (\frac{1}{N \cdot P(i)})^\beta$, where β is annealed from its initial value to 1 over the course of training.

My implementation features a gradual increase of β to ensure proper bias correction as training progresses:

Code 4: **Implementation of Prioritized Experience Replay**

```python
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=0.7, beta=0.4):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.beta_increment = (1.0 - beta) / 200_000  # Increment beta gradually
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.max_priority = 1.0
        self.pos = 0

    def add(self, transition, error=None):
        # Calculate priority based on TD error
        if error is None:
            priority = self.max_priority
        else:
```

```
            priority = (abs(error) + 1e-5) ** self.alpha

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

        # Update priority
        self.priorities[self.pos] = priority
        self.pos = (self.pos + 1) % self.capacity

    def sample(self, batch_size):
        # If buffer is empty or not enough samples, return None
        if len(self.buffer) == 0:
            return None, None, None, None, None, None
        if len(self.buffer) < batch_size:
            batch_size = len(self.buffer)

        # Calculate sampling probabilities
        priorities = self.priorities[:len(self.buffer)]
        probabilities = priorities / np.sum(priorities)

        # Sample indices based on the probabilities
        indices = np.random.choice(len(self.buffer), batch_size, p=probabilities, replace=False)

        # Calculate importance sampling weights
        weights = (len(self.buffer) * probabilities[indices]) ** (-self.beta)
        weights /= weights.max()  # Normalize weights

        # Gradually increase beta to 1
        self.beta = min(1.0, self.beta + self.beta_increment)

        # Get the sampled transitions
        batch = [self.buffer[i] for i in indices]
        states, actions, rewards, next_states, dones = zip(*batch)

        return states, actions, rewards, next_states, dones, indices, weights

    def update_priorities(self, indices, errors):
        # Update the priorities of the sampled transitions
        for idx, error in zip(indices, errors):
            # Ensure idx is within valid range
            if idx < len(self.buffer):
                # Calculate new priority and update
                self.priorities[idx] = (abs(error) + 1e-5) ** self.alpha
                self.max_priority = max(self.max_priority, self.priorities[idx])
        return
```

2.5 Double DQN

Double DQN (DDQN) was proposed by [7] to address the overestimation bias present in standard DQN. In vanilla DQN, both action selection and value estimation for the next state use the same target network, which can lead to overestimation of Q-values.

DDQN decouples action selection and value estimation by: 1. Using the online network to select the best action for the next state 2. Using the target network to evaluate the Q-value of that action

The DDQN loss function is expressed as:

$$L_{DDQN}(\theta) := frac12 \sum_{(s,a,r,s')\in D} \left(r + \gamma Q(s', \arg\max_{a'\in A} Q(s', a'; \theta); \bar{\theta}) - Q(s, a; \theta) \right)^2 \tag{4}$$

Here's my implementation of DDQN:

Code 5: **Implementation of the double DQN (DDQN)**

```
# Get Q-values for current states and actions
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

```

```
with torch.no_grad():
    # DDQN: Use online network to select actions, target network to evaluate
    next_actions = self.q_net(next_states).argmax(1, keepdim=True)
    next_q_values = self.target_net(next_states).gather(1, next_actions).squeeze(1)

    # Calculate target Q-values
    target_q_values = rewards + (1 - dones) * self.gamma * next_q_values

# Calculate loss using Huber loss with importance sampling weights
loss = (weights * torch.nn.functional.smooth_l1_loss(q_values, target_q_values.detach(),
↪  reduction='none')).mean()
```

2.6 Multi-step Return

Multi-step returns, as described in Sutton and Barto's Reinforcement Learning textbook [6], allow the agent to learn from longer reward sequences. Instead of just considering the immediate reward and the estimated value of the next state (1-step return), multi-step returns accumulate rewards over several time steps before bootstrapping with the estimated value.

The n-step return is defined as:

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q(s_{t+n}, a') \tag{5}$$

This approach provides a balance between bias and variance in the return estimates. A larger n reduces bias by incorporating more actual rewards, but can increase variance if the environment is stochastic.

In my implementation, I use a fixed-size deque as a buffer to store the last n transitions. For each new transition, I calculate the n-step return using the stored transitions and add it to the replay buffer:

Code 6: **Implementation of multi-step return**

```
# Main training loop
while not done and step_count < self.max_episode_steps:
    action = self.select_action(state)
    next_obs, reward, terminated, truncated, _ = self.env.step(action)
    done = terminated or truncated

    next_state = self.preprocessor.step(next_obs)

    # Add to n-step buffer
    self.n_step_buffer.append((state, action, reward, next_state, done))

    if len(self.n_step_buffer) == self.n_step:
        # Get initial state and action
        init_state, init_action, _, _, _ = self.n_step_buffer[0]

        # Calculate n-step return: sum of discounted rewards
        n_step_return = sum([r * (self.gamma ** i) for i, (_, _, r, _, _) in enumerate(self.n_step_buffer)])

        # Get final next state and done flag
        final_next_state = self.n_step_buffer[-1][3]
        final_done = self.n_step_buffer[-1][4]

        # Add to replay buffer with n-step information
        self.memory.add((init_state, init_action, n_step_return, final_next_state, final_done), error=None)

    elif self.n_step == 1:  # Fallback to 1-step return if n_step is set to 1
        self.memory.add((state, action, reward, next_state, done), error=None)
```

2.7 Dueling Network

The Dueling Network architecture, introduced by [8], represents a significant advancement in DQN design by decomposing the Q-function into two separate components: a state value function $V(s)$ and an advantage function $A(s, a)$. This decomposition allows the network to learn which states are valuable independently from learning the effect of each action.

Mathematically, the standard Q-function can be expressed as:

$$Q(s,a) = V(s) + A(s,a) \tag{6}$$

Where $V(s)$ represents the intrinsic value of being in state s regardless of the action taken, and $A(s,a)$ represents the relative advantage of taking action a in state s compared to other actions.

However, this decomposition creates an identifiability issue because adding a constant to $V(s)$ and subtracting the same constant from $A(s,a)$ results in the same Q-value. To address this, the dueling architecture forces the advantage function to have zero mean across all actions:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|} \sum_{a'} A(s,a') \right) \tag{7}$$

In my implementation, I created separate value and advantage streams after the shared feature extraction layers:

Code 7: **Implementation of the Dueling Network**

```python
class DQN(nn.Module):
    """
    DQN with optional dueling architecture and noisy networks
    """
    def __init__(self, input_dim, num_actions, conv=False, hidden_dim=64, dueling=False, noisy=False):
        super(DQN, self).__init__()
        self.conv = conv
        self.dueling = dueling
        self.noisy = noisy
        self.num_actions = num_actions

        # Choose the appropriate linear layer based on noisy flag
        LinearLayer = NoisyLinear if noisy else nn.Linear

        if conv:
            # Feature extraction layers for convolutional input
            self.features = nn.Sequential(
                nn.Conv2d(input_dim, 16, kernel_size=8, stride=4),
                nn.ReLU(),
                nn.Conv2d(16, 32, kernel_size=4, stride=2),
                nn.ReLU(),
                nn.Conv2d(32, 32, kernel_size=3, stride=1),
                nn.ReLU(),
                nn.Flatten()
            )

            feature_output_dim = 32 * 7 * 7
            self.value_stream = nn.Sequential(
                LinearLayer(feature_output_dim, 256),
                nn.ReLU(),
                LinearLayer(256, 1)
            )

            self.advantage_stream = nn.Sequential(
                LinearLayer(feature_output_dim, 256),
                nn.ReLU(),
                LinearLayer(256, num_actions)
            )
    def forward(self, x):
        if self.conv:
            x = x / 255.0

        features = self.features(x)
        values = self.value_stream(features)
        advantages = self.advantage_stream(features)

        # Combine value and advantage streams using the dueling formula
        # Q(s,a) = V(s) + (A(s,a) - mean(A(s,:)))
        return values + (advantages - advantages.mean(dim=1, keepdim=True))
```

The dueling architecture offers several key benefits that improved my DQN implementation:

1. **Improved state value estimation**: The dedicated value stream can learn the value of being in specific states without considering all possible actions, which is particularly important in Pong where recognizing advantageous ball positions is critical.

2. **More efficient action selection**: The advantage stream focuses exclusively on determining which actions are better than others in a given state, rather than trying to learn absolute Q-values for each action.

3. **Reduced overestimation bias**: By separating value and advantage estimation, the architecture helps mitigate the overestimation bias that often affects standard DQN, complementing the effect of Double DQN.

4. **Better generalization**: The value stream generalizes across actions, providing better estimates for states even when certain actions are rarely taken during training.

2.8 Noisy Networks

Noisy Networks, introduced by [1], offer a more sophisticated exploration strategy compared to ϵ-greedy. Instead of randomly selecting actions with a fixed probability, Noisy Networks inject parameterized noise directly into the network weights, allowing for state-dependent exploration.

The key advantages of Noisy Networks include:

1. State-dependent exploration that adapts to the agent's uncertainty

2. No need for manually decreasing exploration rates

3. More directed exploration compared to random ϵ-greedy actions

In my implementation, I replace standard linear layers with noisy linear layers that add parameterized noise to weights and biases during forward passes:

Code 8: **Implementation of the Noisy linear layer**

```python
class NoisyLinear(nn.Module):
    """
    Noisy Linear Layer for exploration
    Based on the paper: Noisy Networks for Exploration (Fortunato et al., 2018)
    """
    def __init__(self, in_features, out_features, sigma_init=0.5):
        super(NoisyLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.sigma_init = sigma_init

        # Learnable parameters
        self.weight_mu = nn.Parameter(torch.Tensor(out_features, in_features))
        self.weight_sigma = nn.Parameter(torch.Tensor(out_features, in_features))
        self.bias_mu = nn.Parameter(torch.Tensor(out_features))
        self.bias_sigma = nn.Parameter(torch.Tensor(out_features))

        # Register buffers for noise
        self.register_buffer('weight_epsilon', torch.Tensor(out_features, in_features))
        self.register_buffer('bias_epsilon', torch.Tensor(out_features))

        # Initialize parameters
        self.reset_parameters()
        self.reset_noise()

    def reset_parameters(self):
        mu_range = 1 / np.sqrt(self.in_features)
        self.weight_mu.data.uniform_(-mu_range, mu_range)
        self.weight_sigma.data.fill_(self.sigma_init / np.sqrt(self.in_features))
        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(self.sigma_init / np.sqrt(self.out_features))

    def reset_noise(self):
        epsilon_in = self._scale_noise(self.in_features)
        epsilon_out = self._scale_noise(self.out_features)

        # Factorized Gaussian noise (outer product)
        self.weight_epsilon.copy_(torch.outer(epsilon_out, epsilon_in))
        self.bias_epsilon.copy_(epsilon_out)

    def _scale_noise(self, size):
        # Generate Gaussian noise and apply scaling transformation f(x) = sgn(x) * sqrt(|x|)
        x = torch.randn(size)
        return x.sign().mul_(x.abs().sqrt_())
```

```
45
46      def forward(self, x):
47          if self.training:
48              # During training, use noisy weights
49              weight = self.weight_mu + self.weight_sigma * self.weight_epsilon
50              bias = self.bias_mu + self.bias_sigma * self.bias_epsilon
51          else:
52              # During evaluation, use just the expected weights
53              weight = self.weight_mu
54              bias = self.bias_mu
55
56          return F.linear(x, weight, bias)
```

2.9 Enhanced Frame Preprocessing

Effective preprocessing of visual inputs is crucial for DQN performance in Atari environments. I implemented several key optimizations beyond standard techniques to significantly improve both learning efficiency and final performance. My preprocessing pipeline includes:

1. **Selective region cropping**: I removed the score area and other irrelevant regions (rows 34-194) from the original 210×160 frames. This focused the network's attention on the gameplay area while reducing input dimensionality by approximately 25%.

2. **Binary thresholding**: After grayscale conversion and resizing, I applied binary thresholding with a threshold value of 127. This transformation creates high-contrast frames where the paddles and ball are clearly distinguished from the background. For Pong specifically, this simplification allows the CNN to more easily identify critical game elements without being distracted by unnecessary gradient information.

3. **Frame stacking**: To capture temporal dynamics (particularly critical for determining ball direction and speed), I maintained a history of 4 consecutive frames. This standard technique from the original DQN paper provides the network with motion information despite using static convolutional layers.

These preprocessing steps collectively transform the raw 210×160×3 RGB input into a more efficient 84×84×4 representation of binary frames. The binary thresholding proved particularly effective for Pong, where the high-contrast representation made it easier for the network to track the small, fast-moving balls.

Code 9: **Implementation of Enhanced Frame Preprocessing**

```
1  class AtariPreprocessor:
2      """
3      Preprocessing the state input of DQN for Atari
4      """
5      def __init__(self, frame_stack=4):
6          self.frame_stack = frame_stack
7          self.frames = deque(maxlen=frame_stack)
8
9      def preprocess(self, obs):
10         # Convert to grayscale
11         gray = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
12
13         # Crop irrelevant parts
14         gray = gray[34:194, :]
15
16         # Resize to 84x84
17         resized = cv2.resize(gray, (84, 84), interpolation=cv2.INTER_AREA)
18         _, thresholded = cv2.threshold(resized, 127, 255, cv2.THRESH_BINARY)
19
20         return thresholded
21
22     def reset(self, obs):
23         frame = self.preprocess(obs)
24         self.frames = deque([frame for _ in range(self.frame_stack)], maxlen=self.frame_stack)
25         return np.stack(self.frames, axis=0)
26
27     def step(self, obs):
28         frame = self.preprocess(obs)
29         self.frames.append(frame)
30         return np.stack(self.frames, axis=0)
```

2.10 Weights & Biases Integration

To systematically track the performance of our various DQN implementations, I employed Weights & Biases (wandb), a powerful experiment tracking tool. This integration allowed for real-time visualization of training progress, comparison between different algorithmic variants, and efficient hyperparameter tracking.

The implementation consists of three main components:

Code 10: **Implementation of Weights & Biases initialization**

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

For tracking training progress, I implemented structured logging at different timescales to capture both immediate feedback and long-term trends:

Code 11: **Periodic metric logging during training**

```
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
    })
    # Additional wandb logs for debugging
    wandb.log({
        "Memory Size": len(self.memory),
        "Beta": self.memory.beta
    })

if self.train_count % 1000 == 0:
    print(f"[Train #{self.train_count}] Loss: {loss.item():.4f} Q mean: {q_values.mean().item():.3f} std:
    ↪  {q_values.std().item():.3f}")
    wandb.log({
        "Training Loss": loss.item(),
        "Q Values Mean": q_values.mean().item(),
        "Q Values Std": q_values.std().item(),
        "Target Q Values Mean": target_q_values.mean().item()
    })
```

For evaluation and milestone tracking, I implemented as follow:

Code 12: **Evaluation and milestone logging**

```
# Check for milestone steps
for milestone in self.milestone_steps:
    if self.env_count >= milestone and milestone not in self.saved_milestones:
        model_path = os.path.join(self.save_dir, f"model_step{milestone}.pt")
        torch.save(self.q_net.state_dict(), model_path)
        print(f"[Milestone] Saved checkpoint at {milestone} environment steps to {model_path}")
        wandb.log({
            "Milestone": milestone,
            "Checkpoint Saved": True
        })
        self.saved_milestones.add(milestone)

        # Evaluate at milestone
        eval_rewards = []
        for _ in range(10):
            eval_reward = self.evaluate()
            eval_rewards.append(eval_reward)
```

```
        avg_eval_reward = sum(eval_rewards) / len(eval_rewards)
        print(f"[Milestone Eval] Step: {milestone} Avg Eval Reward: {avg_eval_reward:.2f}")
        wandb.log({
            "Milestone": milestone,
            "Milestone Eval Reward": avg_eval_reward
        })
```

In the end of episode, I also evaluate the agent performance, and logged as follow:

Code 13: **Implementation of end-of-episode logging**

```
wandb.log({
    "Episode": ep,
    "Total Reward": total_reward,
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Epsilon": self.epsilon,
})
# Additional end-of-episode logging
wandb.log({
    "Episode Length": step_count,
    "Average Reward": total_reward / max(1, step_count)
})
if ep % 20 == 0:
    eval_rewards = []
    for _ in range(10):
        eval_reward = self.evaluate()
        eval_rewards.append(eval_reward)
    avg_eval_reward = sum(eval_rewards) / len(eval_rewards)
    if avg_eval_reward > self.best_reward:
        self.best_reward = avg_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 {avg_eval_reward}")
    print(f"[TrueEval] Ep: {ep} Avg Eval Reward: {avg_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": avg_eval_reward
    })
```

3 Analysis and Discussions

3.1 Hyperparameters

The selection of appropriate hyperparameters is critical for DQN performance. Following Rainbow [2], I used Adam optimizer with learning rate of 0.0001 and ϵ = 1.5e-4 to ensure stable optimization. My complete hyperparameter configuration for the enhanced DQN is detailed below:

Table 1: Hyperparameter settings for the enhanced DQN implementation

Hyperparameter	**Value**
Optimization	
Learning rate	1e-4
Optimizer	Adam ($\epsilon = 1.5\text{e-}4$)
Batch size	32
Discount factor (γ)	0.99
Network Architecture	
Double DQN	Enabled
Dueling architecture	Enabled
Noisy networks	Enabled
Replay Buffer	
Memory size	50,000 transitions
Replay buffer type	Prioritized (PER)
PER α	0.7
PER initial β	0.4
n-step returns	3
Training Strategy	
Target network update frequency	Every 500 training steps
Replay start size	5,000 transitions
Maximum episode steps	100,000 steps
Training frequency	Every 4 environment steps
Updates per training cycle	4 updates

These hyperparameters were selected through grid search, with particular attention to the interaction between n-step returns, update frequency, and prioritized replay parameters, as these components significantly impact sample efficiency.

3.2 Training Curves

Figure 1: Training curves for CartPole-v1 environment. The x-axis represents environment steps, and the y-axis represents the average evaluation reward.

Figure 2: Training curves of vanilla DQN for Pong-v5 environment. The x-axis represents environment steps, and the y-axis represents the average evaluation reward.

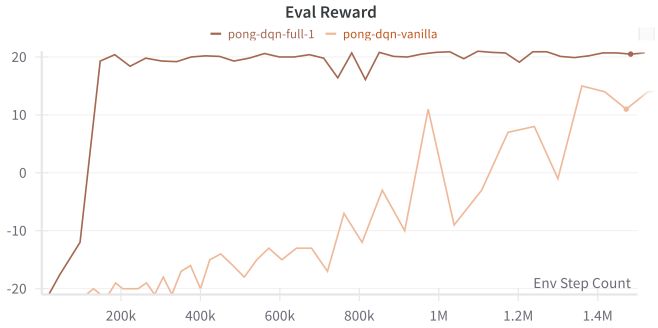

Figure 3: Training curves comparing vanilla DQN and enhanced DQN variants on Pong-v5. The enhanced DQN reaches high performance much faster.

3.3 Sample Efficiency Analysis

The enhanced DQN variants demonstrated dramatically improved sample efficiency compared to vanilla DQN. Table 2 presents a comprehensive comparison of environment steps required to reach key performance thresholds across different algorithmic configurations.

Table 2: Environment steps required to reach specific performance thresholds on Pong-v5. The full enhanced DQN integrates all improvements, while each row below shows the impact of removing a single enhancement from the full configuration.

Method	**Steps to Score 0**	**Steps to Score 10**	**Steps to Score 20**
Vanilla DQN	969,785	1,084,977	1,795,184
Full Enhanced DQN	**115,518**	**115,518**	**122,834**
w/o Huber Loss	156,116	191,845	471,005
w/o PER	241,635	418,820	599,478
w/o Multi-step	203,658	413,582	>750,000
w/o Dueling Network	144,277	180,154	249,009
w/o Noisy Network	126,703	189,507	451,812
w/o Enhanced Frame Preprocessing	186,692	208,074	422,831

The fully enhanced DQN achieves a perfect score of 20 in just 122,834 environment steps, representing a remarkable 14.6× improvement in sample efficiency over the vanilla implementation, which requires approximately 1.8 million steps to reach the same performance. Notably, the enhanced agent reaches an average score of 10 at essentially the same step count as when it reaches a score of 0, indicating extremely rapid learning once it discovers an effective strategy.

3.4 Ablation Study

To quantify the contribution of each enhancement, I conducted a comprehensive ablation study by systematically removing one component at a time from the full enhanced DQN configuration. Figures 4 and 5 visualize the comparative learning curves, while Table 2 provides precise step counts for reaching specific performance thresholds.

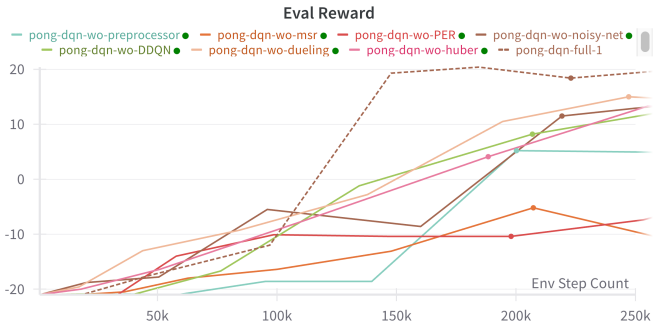

Figure 4: Evaluation reward curves comparing the full enhanced DQN against variants with single components removed. The steep curve of the full enhanced version demonstrates the synergistic effect of combining all enhancements.

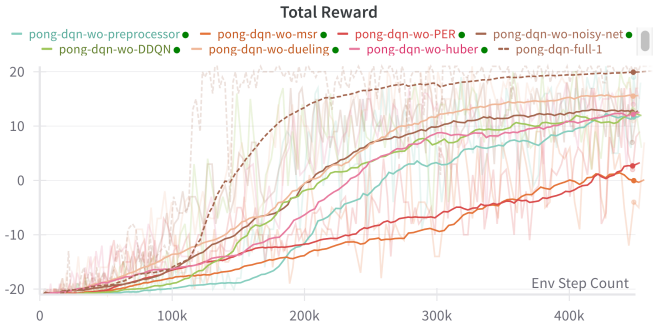

Figure 5: Total reward across training for different DQN variants. This metric captures both the learning speed and stability of each configuration, with higher area under the curve indicating more efficient learning.

The ablation results reveal specific contributions from each component:

- **Prioritized Experience Replay (PER):** Produced the most significant individual impact. The annealing of the β parameter from 0.4 to 1.0 was crucial for balancing initial learning speed with long-term stability by gradually correcting the sampling bias.
- **Multi-step Return:** The second most impactful enhancement. Using $n = 3$ provided an optimal balance between bias reduction and variance control. This approach was particularly valuable in Pong due to the delayed nature of rewards.
- **Huber Loss:** While having modest impact on early learning. This suggests its outlier resistance becomes increasingly important in later stages of policy refinement when the agent is making smaller, more precise adjustments.
- **Noisy Networks:** This technique proved particularly effective in Pong, where strategic exploration around promising ball trajectories is more valuable than random paddle movements provided by ϵ-greedy exploration.
- **Enhanced Frame Preprocessing:** Binary thresholding and proper cropping contributed substantially to efficiency. These enhancements enabled the convolutional layers to more quickly identify key game elements.
- **Dueling Network:** While showing the smallest individual impact, the dueling architecture provided consistent benefits across all training stages. Its separation of state value and action advantage estimation proved particularly effective in Pong, where understanding good positions (state value) can be learned somewhat independently from optimal paddle movements (action advantage).

The most striking finding from this ablation study is that the combined effect of all enhancements far exceeds what would be expected from their individual contributions. This synergistic interaction occurs because each enhancement addresses a different limitation of the base DQN algorithm:

1. PER focuses computation on valuable experiences
2. Multi-step returns address the delayed reward problem
3. Huber loss improves optimization stability
4. Noisy networks provide adaptive exploration
5. Enhanced preprocessing simplifies the feature learning challenge
6. Dueling architecture decouples state value and action advantage learning

When combined, these enhancements create a virtuous cycle where improvements in one aspect amplify the benefits of others. For example, better exploration from noisy networks discovers more informative experiences, which PER then prioritizes for learning, while multi-step returns help propagate the resulting value information more effectively through time. This mutually reinforcing system explains why the full enhanced DQN achieves such dramatic performance improvements over vanilla DQN.

4 Additional Analysis

4.1 Training Instability in DQN

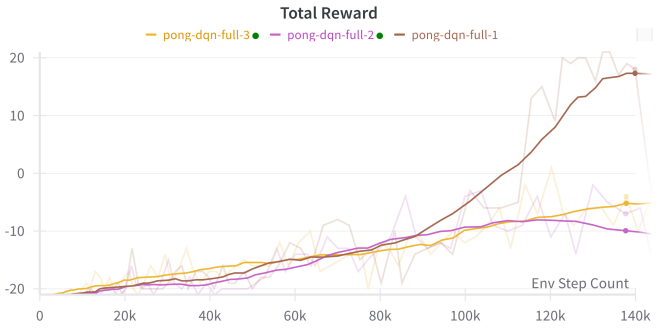

Figure 6: The instability issue in the DQN training process.

Throughout the training process, I observed that DQN exhibits significant training instability even when using identical hyperparameter configurations. As shown in Figure 6, three training runs with the same hyperparameters can yield remarkably different learning curves and final performance.

This instability can be attributed to several factors:

1. **Initialization sensitivity**: Small differences in network weight initialization can be amplified through the bootstrapping process in Q-learning, where current value estimates depend on previous estimates.
2. **Experience replay sampling variance**: The stochastic nature of experience replay sampling introduces variability between runs. Even with prioritized replay, the subset of experiences used for each update can significantly impact gradient direction and magnitude.
3. **Environment stochasticity**: In environments with inherent randomness (like starting positions in Pong), different trajectories can lead to entirely different learned strategies.
4. **Non-stationarity**: The Q-learning target is non-stationary, as it depends on the current network's estimates, creating a "moving target" problem that can lead to oscillations during training.

This observation highlights the importance of running multiple seeds when evaluating DQN variants and suggests that single-run comparisons between algorithms may lead to misleading conclusions.

Nonetheless, I think the ablation studies still show the effectiveness of each components.

4.2 Emergent Exploitative Strategy in Pong

I analyze the best-performing model, which achieved an average score of 19 within just 140K environment steps, revealed an interesting phenomenon. The agent discovered an exploitative strategy that was remarkably effective against the built-in AI opponent.

The agent learned to position its paddle at a specific vertical position where the opponent's algorithm consistently failed to return the ball. This position appears to exploit a blind spot in the opponent's control algorithm, causing it to misjudge the trajectory of the ball and miss consistently.

5 Conclusion

In this lab, I successfully implemented and enhanced the Deep Q-Network algorithm for reinforcement learning in both CartPole and Pong environments. The vanilla DQN implementation achieved strong performance in both tasks, while the enhanced DQN demonstrated dramatically improved sample efficiency, particularly in the more complex Pong environment.

The ablation study revealed that while each enhancement contributed to performance improvements, the combination of all techniques yielded synergistic benefits that exceeded the sum of individual contributions. This highlights the importance of a holistic approach to algorithm design in deep reinforcement learning.

Future work could explore additional enhancements such as distributional RL approaches (C51, QR-DQN) and curiosity-driven exploration. Integration with more complex environments like 3D games would also be an interesting extension of this work.

References

[1] Meire Fortunato et al. "Noisy Networks for Exploration". In: *International Conference on Learning Representations (ICLR)*. 2018. arXiv: 1706.10295 [cs.LG]. URL: https://openreview.net/forum?id=rywHCPkAW.

[2] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018. DOI: 10.1609/aaai.v32i1.11796. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11796.

[3] Volodymyr Mnih et al. "Human-level Control through Deep Reinforcement Learning". In: *Nature* 518.7540 (Feb. 26, 2015), pp. 529–533. DOI: 10.1038/nature14236.

[4] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[5] Tom Schaul et al. "Prioritized Experience Replay". In: *International Conference on Learning Representations (ICLR)*. 2016. arXiv: 1511.05952 [cs.LG]. URL: https://arxiv.org/abs/1511.05952.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, MA: MIT Press, Nov. 13, 2018. ISBN: 9780262039246. URL: http://incompleteideas.net/book/the-book-2nd.html.

[7] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016, pp. 2094–2100. DOI: 10.1609/aaai.v30i1.10295.

[8] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. Vol. 48. 2016, pp. 1995–2003. DOI: 10.48550/arXiv.1511.06581. URL: https://proceedings.mlr.press/v48/wang16f.html.