Lab 05: Value-Based Reinforcement Learning

112550127 Pin-Kuan Chiang

August 27, 2025

1 Introduction

This lab focused on learning the Deep Q-Network (DQN) algorithm and several of its key enhancements, including Double DQN, Prioritized Experience Replay, and Multi-step Returns. In addition, we explored the impact of different hyperparameter settings by conducting ablation studies on these enhancements. Specifically, we examined various ϵ-scheduling strategies and compared multiple values of n in the multi-step return. Through these experiments, we found that the combination of the baseline exponential ϵ-decay and n-step returns with $n = 5$ achieved the best overall performance.

2 Implementation

2.1 Bellman Error for DQN (Tasks 1–3)

High-level idea. Across Tasks 1–3, the Bellman error is the TD mismatch between the online network's estimate $Q_\theta(s, a)$ and a bootstrapped target y. We sample $(s, a, r, s', done)$, compute $Q_\theta(s, a)$, build y with a target network (and, in Double DQN, decouple action selection and evaluation), then minimize the loss between $Q_\theta(s, a)$ and y. Terminal transitions remove the bootstrap term via $(1 - \texttt{done})$.

Task 1 (Vanilla DQN). Use the target network to compute the next-state maximum and form the TD target:

$$y = r + (1 - done)\,\gamma \max_{a'} Q_{\theta^-}(s', a').$$

The loss function (`self.criterion`) is **Smooth L1 Loss** (Huber loss), which is less sensitive to outliers than pure MSE.

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

q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
loss = self.criterion(q_values, expected_q_values.detach())

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Task 2 (Concept—toward DDQN). The goal is to reduce overestimation by using the *online* network to pick the next action and the *target* network to evaluate it. The same **Smooth L1 Loss** is used to minimize the TD error.

Task 3 (Double DQN Implementation). Select the greedy next action with the online net, then evaluate that action with the target net:

$$y = r + (1 - done)\,\gamma\, Q_{\theta^-}\big(s', \arg\max_a Q_\theta(s', a)\big).$$

The loss criterion remains the **Smooth L1 Loss**, computed between $Q_\theta(s, a)$ and the detached TD target.

```
next_actions = self.q_net(next_states).argmax(1)
target_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
expected_q_values = rewards + (1 - dones) * self.gamma * target_q_values
```

2.2 Double DQN Modification

```
with torch.no_grad():
    if self.args.DDQN:
        # DDQN: use the online network to select the action
        next_actions = self.q_net(next_states).argmax(1)  # action selection
        # evaluate the selected action using the target network
        target_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).
            squeeze(1)
    else:
        # Vanilla DQN: directly select and evaluate using the target network
        target_q_values = self.target_net(next_states).max(1)[0]
```

Double DQN reduces overestimation by *decoupling* action selection and evaluation for the bootstrap target. If `DDQN` is enabled, the *online* network selects the next action ($\arg\max_a Q_\theta(s', a)$), and the *target* network evaluates that action $Q_{\theta^-}(s', a^\star)$. Otherwise (vanilla DQN), the target network both selects and evaluates via $\max_{a'} Q_{\theta^-}(s', a')$.

2.3 PER Implementation

Prioritized Experience Replay (PER) improves sample efficiency by biasing the replay buffer toward transitions with higher learning potential, measured by the magnitude of their TD errors. Instead of uniform sampling, each transition i is drawn with probability

$$P(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}, \quad p_i = |\delta_i| + \epsilon,$$

where α controls prioritization strength and ϵ prevents zero probability. To correct for the bias introduced by non-uniform sampling, importance-sampling (IS) weights

$$w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta$$

are applied to each sample's loss, with β annealed toward 1 during training.

```
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta=0.4, eps=1e-6):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.eps = eps
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.pos = 0
        self.max_priority = 1.0

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

    def add(self, transition, error=None):
        if error is None:
            priority = self.max_priority
        else:
            priority = (abs(error) + self.eps) ** self.alpha

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

        self.priorities[self.pos] = priority
        self.max_priority = max(self.max_priority, priority)
```

```
        self.pos = (self.pos + 1) % self.capacity

    def sample(self, batch_size):
        current_len = len(self.buffer)
        prios = self.priorities[:current_len]

        prios = np.maximum(prios, self.eps)
        prob_sum = prios.sum()
        if not np.isfinite(prob_sum) or prob_sum <= 0:
            probs = np.ones(current_len, dtype=np.float32) / current_len
        else:
            probs = prios / prob_sum

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

        # IS weights
        weights = (current_len * probs[indices]) ** (-self.beta)
        weights = weights / weights.max()

        states, actions, rewards, next_states, dones = zip(*samples)
        return (states, actions, rewards, next_states, dones, indices, weights)

    def update_priorities(self, indices, errors):
        for idx, e in zip(indices, errors):
            p = (abs(e) + self.eps) ** self.alpha
            self.priorities[idx] = p
            self.max_priority = max(self.max_priority, p)
```

This buffer stores transitions and a priority for each one. Sampling is proportional to priority$^{\alpha}$ (with ϵ to avoid zeros), and importance-sampling (IS) weights with exponent β are returned to correct bias. After learning, priorities are updated using the latest TD errors.

```
if self.args.PER:
    t = min(1.0, self.train_count / self.per_beta_frames)
    self.memory.beta = self.per_beta_start + t * (1.0 - self.per_beta_start)
per_sample_loss = self.criterion(q_values, expected_q_values)
```

When PER is enabled, β is annealed from a start value toward 1.0 over training steps to gradually reduce bias. The criterion (Smooth L1 / Huber) returns a per-sample loss that will be reweighted by the IS weights.

```
if self.args.PER:
    weights_tensor = torch.tensor(weights, dtype=torch.float32, device=self.device)
    loss = (per_sample_loss * weights_tensor).mean()
    self.memory.update_priorities(indices, td_errors.detach().abs().cpu().numpy())
else:
    loss = per_sample_loss.mean()
```

With PER, each sample's loss is multiplied by its IS weight before averaging, yielding an unbiased (or less biased) gradient estimate. The buffer's priorities are then updated using the absolute TD errors; otherwise, we fall back to a uniform mean loss.

2.4 Multi-step Return Implementation

Multi-step return extends the 1-step TD target by incorporating multiple future rewards before bootstrapping from the n-step future state. For an n-step return, the target is:

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

which can speed up learning by providing richer target signals and balancing bias-variance trade-offs.

```
if self.args.MSR:
    nstep_buffer = deque(maxlen=self.n_step)
```

An n-step buffer temporarily stores recent transitions until it contains n elements, enabling computation of the n-step return.

```
if self.args.MSR:
    nstep_buffer.append((state, action, reward, next_state, done))

if self.args.MSR and len(nstep_buffer) == self.n_step:
    R = sum([nstep_buffer[i][2] * (self.gamma ** i) for i in range(self.n_step)])
    s, a, _, _, _ = nstep_buffer[0]
    ns, _, _, _, d = nstep_buffer[-1]
    self._push_transition(s, a, R, ns, d)
elif not self.args.MSR:
    self._push_transition(state, action, reward, next_state, done)
```

Once the buffer reaches length n, the n-step discounted reward R is computed, and a single aggregated transition is pushed to memory. If MSR is disabled, standard 1-step transitions are pushed instead.

```
if self.args.MSR:
    while len(nstep_buffer) > 0:
        R = sum((self.gamma ** i) * nstep_buffer[i][2] for i in range(len(nstep_buffer)))
        s, a = nstep_buffer[0][0], nstep_buffer[0][1]
        ns, d = nstep_buffer[-1][3], nstep_buffer[-1][4]
        self._push_transition(s, a, R, ns, d)
        nstep_buffer.popleft()
```

At episode end, remaining transitions in the buffer are processed to ensure all rewards contribute to training, even if fewer than n steps remain.

2.5 Weights & Biases Logging

```
wandb.log({
    "Train Loss": loss.item(),
    "Q Mean": q_values.mean().item(),
    "Q Std": q_values.std().item()
})
```

Logs the current training loss and Q-value statistics (mean and standard deviation) for each training step. This helps monitor learning stability and detect divergence.

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

Logs evaluation metrics per episode, including total reward, total environment steps, update count, and current exploration rate ϵ, allowing progress tracking over time.

3 Training Curves

The training performance for each task is visualized below, showing the evaluation score as a function of environment steps. These curves illustrate how different algorithmic enhancements influence sample efficiency and final performance.

Figure 1: Task 1: Evaluation score vs. environment steps

In Task 1, the agent quickly reaches a stable score of 500, demonstrating high sample efficiency and fast convergence.

Table 1: Task 1 (CartPole) Hyperparameters

Parameter	Default Value	Description
batch-size	64	Batch size
memory-size	50000	Replay buffer size
lr	0.001	Learning rate
discount-factor	0.99	Discount factor γ
epsilon-start	1.0	Initial ϵ
epsilon-decay	0.995	Epsilon decay rate
epsilon-min	0.05	Minimum ϵ
target-update-frequency	500	Target network update freq.
replay-start-size	2000	Min buffer size before training
max-episode-steps	10000	Max steps per episode
train-per-step	1	Training steps per env step

Figure 2: Task 2: Evaluation score vs. environment steps

In Task 2, the training process is significantly longer. Even after the ϵ value decays to 0.01, the agent continues training for an additional 2 million steps before stabilizing.

Table 2: Task 2 (Pong baseline) Hyperparameters

Parameter	Default Value	Description
batch-size	32	Batch size
memory-size	100000	Replay buffer size
lr	0.0000625	Learning rate
discount-factor	0.99	Discount factor γ
epsilon-start	1.0	Initial ϵ
epsilon-decay-step	0.9999977	Steps Epsilon decay to zero
epsilon-min	0.03	Minimum ϵ
target-update-frequency	10000	Target network update freq.
replay-start-size	50000	Min buffer size before training
max-episode-steps	10000	Max steps per episode
train-per-step	2	Training steps per env step

Figure 3: Task 3: Evaluation score vs. environment steps

In Task 3, the enhancement proves to be highly successful. After the ϵ decay is completed, the agent exhibits a rapid performance climb, quickly approaching its maximum score.

Table 3: Task 3 (Pong + enhancements) Hyperparameters

Parameter	Default Value	Description
batch-size	32	Batch size
memory-size	50000	Replay buffer size
lr	0.0000625	Learning rate
discount-factor	0.99	Discount factor γ
epsilon-start	1.0	Initial ϵ
epsilon-min	0.01	Minimum ϵ
epsilon-decay-steps	850000	Decay steps to epsilon-min
target-update-frequency	1000	Target network update freq.
replay-start-size	50000	Min buffer size before training
max-episode-steps	10000	Max steps per episode
train-per-step	1	Training steps per env step
DDQN	True	Enable Double DQN
PER	True	Enable Prioritized Replay
MSR	True	Enable Multi-Step Return
n-step	5	Steps for Multi-Step Return

3.1 Sample Efficiency

Table 4 compares the number of environment steps required for each algorithm to reach specific evaluation scores. Lower values indicate higher sample efficiency, meaning the agent learns faster with fewer interactions.

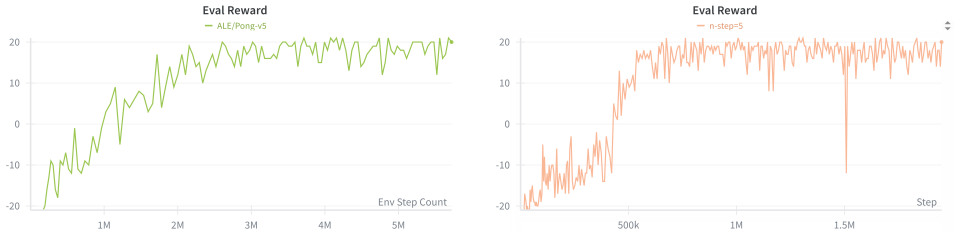

Figure 4: DQN vs exDQN

Table 4: Environment steps required to reach given evaluation scores

Score	Vanilla DQN	Enhanced DQN
-9	276,539	102,674
0	1,023,996	405,122
9	1,150,674	495,942
19	2,155,267	704,959

From the results, it is evident that the Enhanced DQN achieves the same evaluation scores with significantly fewer environment steps across all benchmarks, indicating a substantial improvement in sample efficiency.

3.2 Ablation Study

Figure 5: Total Reward without specific enhancements

Observations:

- **w/o DDQN:** The overall total reward increases slowly and remains relatively low, although the variance is small. This suggests that DDQN significantly improves training speed.
- **w/o PER:** The total reward curve shows smaller fluctuations, indicating that PER contributes to improved training stability.
- **w/o MSR:** The curve exhibits larger fluctuations compared to the baseline, suggesting that MSR helps reduce variance during training.

Overall, the relatively poor learning performance observed in some ablation settings may be due to the absence of a critical enhancement, which will be discussed in detail in the next section.

4 Additional Analysis on Other Training Strategies

4.1 Additional Preprocessing for Atari Frames

In the Atari environment, the raw observation includes a scoreboard at the top of the frame. Since the agent's policy should ideally remain unaffected by the instantaneous score (as the optimal action depends on the game state, not the score display), this region was cropped out during preprocessing. Specifically, the grayscale image was cropped from pixel row 34 to 194, effectively removing the scoreboard and leaving only the gameplay area. This helps prevent the model from overfitting to score-related visual patterns that are not causally related to optimal decision-making.

In addition to cropping, we applied a *max-over-two-frames* operation, taking the element-wise maximum between the current frame and the previous frame. This step is a common Atari preprocessing technique that mitigates issues caused by flickering sprites and intermittent object visibility, ensuring that transient visual artifacts do not degrade the agent's perception of the environment. By combining scoreboard removal and max-frame processing, the agent receives cleaner and more stable inputs, potentially improving both training stability and final performance.

It is worth noting that this preprocessing step corresponds to the **critical enhancement** mentioned before, whose absence in the ablation experiments resulted in noticeably poorer learning performance. This highlights its significant impact on the agent's ability to learn efficiently and stably.

4.2 Epsilon decay strategy

In addition to the main experiments, we conducted an analysis on the impact of different ϵ-decay strategies in the ϵ-greedy policy. Three decay types were tested: **Cosine Decay**, **Linear Decay**, and **Baseline (Exponential Decay)**.

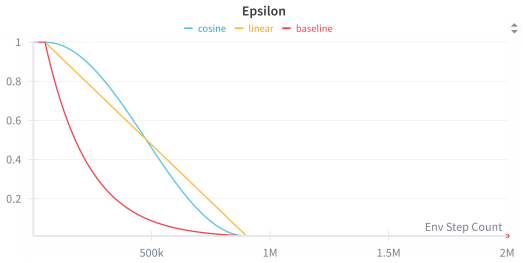

Figure 6: Epsilon decay for three strategies

Figure 7: Training reward curves for Linear Decay (left), Cosine Decay (middle), and Baseline (Exponential Decay, right)

Observations:

- **Linear Decay:** This strategy yielded the weakest performance. Since ϵ decreased too quickly (from 0.1 down to 0.01), the agent lacked sufficient exploration during the mid-training phase. As a result, it failed to consistently discover and refine better strategies.
- **Cosine Decay:** Performance was comparable to the baseline, showing smooth improvements. The gradual oscillatory reduction of ϵ provided a balanced trade-off between exploration and exploitation.

- **Baseline (Exponential Decay):** Similar to cosine decay, this method maintained stable learning dynamics. Due to its distribution, however, the baseline decay allowed the agent to reach high-reward strategies slightly faster, leading to quicker convergence.

In summary, linear decay underperformed due to overly aggressive reduction in exploration, while cosine and exponential decay achieved similar results, with the exponential baseline learning marginally faster.

4.3 Multi-step return analysis

We further investigated the effect of varying the number of steps n in the multi-step return. Three configurations were tested: $n = 3$, $n = 4$, and $n = 5$.

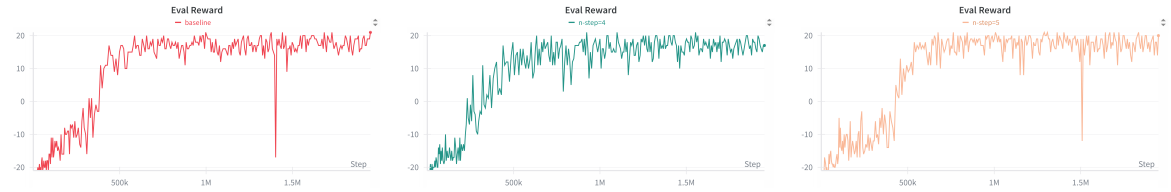

Figure 8: Evaluation reward of different n-step settings ($n = 3, 4, 5$).

Observations:

- $n = 3$**:** Produced stable learning curves, but the improvement in evaluation reward was relatively gradual, without noticeable jumps in performance.
- $n = 4$**:** Similar overall trend to $n = 3$, though occasional jumps in reward were observed. However, the curve still exhibited moderate fluctuations after these increases.
- $n = 5$**:** Displayed the clearest "jump" in evaluation reward, with a sharp increase followed by a more stable high-reward plateau. Around 1M steps, the agent achieved an average score of 19, which was the best among the tested settings.

In summary, although the overall performance differences between $n = 3$, $n = 4$, and $n = 5$ were not dramatic, larger n values tended to produce more distinct jumps in performance. Given its stability after improvement and the best evaluation score, we selected $n = 5$ for the final snapshot.

5 Screenshot of Evaluation

Figure 9 presents a screenshot of the final evaluation result. In this run, the agent successfully reached a score of 19 at step 1,005,843 (epoch 555) using seed 126. This confirms the effectiveness of the chosen configuration in achieving stable performance.

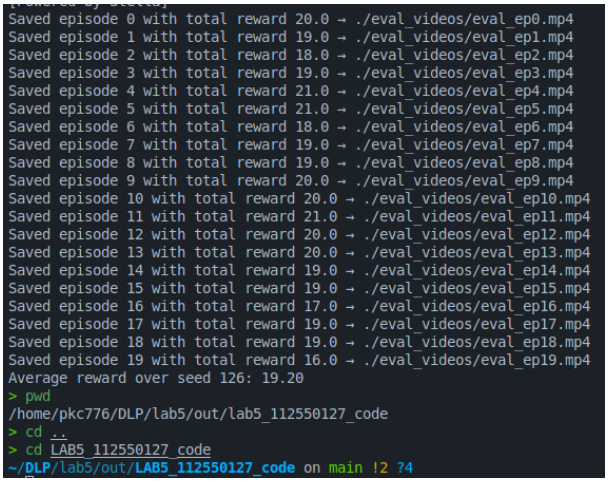

Figure 9: Screenshot of the evaluation interface at the checkpoint reaching an average score of -19 (global step 1,005,843, epoch 555, seed 126).