Lab 5 - Report

B1126006 郭兆揚

Introduction

This report aims to explore the application and performance enhancement of a value-based reinforcement learning method—Deep Q-Network (DQN)—under different input dimensionalities. I implemented and evaluated various DQN variants across three tasks: Task 1 and Task 2 focus on vanilla DQN, applied to low-dimensional numerical input (CartPole) and high-dimensional visual input (Atari Pong), respectively. Task 3 integrates three enhancement techniques—Prioritized Experience Replay (PER), Double DQN, and Multi-Step Return—and compares their learning efficiency against the vanilla version.

In Task 1 and Task 2, I was able to achieve good results by empirically tuning the hyperparameters. However, in Task 3, despite repeated experiments, I could not significantly improve the model's performance. After several unsuccessful attempts, I switched to using the hyperparameter settings suggested in the Rainbow paper as a baseline and made further adjustments based on them. Eventually, I found that while these enhancement techniques do improve performance, they also make the model far more sensitive to hyperparameter choices—so much so that without a well-tuned baseline, it becomes extremely difficult to achieve good results.

Implementation

In **Task 1**, considering the simplicity of the CartPole environment, where the input state consists of only four dimensions, I used a simple fully connected neural network as the value function approximator to reduce the risk of overfitting.

For the replay buffer, I implemented it using Python’s deque structure, which naturally fits the requirement of discarding the oldest transitions when the buffer reaches its maximum capacity. Uniform sampling was performed using random.sample, which allows for randomly selecting a fixed-size batch from the buffer.

```
self.memory = deque(maxlen=args.memory_size)
```

```
random.sample(self.memory,self.batch_size)
```

In terms of loss computation, the objective of DQN is to minimize the Bellman error—that is, to reduce the difference between the Q-values estimated by the online network and the target network. To achieve this, we need to compute both values explicitly. The Q-values from the online network are obtained by passing the sampled states through the network and extracting the values corresponding to the sampled actions. This represents the expected return when taking a specific action in a given state, as predicted by the current policy.

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

As for the Q-values from the target network, since this task implements vanilla DQN, the target Q-value (i.e., the TD target) is computed by simply adding the immediate reward to the discounted maximum Q-value of the next state, as predicted by the target network. This value serves as the update target in the Bellman equation.

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

Finally, the loss is computed as the mean squared error between the Q-value predicted by the online network and the TD target derived from the target network.

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

In **Task 2**, the main difference from Task 1 lies in the architecture of the neural network. Since the input state in this task is an image rather than a low-dimensional vector, the fully connected network used in Task 1 was replaced with a convolutional neural network (CNN). Following the principle of simplicity and practical experience, I first adopted the architecture provided in test_model.py by the teaching assistant, which consists of three convolutional layers followed by two fully connected layers. After further tuning the hyperparameters and conducting experiments, the model achieved satisfactory performance.

In **Task 3**, I implemented three enhancement techniques: Multi-Step Return, Double DQN, and Prioritized Experience Replay (PER).

Starting with Multi-Step Return, this method can be viewed as a form of “lookahead,” where the actual rewards from the next n steps are used to partially replace the estimated Q-values from the network. By incorporating longer-term actual

rewards, this approach reduces the dependency on bootstrapped Q-values, thereby mitigating estimation bias. In the implementation, I used Python's deque to construct the return buffer, which naturally fits the requirement of storing a sequence of recent transitions over n steps.

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

Subsequently, the `_get_n_step_info` method is used to extract the accumulated n-step reward and the corresponding next state from the buffer. These values are then used to construct a new transition, which is stored in the replay buffer.

```
R += (self.gamma ** idx) * r

next_state, done = self.n_step_buffer[-1][3], self.n_step_buffer[-1][4]
```

In the case of Double DQN (DDQN), this technique was proposed to address the overestimation bias commonly observed in standard DQN. The core issue arises from the use of the max operator in Q-learning, which tends to overestimate action values due to the presence of noise:

$$\mathbb{E}\left(\max_i(Q_i)\right) \geq \max_i(x_i)$$

To mitigate this, Double DQN decouples the action selection and evaluation steps. Specifically, the action is first selected using the **online network**, and then its Q-value is evaluated using the **target network**. This reduces the upward bias by avoiding using the same network to both select and evaluate the action. In implementation, the online network is first used to select the action with the highest estimated Q-value for the next state. This action is then passed to the target network to obtain the corresponding Q-value.

```
next_q_values = self.q_net(next_states)

next_actions = next_q_values.argmax(1).unsqueeze(1)
```

The target network then uses the action selected by the online network to compute the corresponding Q-value for the next state.

```
next_q_values_target = self.target_net(next_states)

next_q_target = next_q_values_target.gather(1, next_actions).squeeze(1)
```

Finally, the TD target is computed by combining the immediate multi-step return with the discounted Q-value of the selected action from the target network.

```
target_q = rewards + (self.gamma ** self.n_step) * (1 - dones) * next_q_target
```

For Prioritized Experience Replay (PER), the key idea is to perform non-uniform sampling based on a priority mechanism, where transitions with higher learning potential—typically those with larger Bellman errors—are sampled more frequently. This allows the agent to focus on more informative or challenging transitions, thereby improving sample efficiency. In implementation, two separate buffers are maintained: one to store the transitions, and the other to store their corresponding priorities. Each priority value is associated with the same index as its corresponding transition in the buffer.

```
self.buffer = []
```

```
self.priorities = np.zeros((capacity,), dtype=np.float32)
```

In PER, three key functions must be implemented: add, sample, and update_priorities. The add function is responsible for inserting new transitions into both the transition buffer and the priority buffer. The priority is typically computed based on the TD error (i.e., the Bellman error). To ensure that newly added transitions have a chance to be sampled, they are assigned the current maximum priority by default.

```
priority = (abs(error) + 1e-7) ** self.alpha

self.buffer[self.pos] = transition

self.priorities[self.pos] = priority

max_prio = self.memory.priorities.max() if len(self.memory) > 0 else 1.0

self.memory.add((n_state, n_action, n_reward, n_next_state, n_done), error=max_prio)
```

In the sample function, the stored priorities are first normalized into a probability distribution. Sampling is then performed using np.random.choice, where transitions with higher priority are more likely to be selected.

```
sampling_probs = priorities / np.sum(priorities)
```

```
indices = np.random.choice(len(self.buffer), batch_size, p=sampling_probs, replace=True)

samples = [self.buffer[i] for i in indices]
```

Since the sampling process in PER is non-uniform, a correction factor known as importance-sampling weights is applied to adjust for the introduced bias. At the beginning of training, larger bias is tolerated to allow more frequent reuse of informative samples. As training progresses, the correction becomes stronger to ensure the stability and fairness of learning.

```
weights = (len(self.buffer) * sampling_probs [indices]) ** (-beta)

fraction = min(1.0, self.train_count / self.beta_anneal_steps)

beta = self.beta_start + fraction * (self.beta_end - self.beta_start)

samples, indices, weights = self.memory.sample(self.batch_size, beta)
```

The update_priorities function is responsible for updating the priority values in the buffer. After training, the TD errors computed from the current batch are used to update the corresponding priorities of the sampled transitions, so that the sampling distribution reflects the latest learning progress.

```
td_errors = torch.abs(target_q.detach() - q_values.detach())

self.memory.update_priorities(indices, td_errors)

self.priorities[i] = (abs(e) + 1e-7) ** self.alpha
```

Regarding Weights & Biases logging, I only added additional logging features for Task 3 due to its higher implementation complexity. For the other tasks, the logging setup follows the default configuration provided by the teaching assistant.

```
wandb.log({

    "Train/Loss": loss.item(), "Train/Epsilon": self.epsilon,

    "Train/Beta": beta,"Train/Q_mean": q_values.mean().item(),

    "Train/Q_std": q_values.std().item()

})

current_lr = self.optimizer.param_groups[0]['lr']
```

```
wandb.log({"LR": current_lr})
```

As an additional training strategy, I incorporated dynamic learning rate adjustment using StepLR, based on experimental observations. This scheduler reduces the learning rate by a fixed factor at predefined step intervals to stabilize training in the later stages.

```
StepLR(self.optimizer, step_size=args.scheduler_step_size, gamma=args.scheduler_gamma)
```

In addition, I modified the evaluation procedure to perform multiple inference runs per evaluation and report the averaged score. This approach yields more accurate and stable results by reducing the variance introduced by single-run evaluations.

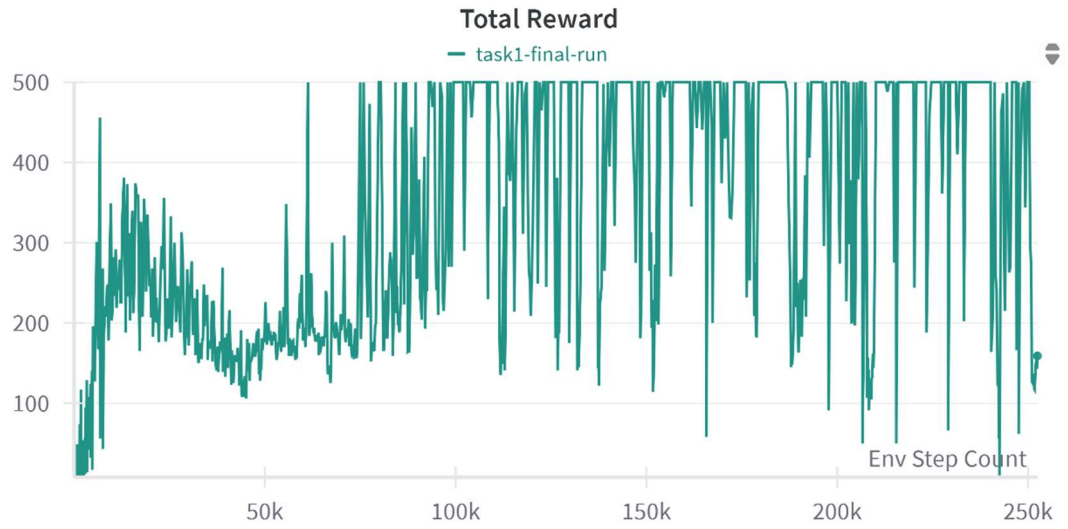
```
def evaluate(self, num_episodes=5)
```

I also implemented a **reward shaping mechanism**, aiming to amplify the reward signal in order to make the model more sensitive to undesirable actions. The intuition was that stronger rewards and penalties would allow the agent to better distinguish between good and bad behaviors. However, experimental results showed that this modification had a **negative effect on training stability**, and in some cases even degraded the agent's performance. As a result, I chose **not to include this mechanism** in the final version of the model.

```
if reward == 1:
    reward = 1.5
elif reward == -1:
    reward = -2.0
```

Analysis & Discussion

Task 1



total episodic rewards versus environment steps of task 1

evaluation score versus environment steps of task 1

The two figures below show the performance of the vanilla DQN on the CartPole-v1 environment. The upper plot displays the **total reward during training**, measured as the episodic reward collected by the agent while performing both

exploration and exploitation. The lower plot shows the **evaluation reward**, measured after disabling exploration (i.e., with a greedy policy).

The hyperparameters used in this experiment are as follows:

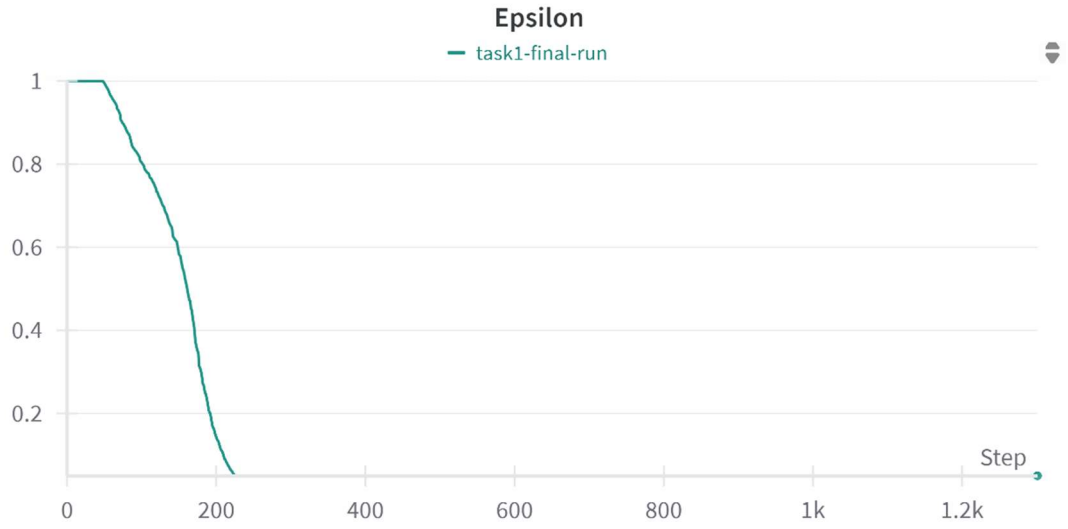
```
--lr 0.0006 ^
--epsilon-decay 0.9998 ^
--target-update-frequency 300 ^
--replay-start-size 1000 ^
--max-episode-steps 500
```

Note: All other parameters follow the default settings provided by the teaching assistant.

In this experiment, a relatively large learning rate and a frequent target network update frequency were used to accelerate convergence. Additionally, a smaller replay start size and a higher epsilon decay were adopted to better suit the CartPole environment. This is because the default settings—originally designed for Pong—tended to result in excessive exploration in CartPole, such that even after exhausting the episode budget, the agent was still predominantly exploring. This led to an imbalance between exploration and exploitation.

With the adjusted configuration, the agent was able to consistently achieve the maximum reward of 500. However, as shown in the training curves, both the total reward and evaluation reward exhibited fluctuations even during the later stages, where the agent was already acting under a purely greedy (exploitation) policy. I believe the instability may stem from the following reasons:

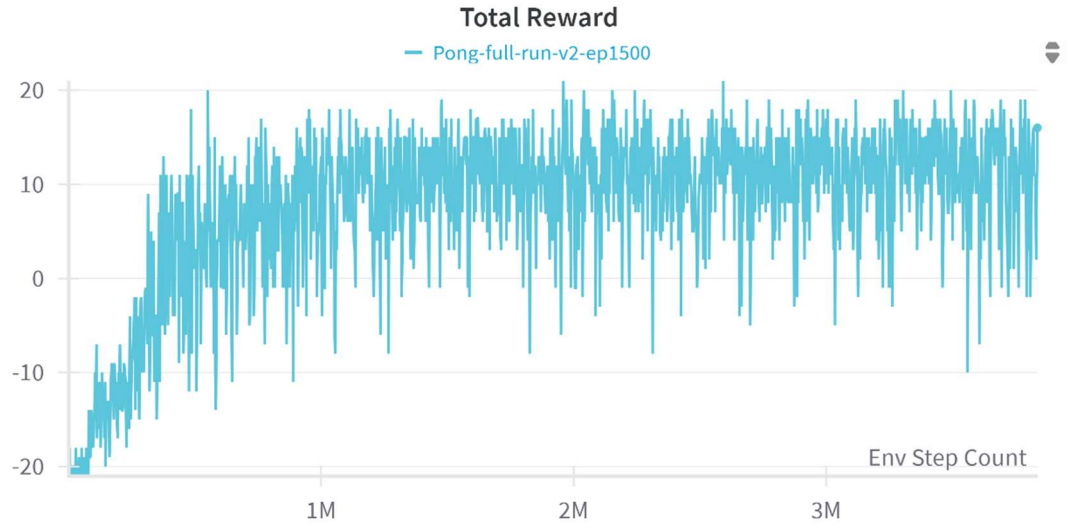
1. The target network was updated too frequently, potentially causing instability in the Q-value estimates.
2. The default replay buffer size may be too large for CartPole, causing the buffer to retain many early, highly exploratory transitions that no longer reflect the agent's current policy.



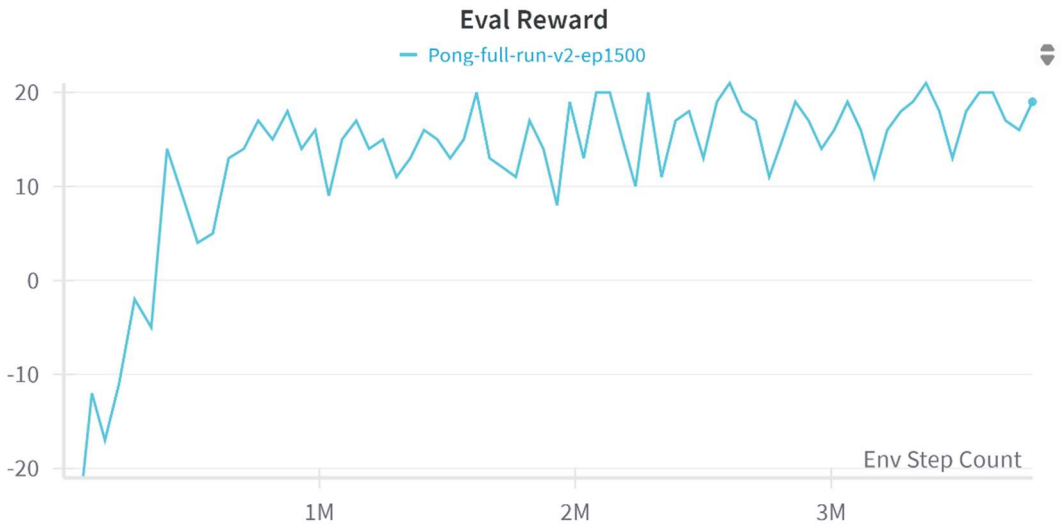
Epsilon value of epsilon greedy of task 1

In summary, although the model exhibited some instability during training, the simplicity of the CartPole environment and its ease of generalization allowed the agent to still achieve consistent maximum performance. Moreover, since the training script was designed to save the best-performing model, the final result was able to reliably reach the perfect score of 500.

Task 2



total episodic rewards versus environment steps of task 2



evaluation score versus environment steps of task 2

The two figures below show the performance of the vanilla DQN on the ALE/Pong-v5 environment. The hyperparameters used in this experiment are as follows:

```
--lr 0.0002 ^
```

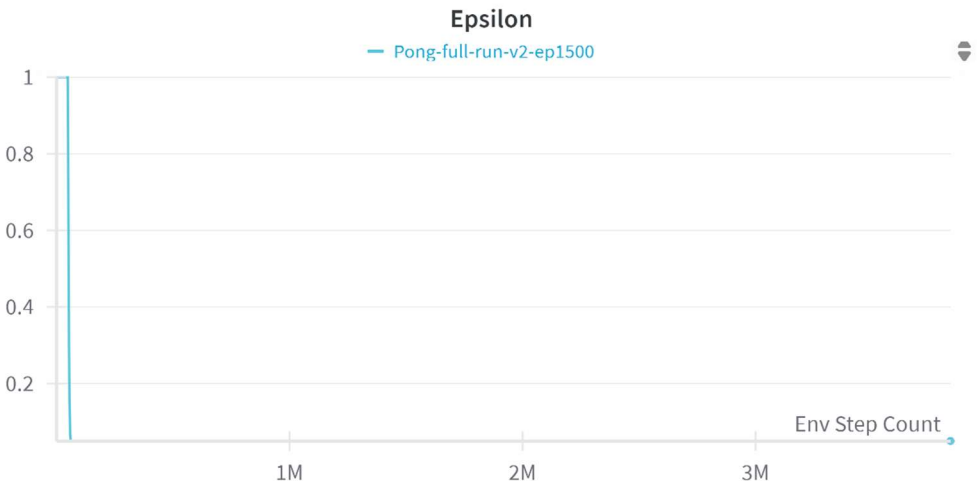
```
--epsilon-decay 0.9997
```

All other settings follow the default values provided by the teaching assistant.

Compared to Task 1, the number of environment steps required in this task is significantly higher, highlighting the greater complexity of the Pong environment relative to CartPole. To improve learning efficiency under this more challenging setup, I made the following adjustments:

1. **Increased learning rate**: Inspired by the learning rate of 0.00025 suggested in the original DQN paper, I adjusted it slightly through experimentation. A higher learning rate was found to improve convergence speed and reduce the likelihood of the agent getting stuck in poor local optima.
2. **Decreased epsilon decay rate**: While the default value of 0.999999 ensures thorough exploration, it also leads to extremely slow training progress. Given the limited computational resources available, I decided to accelerate the

transition from exploration to exploitation to improve training efficiency, accepting a trade-off in exploratory coverage.



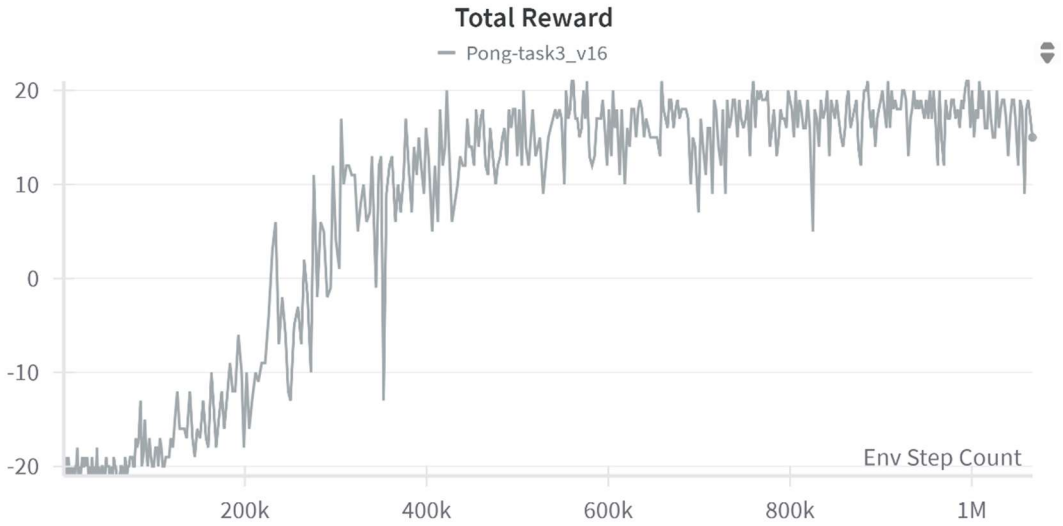
Epsilon value of epsilon greedy of task 2

3. **Episodes**: After experimentation, I extended the number of training episodes from 1000 to 1500 to ensure that this particular set of trade-off hyperparameters could reach its full potential.

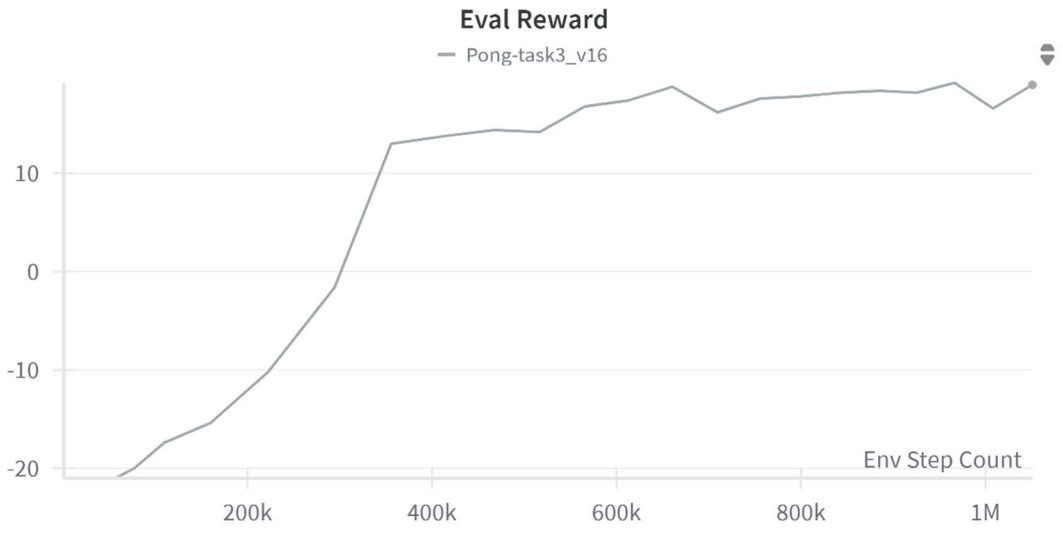
As for the results, this configuration was able to reach scores between 17 and 19 within 4 hours of training, which is already close to the full score threshold defined in the grading rubric. The learning curve shows relatively fast improvement in the early stage (reaching 16–18 points around 1 million environment steps), but the progress slows down in the later phase. I believe this is due to the reduced amount of exploration—the agent has likely already extracted most of the useful information from the buffer, leading to limited gains afterward.

However, since a minimum epsilon value was retained, the agent continued to explore to some extent, which eventually pushed the score higher. That said, this residual exploration also contributed to the fluctuations observed in the total reward curve, as the agent occasionally performed random actions during evaluation.

Task 3



total episodic rewards versus environment steps of task 3



evaluation score versus environment steps of task 3

The figure above shows the final result of my Enhanced DQN implementation for Task 3 on the ALE/Pong-v5 environment. Compared to the vanilla DQN, the enhanced DQN achieves higher scores more quickly and with greater stability, using significantly fewer environment steps. I attribute this improvement in sample efficiency and performance to the following enhancements:

1. **Prioritized Experience Replay (PER):**

   PER allows the agent to focus on high-value transitions during the early stages

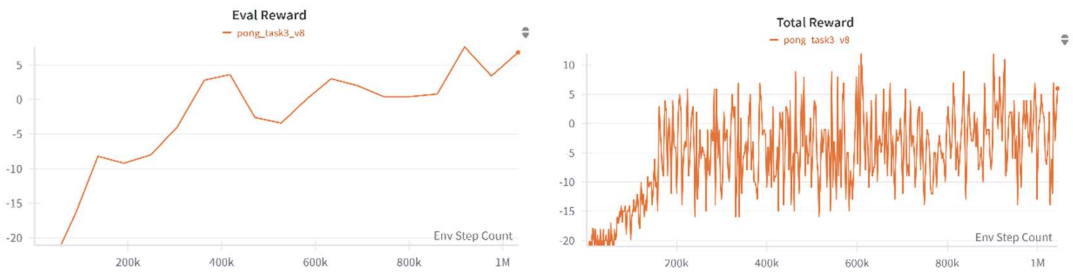
of training. In the later stages, with the help of **beta-annealing**, the sampling gradually returns to uniform, reducing the risk of overfitting to a small set of experiences. In contrast, vanilla DQN uses purely uniform sampling, which avoids overfitting but sacrifices sample efficiency.

2. **Double DQN (DDQN):**
   DDQN addresses the Q-value overestimation problem. While vanilla DQN already uses a target network to mitigate this issue, it still selects actions by directly maximizing over the Q-values estimated by the online network. This tight coupling can still lead to overestimation. DDQN decouples action selection and evaluation by using the online network to choose the action and the target network to evaluate it, resulting in a more stable learning process.

3. **Multi-Step Return:**
   This enhancement reduces the agent's reliance on single-step bootstrapped Q-values by incorporating several future rewards into the return calculation. This allows the agent to better estimate long-term returns and makes it more robust to short-term reward variance. In environments like Pong, where long-term planning is critical, multi-step return helps the agent converge faster and reduces the sensitivity to bootstrapping error.

4. **Dynamic Learning Rate Adjustment:**
   A learning rate scheduler (e.g., StepLR) was applied to start with a higher learning rate in the early phase—accelerating the rise in reward—and gradually reduce it in later stages to allow for finer-grained learning. This setup was inspired by the hyperparameters used in the Rainbow DQN paper, with final values adjusted empirically based on training results.

Below is a summary of the hyperparameter experiments I conducted:



Result of pong_task3_v8

```
--lr 0.00025 ^
--epsilon-decay 0.9999 ^
--n-step 3 ^
--per-alpha 0.3
```

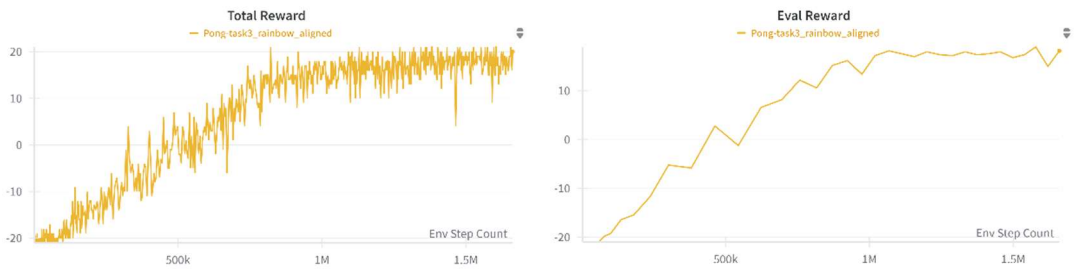
As shown in the figure, under this set of hyperparameters, the performance of the enhanced DQN was poor in terms of convergence speed, final reward, and overall stability. Notably, this was already the best outcome among several configurations I experimented with.

After multiple unsuccessful attempts at manual tuning, I decided to reference the hyperparameter settings from the Rainbow DQN paper and use them as a baseline for further adjustments. The Rainbow-inspired configuration is shown below:

```
--lr 0.0000625 ^
--epsilon-decay 0.99995 ^
--epsilon-min 0.01 ^
--n-step 3 ^
--per-alpha 0.5 ^
--memory-size 250000 ^
--target-update-frequency 8000
```

Note: The only major difference from the Rainbow configuration is the reduced memory-size, which was necessary due to GPU memory constraints.

The results obtained under this configuration are shown below:



Result of pong_task3_rainbow_aligned

Using this configuration, the model achieved significant improvements in both performance and training stability. Although it still did not reach the target score of 19 within 1 million environment steps, the observed stability and convergence timing are sufficient to serve as a strong baseline.

One particularly notable finding was the substantial influence of the epsilon-min parameter on convergence speed. I believe the negative impact of a **higher minimum exploration rate** in Enhanced DQN can be explained by the following:

1. **Prioritized Experience Replay (PER)** assigns higher sampling priority to transitions with large Bellman errors. Since exploratory actions typically result in high TD errors, they are sampled more frequently, which may introduce instability into the training process.
2. **Multi-Step Return** further propagates the rewards (and hence TD errors) of these exploratory transitions across multiple neighboring transitions, potentially amplifying the noise and introducing bias into TD target calculations.

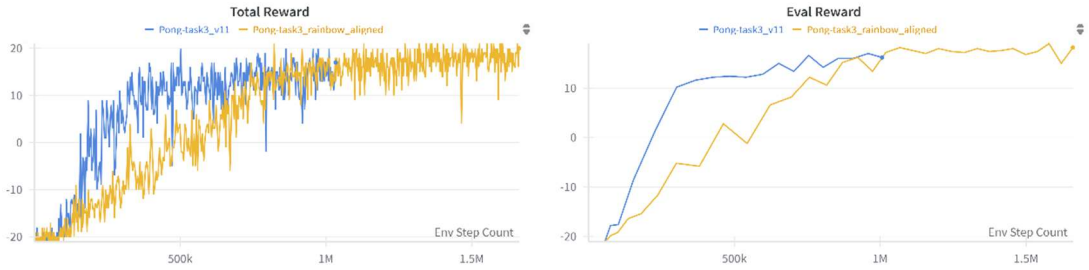
In short, the reason epsilon-min has a greater impact in Enhanced DQN compared to Vanilla DQN is due to the **compounding effect of PER and Multi-Step Return**, which causes exploratory behavior to be emphasized and propagated more heavily. In contrast, Vanilla DQN uses uniform sampling and does not apply multi-step updates, so it tolerates a higher epsilon-min without significantly compromising convergence.

Returning to hyperparameter tuning: although Rainbow's original learning rate of 0.0000625 leads to stable long-term training, Task 3 is graded based on **sample efficiency**, making **early convergence** critically important. To improve convergence speed, I experimented with increasing both the learning rate and the frequency of target network updates.

The configuration for Pong_task3_v11 is as follows:

```
 --target-update-frequency 4000 ^
--lr 0.0001 ^
```

Note: other settings are identical to Pong_task3_rainbow_aligned



Larger Learning Rate + Higher Update-Frequency compared to Rainbow Aligned

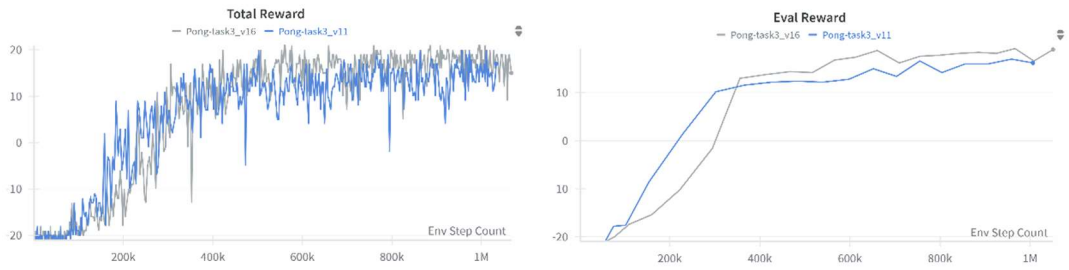
It can be observed that the Pong_task3_v11 configuration—featuring a higher learning rate and more frequent target network updates—indeed accelerates the convergence speed. However, this comes at the cost of reduced fine-tuning capability in the later stages of training. As a result, the evaluation reward plateaued at around 14–16, indicating difficulty in achieving further improvements.

To address this issue, I introduced a **learning rate decay** mechanism. The idea is to maintain a high learning rate during the early training phase to boost reward quickly, then gradually reduce it in the later phase to enable finer updates and avoid the performance plateau.

The configuration for Pong_task3_v16, which includes this scheduling mechanism, is as follows:

```
--lr 0.0001 ^
--scheduler-step-size 200000 ^
--scheduler-gamma 0.5
```

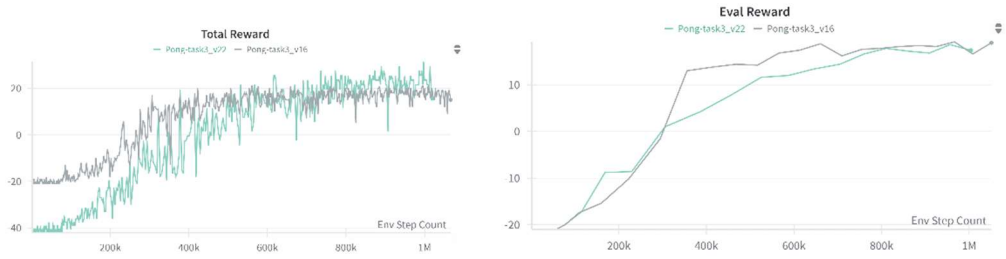
Note: other parameters are identical to Pong_task3_v11



With / Without learning rate decay (Pong-task3_v16 / Pong_tak3_v11)

Pong_task3_v16 successfully retained the fast convergence behavior of earlier configurations while also enabling more stable fine-tuning in later stages of training. It was able to achieve evaluation scores between 17 and 19, demonstrating that the learning rate decay mechanism provided substantial benefits in terms of both performance and sample efficiency.

After observing that the model under Pong_task3_v16 consistently reached scores around 17–19 but struggled to improve further, I attempted to introduce **reward shaping** to strengthen the reward signal. The intention was to facilitate more fine-grained updates in the later stages of training by emphasizing the feedback signal. The experimental results are summarized below:



Result of pong_results_task3_v22 with Reward shaping

As shown in the figure, reward shaping did not improve the model's final performance, and instead slowed down the convergence. I believe this effect was due to the **interaction between reward shaping and Prioritized Experience Replay (PER)**. Strengthening the reward signal also amplifies the Bellman error associated with some incorrect transitions. As a result, PER may over-prioritize misleading samples, reducing training stability and slowing learning progress.

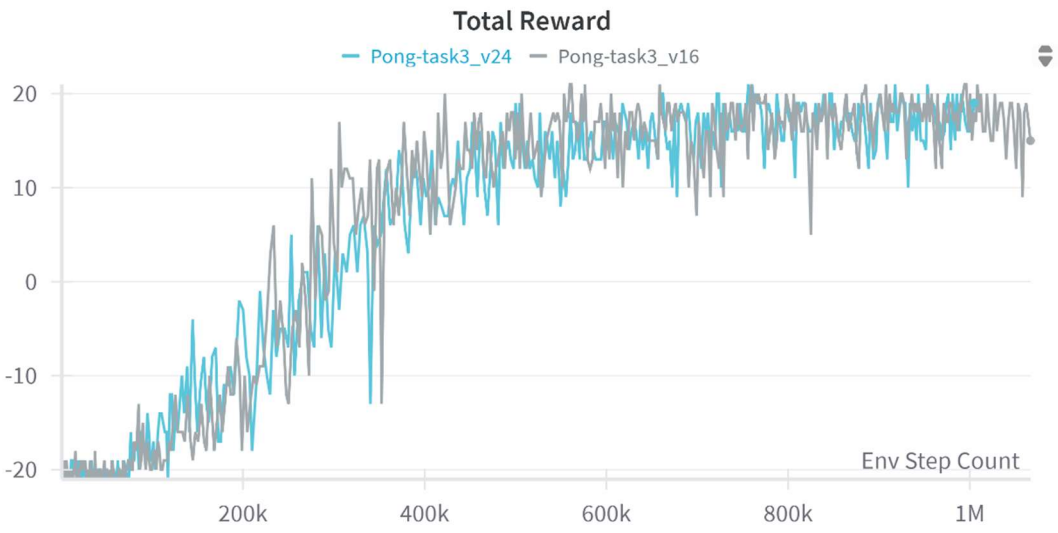
After multiple attempts with various hyperparameter combinations, I observed that the model's performance consistently fluctuated between 17 and 19. This led me to hypothesize that the **late-stage fluctuations** were caused by residual **exploration behavior**. To verify this hypothesis, I modified the Pong_task3_v16 configuration by setting epsilon-min to 0.0, thereby **completely disabling exploration** during the final stages of training. This new configuration is referred to as Pong_task3_v24.

The hyperparameter change is as follows:

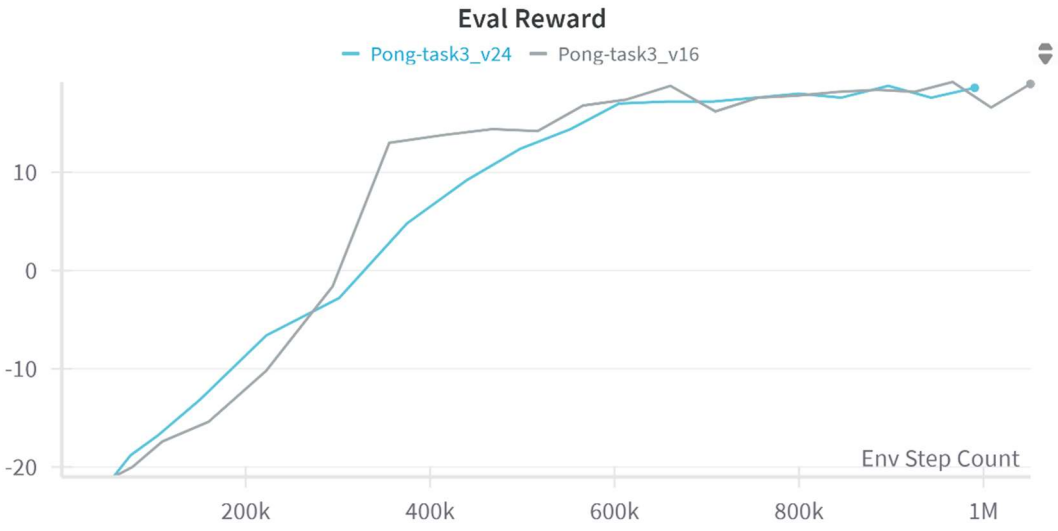
```
--epsilon-min 0.0
```

Note: All other settings remain identical to Pong_task3_v16

Below is the reward curve observed during training under this configuration:



Total Reward: with(v24) / without (v16) setting epsilon-min as 0



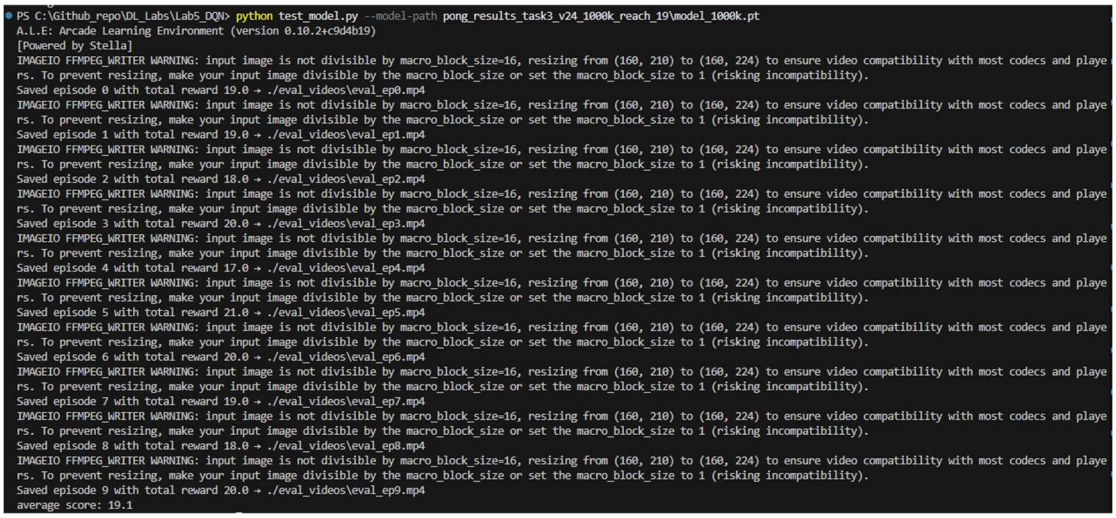
Eval Reward: with(v24) / without (v16) setting epsilon-min as 0

From the evaluation reward curve, it is clear that the model becomes **significantly more stable** after exploration is disabled in the later stages. Under this setting, the model successfully achieved an **average evaluation reward of 19** within

1 million environment steps, meeting the Task 3 scoring criteria. (This result was based on 10 evaluation episodes using the default seed: 313551076.)

Although disabling exploration entirely is generally not advisable—since it may lead to overfitting on limited data and hurt generalization—in this specific context, the strategy proves effective. Task 3 imposes a **strict environment step limit** and requires a score of 19 to receive credit. Under these constraints, the **benefit of eliminating late-stage exploration outweighs the risks**, and overfitting is less of a concern since full convergence is unlikely within the limited training budget.

Therefore, I decided to adopt this strategy as my final submission configuration.



Evaluate result of Pong-task3_v24 ‘s 1M checkpoint

Final Observations:

Across both vanilla and enhanced DQN implementations, I found that training time and learning effectiveness are heavily influenced by hyperparameter settings. For instance, in Task 2, the default configuration provided by the TA required nearly 20 hours of training on an RTX 3090 to reach the target performance. In contrast, a well-tuned custom configuration achieved similar results in just 4 hours on my local RTX 4060Ti (8GB) system.

This sensitivity is even more pronounced in the enhanced DQN, where different hyperparameter combinations could cause the model to converge anywhere between

under 1 million and over 10 million environment steps (based on my observation that some settings still scored under 10 at 3M steps).

This highlights an important characteristic of reinforcement learning: **convergence is guaranteed in theory, but the speed of convergence is not**. Therefore, good model architecture design, the choice of enhancement techniques, and effective hyperparameter tuning are not just implementation details—they are essential for achieving fast and stable learning.

Addition – Correction to Presentation Slide

It was later discovered that there was a mistake in the presentation slide: In Double DQN, the **online network should use the next state s′** to select the action that is then passed to the **target network** for evaluation.
The action selection should **not** be based on the current state s.

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

DDQN on lecture slide

Double Q-learning. Conventional Q-learning is affected by an overestimation bias, due to the maximization step in Equation 1, and this can harm learning. Double Q-learning (van Hasselt 2010), addresses this overestimation by decoupling, in the maximization performed for the bootstrap target, the selection of the action from its evaluation. It is possible to effectively combine this with DQN (van Hasselt, Guez, and Silver 2016), using the loss

$$(R_{t+1} + \gamma_{t+1} q_{\bar{\theta}}(S_{t+1}, \operatorname*{argmax}_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2.$$

This change was shown to reduce harmful overestimations that were present for DQN, thereby improving performance.

DDQN of Rainbow

Reference

1. lecture slide
2. Rainbow paper