Ghent University

Faculty of Engineering and Architecture
Department of Information Technology

Deep Q-Networks

Assignment 1

Enrico Saro
Francesco Pisacane
Simone Santin

Prof. Pieter Simoens

Reinforcement Learning

Academic Year 2025/26

1 Experimental Analysis

In the following sections all experiments are conducted on the *CartPole* environment. We chose this environment because the training time is sufficiently fast while still being challenging enough to reveal meaningful performance differences between models. The insights obtained on *CartPole* are intended to serve as a solid baseline for configuring the agent on the more challenging *Breakout* environment. The results obtained on *Breakout* are discussed in subsection 1.6. .

In this report we discuss Training Reward and Mean Evaluation Return. The first is a moving average (window size 100) of the episode rewards, the second is the average return during periodic evaluations with greedy actions.

The visualization tool provided in the notebook plotted only single runs, making cross-comparisons difficult. To improve analysis, we extended the codebase to:

1. Save detailed logs (rewards, evaluation returns, etc.) for each run.
2. Load multiple logs simultaneously to plot their curves on shared axes.

One issue emerged when comparing reward curves: since training length is fixed by total steps, better agents (surviving longer) complete fewer episodes. To ensure fairness, we normalized the plot's x-axis (episodes) to the $[0, 1]$ range. This enables direct comparison of learning progress across runs.

Parameter	**Value**	**Parameter**	**Value**
γ	0.99	use Double DQN	False
learning rate	1×10^{-4}	use Dueling DQN	False
batch size	64	use N-step	False
buffer capacity	20,000	n-step	3
min buffer size	5,000	use Noisy Nets	False
max steps per env	100,000	vector envs	1
max episode len	1000	use PER	False
ϵ_{start}	1.0	α	0.6
ϵ_{end}	0.05	β	0.4
ϵ_{decay_steps}	20,000	hidden size	256
target update int.	500	log interval (episodes)	10
train freq.	4	eval episodes	5
gradient clip	10.0	reward clip	1

Table 1: Base configuration used for testing.

1.1 Extensions performance

In these sections, all comparisons refer to the *Base Configuration*: the vanilla DQN model, without any extensions and without specific hyperparameter tuning. This configuration is summarized in Table 1.

We now compare the performance of the *Base Configuration* against *Extension A: Double DQN and Dueling DQN*, *Extension B: Prioritized Experience Replay (PER)* and *Extension C: N-Step Returns and Noisy Networks*. Moreover, we analyze the combined effect of all extensions activated.

All experiments used the base parameters' configuration already provided, with only the respective extension flags (`use_double_dqn`, `use_dueling`, `use_prioritized_experience_replay`, `use_n_step`, `use_noisy`) toggled ON.

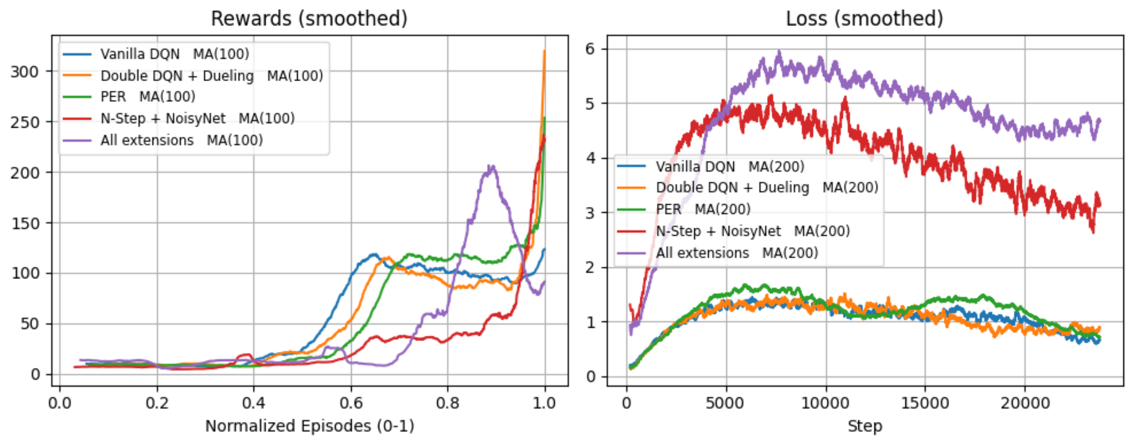

Figure 1: Comparison of Rewards (MA) and Loss (MA) in the CartPole environment for the Vanilla Baseline versus agents with extensions activated.

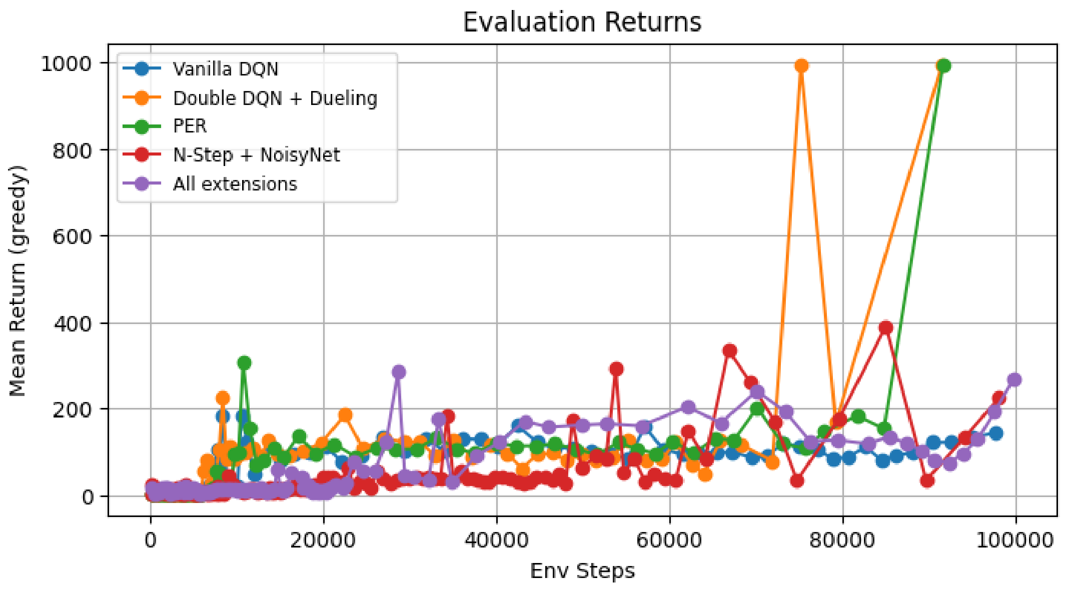

Figure 2: Comparison of mean evaluation returns in the CartPole environment for the Vanilla Baseline vs. agents with extensions activated.

Analyzing the reward plot, all major extensions provide a substantial performance boost over the Vanilla DQN, at least doubling its final average return. In all the cases agents begin to increase rewards at approximately the same point (around 0.4 on the normalized x-axis). Extension C exhibits a slower initial ramp-up than the others but toward the end of the run this agent surpasses the Vanilla DQN.

- Double + Dueling DQN (Extension A) achieves the most significant performance gain and highest peaks.
- PER (Extension B) has the most stable learning curve and highest consistent average reward.
- N-Step + NoisyNet (Extension C) exhibits the slowest initial ramp-up but shows high late potential; this may be due to the higher initial variance of N-Step returns and the time required for NoisyNet's parameter-space exploration to stabilize.

However, combining all extensions results in lower performance than using any single extension alone, which was not expected. This outcome may be caused by suboptimal hyperparameter settings, or it might indicate that the added complexity is actually detrimental in a relatively simple environment such as CartPole. Nevertheless, in the following section we continue using all extensions, aiming to identify better hyperparameter values and achieve more stable and higher performance.

1.2 Hyperparameter Tuning

After evaluating the extensions, we focused on hyperparameter tuning.

Since the search space includes roughly ten to twenty parameters, an exhaustive grid search was not feasible. Instead, we tested one hyperparameter at a time while keeping the others fixed at their initial values (visible in Table 1). All extensions are activated in this phase. This approach allowed us to analyze each parameter's effect on training stability and performance, with the expectation that combining the improved values would yield a stronger overall model. As will be discussed later, this is proven not to be true.

In the following, we analyze the impact of different n-step values and replay buffer capacities, and discuss some of the most relevant findings from the hyperparameter experiments.

1.3 Testing Different Buffer Capacity Values

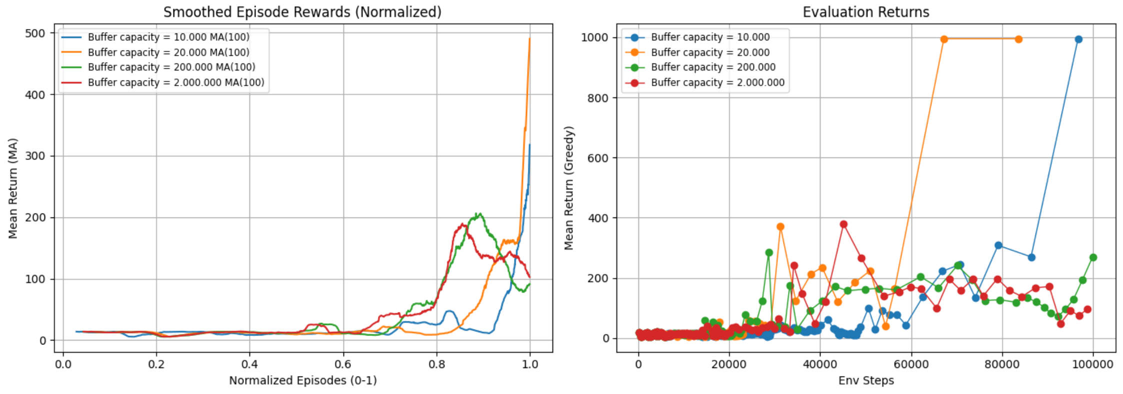

Figure 3: Comparison of results in the CartPole environment for different values of the buffer capacity.

Figure 3 highlights the impact of buffer capacity on learning performance:

- Extremely high buffer sizes (2,000,000) result in lower performance. This is probably due to the fact that older transitions remain in the buffer for too long, making the sampled experiences less representative of the current policy. As a consequence, the agent keeps learning from outdated data, which slows convergence and introduces instability.
- Conversely, smaller buffer sizes (e.g., 20,000) led to faster and more stable learning. In a simple and low-dimensional environment such as *CartPole*, recent experiences already cover most of the relevant state–action space. A smaller buffer therefore allows the agent to focus on more up-to-date transitions, improving learning efficiency and reducing noise in the value estimates.

1.4 Testing different N-step values

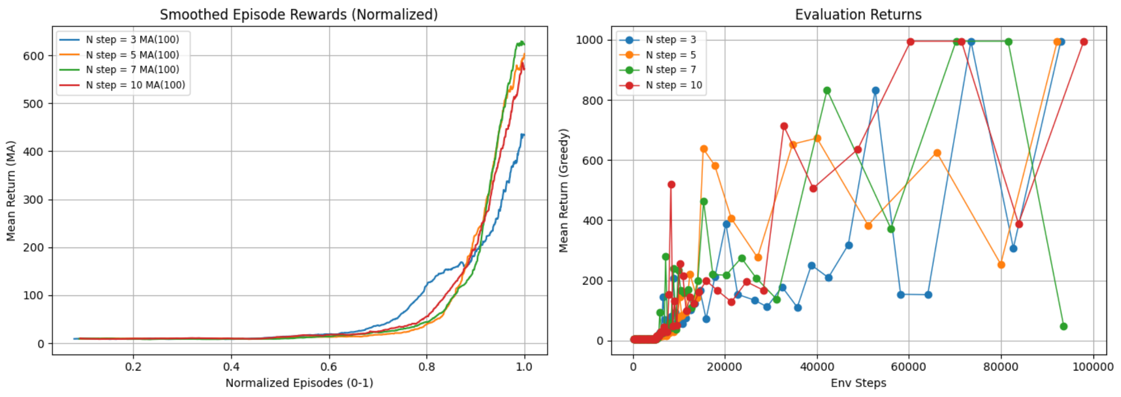

Figure 4: Comparison of results in the CartPole environment for different values of the number N of steps.

Figure 4 shows the performance for various N-step values.

While all agents manage to reach the maximum score of 1000 at various points, suggesting that each configuration is capable of learning an optimal policy, the moving average reward plot shows that larger values of N ultimately achieve higher average rewards (exceeding 600).

We believe this behavior can be explained by the nature of the CartPole environment. Since rewards are dense and highly correlated across consecutive timesteps, using a larger N allows the return estimate to incorporate more meaningful future information while still maintaining a reliable signal. The longer return horizon reduces the bias of the one-step target and provides smoother gradients, helping the agent converge faster toward stable high-reward behaviors.

However, higher values of N also introduce greater variance in the return estimates, leading to less stable evaluation performance. This trade-off is visible in the fluctuating mean evaluation curves.

1.4.1 Other hyperparameters

This section presents a selection of graphs from the hyperparameter tuning experiments, with the key observations discussed in the figure captions.

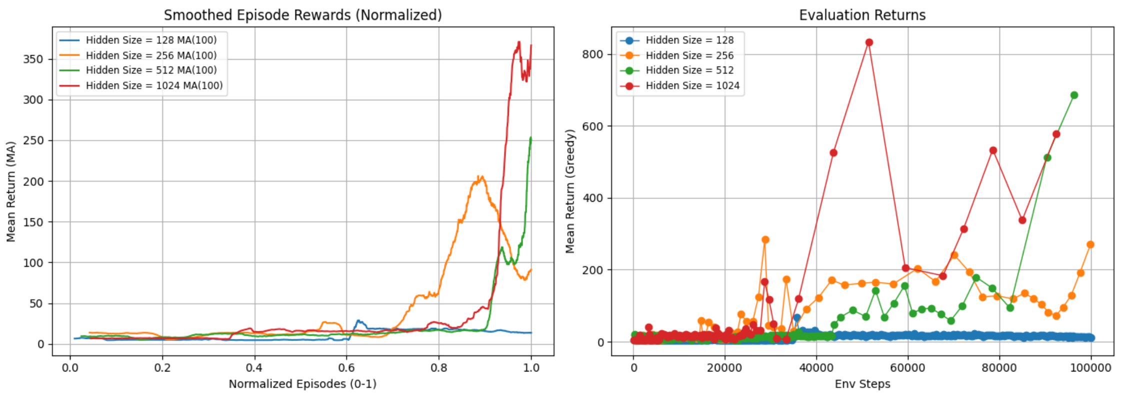

Figure 5: Larger **hidden layer sizes** (512 and 1024) led to higher rewards. However, a size of 1024 also caused instability in learning, while 512 achieved consistent high performance more quickly.

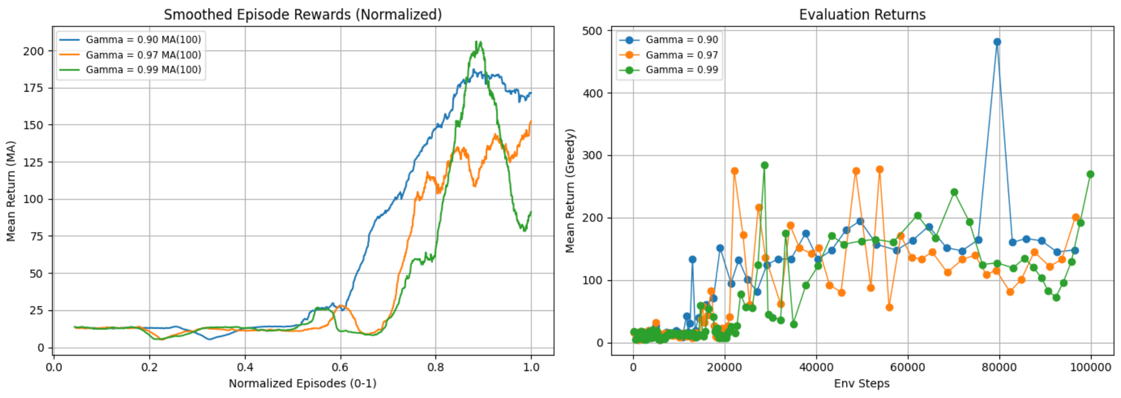

Figure 6: Decresing **Gamma** (γ) values down to 0.9 led to faster and more stable convergence.

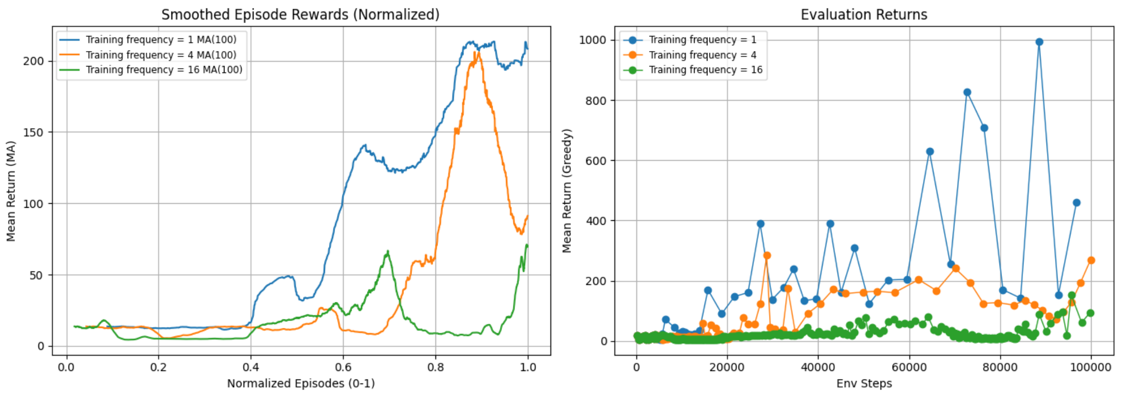

Figure 7: Increasing the **Training frequency** up to 1 allowed to achieve the highest evaluation return peaks, but with extreme volatility; lower frequencies $(4, 16)$ offered greater stability at the cost of lower maximum performance.

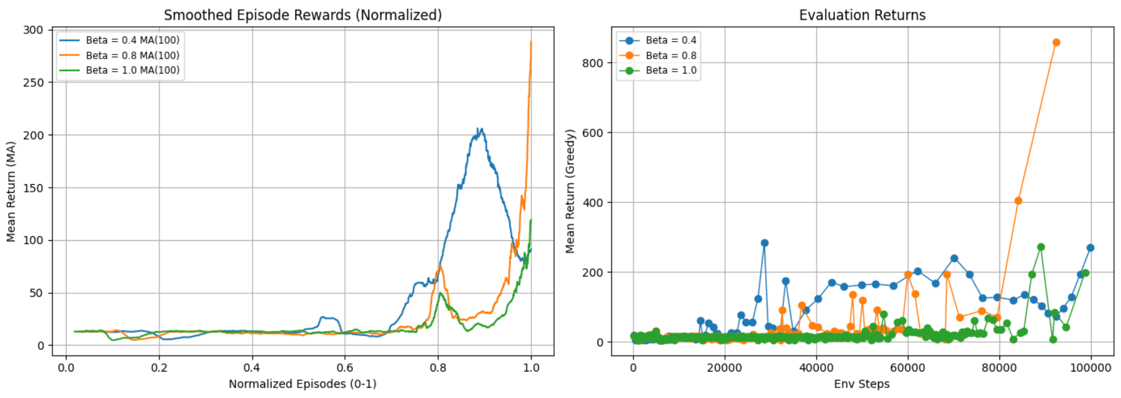

Figure 8: **Beta** (β) = 0.8 generated the highest reward peak and a strong acceleration at the end of the run.

1.5 Final Cartpole results

After the hyperparameter testing, we combined the best-performing values and trained a model using all extensions with those settings. This model differs to the *Base Configuration* in six hyperparameter values. However, as mentioned earlier, the results were disappointing, performing only slightly better than no parameter tuning. This suggests conflicts between different hyperparameters choices. Therefore, we further experimented with several alternative configurations. Figure 9 compares the performance of four setups: the *Vanilla DQN*, the model with all extensions enabled and no parameters tuning, the model with all extensions enabled and theoretically optimal hyperparameters, and the model with all extensions enabled and empirically tuned hyperparameters that produced the best actual performance.

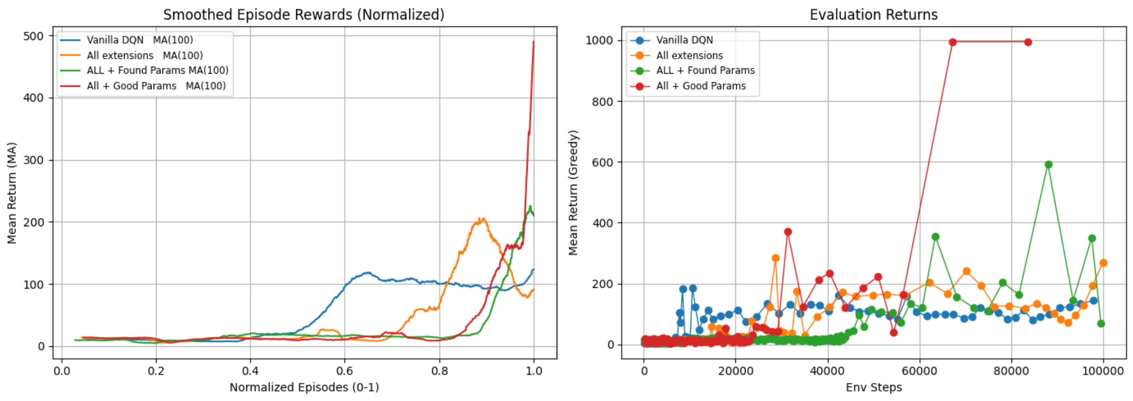

Figure 9: Comparison of the *Vanilla DQN*, the model with all extensions and theoretically optimal hyperparameters, and the model with all extensions and empirically optimized hyperparameters on the CartPole environment.

The hyperparameters in the best-performing model that differ from the *Base Configuration* are the following:

- `buffer_capacity` = 20,000 (previously 200,000)
- `reward_clip` = 1 (previously `None`)

With these adjustments, the model reaches the optimal evaluation return after only 60,000 training steps.

1.6 Breakout Experiments

After the extensive experimentation on the CartPole environment, our next step was to apply the insights gained to the more complex *Breakout* environment. We first explored a few configurations with 5 million steps per environment and then scaled the most promising setup to 15 million steps. The final configuration is reported in Table 2.

Given the significantly higher state dimensionality and the longer temporal dependencies of Breakout, we increased the **replay buffer capacity** and the **minimum buffer size** to ensure that the agent samples from a broader and more diverse distribution of past experiences. This was meant to reduce overfitting to recent transitions and improves sample efficiency. We also enlarged the **hidden layer size** to provide the network with greater representational capacity to model the more complex features present in the game.

To improve training stability, we **clipped the rewards to** ± 1. This prevents large reward magnitudes from inflating Q-values and destabilizing learning dynamics. Finally, we increased the **target update interval** to 4000 steps, which delays the updates to the target network and should mitigate oscillations in the Q-value estimates.

Parameter	**Value**
Replay buffer capacity	1,000,000
Minimum buffer size	10,000
Target update interval	4,000
Reward clipping	1
Hidden layer size	512

Table 2: Final configuration used for the *Breakout* experiments.

The results obtained with the final configuration are shown in Figure 10. Apart from an unusual sharp drop followed by a quick recovery, the mean evaluation return stabilizes around 200–250 after roughly 7 million steps, with occasional peaks reaching 300. This suggests that the training has reached a plateau, and extending it to 15 million steps was likely unnecessary. Although the model is still far from the optimal reward of 864, the results are within the same order of magnitude. This supports the conclusion that our DQN implementation and its extensions are functioning correctly, and that further tuning of the hyperparameters could narrow the remaining performance gap.

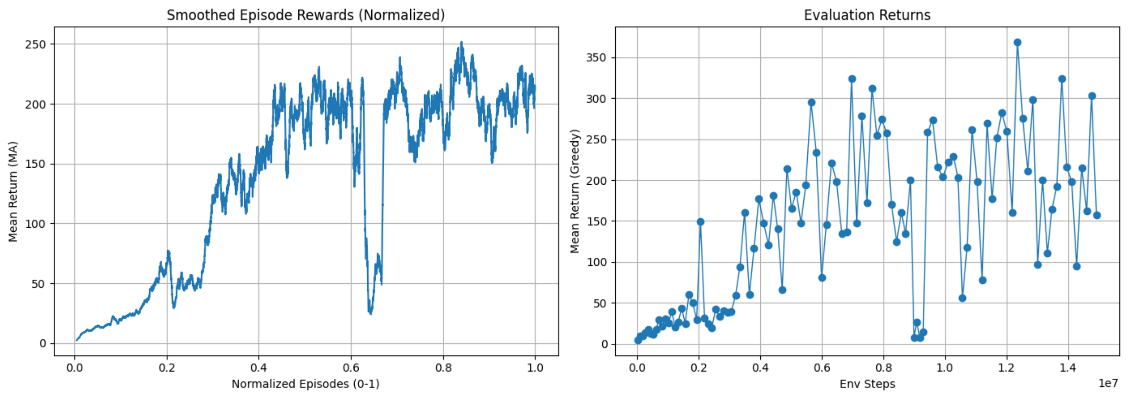

Figure 10: Final results on Breakout environment.

2 Conceptual Questions

Q1. Explain ”maximization bias” in Q-learning. How does your **Double DQN** implementation address it?

When Q-learning takes a max over noisy value estimates, it tends to pick overestimated actions and thus produces an optimistic bias in the Q-value.

Our Double DQN implementation addresses this issue by decoupling selection and evaluation, using target and current Q-networks. The online network selects the best action, and the target network evaluates its value:

$$y = r + \gamma\, Q_{\text{target}}\Big(s', \arg\max_a Q_{\text{online}}(s', a)\Big)$$

Q2. What is the theoretical motivation for the **Dueling DQN** architecture? Why is the special averaging mechanism important?

In the Dueling DQN architecture, the action-value function Q is decomposed into a state-value $V(s)$ and an advantage function $A(s, a)$. This decomposition lets the network learn which states are valuable without having to estimate the effect of every action, which is advantageous when many actions have similar or negligible impact on the environment. Moreover, the dueling architecture is more sample-efficient: observing $Q(s, a)$ for one action in a state also improves the estimates of the other actions' values for that state. The averaging mechanism $Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a')$ is important because it makes the decomposition identifiable: given Q, one can uniquely recover V and A.

Q3. Why is uniform sampling from the replay buffer inefficient? How do the **importance sampling weights** in **PER** correct for the biased sampling you introduced?

Uniform sampling from the replay buffer is inefficient because it treats all transitions equally, causing the agent to repeatedly learn from low-information samples (i.e., those with small TD-errors), which slows learning.

To address the bias introduced by prioritized sampling, PER applies importance sampling weights. These weights rescale the gradient updates during loss computation, reducing the influence of highly prioritized samples and ensuring that the learning process remains approximately unbiased.

Q4. Explain the difference between epsilon-greedy exploration and the exploration provided by **Noisy Networks**. What is an advantage of the latter?
Epsilon-greedy exploration introduces decorrelated, state-independent noise to the policy: with probability ε the agent picks a random action, otherwise it selects the greedy action. In contrast, NoisyNet injects learned, parameterized perturbations into the network weights, producing state-dependent stochasticity that is adapted through gradient updates. This results in structured and adaptive exploration, removes the need for a manual ε-schedule, and often leads to more efficient exploration.

Q5. How does changing N in **N-Step Returns** affect the bias-variance trade-off in your Q-learning updates?
Increasing N moves the target from temporal difference towards Monte Carlo. This reduces bias, because the target relies more on realized rewards and less on current value estimates, but it increases variance since it accumulates randomness over a longer reward sequence. Larger N also decrease bootstrapping, making the update of an estimate less based on another estimate. Conversely, smaller N yields lower-variance, faster, and more stable updates, at the cost of higher bias and greater sensitivity to initial values.

3 Reflection

The main challenge our group encountered was hyperparameter tuning. Exhaustively testing all possible configurations was clearly infeasible, and their effects on performance were often difficult to predict. To address this issue, we initially tested one hyperparameter at a time. The better performing values are then combined in a final run. However, this approach did not yield the expected results: the combined configuration performed poorly, indicating that some hyperparameters interfered with each other. We therefore conducted additional tests to identify such conflicts. For example, we found that setting the PER parameter $\beta = 0.8$ worked well in isolation but led to poor performance when combined with reward clipping and a smaller buffer capacity.

Another major limitation was time and computational resources. While the CartPole experiments required about six minutes per run, running meaningful tests on Breakout was far more demanding, as 100,000 environment steps were not nearly sufficient. For this reason, we only experimented with Breakout after gaining a solid understanding of each hyperparameter's role, allowing us to make educated estimates of suitable values.

The results of the run with all extensions combined were unexpected: the model performed worse than when using any single extension alone. As discussed earlier, this may be due to the additional complexity introduced by combining multiple components, which could be detrimental in a relatively simple environment like CartPole. To verify this hypothesis, the same experiment should be repeated in the Breakout environment.

In the end, we managed to achieve satisfactory results on Breakout by running the agent for several million steps. Even better results could likely be obtained by implementing vectorized environments and automating hyperparameter testing. Such improvements would allow long runs without needing human intervention, hence exploring many parameter combinations over millions of steps.