Reinforcement Learning Assignment 1

Andrei Baroian (s4422090)

Abstract

This paper presents ablation studies for different hyperparameters values for Naive Deep Q-learning as well as an empirical investigation of Deep Q-Network (DQN) variations, including Target Network (TN), Experience Replay (ER), and their combination (TNER) for the CartPole-v1 environment. Under current conditions, the naive method performed surprisingly well compared to the more sophisticated methods and possible reasons are presented.

1. Introduction & Theory

Richard E. Bellman published the book "Dynamic Programming" (Bellman, 1957) which described a systematic method for solving sequences of decisions. He introduced the principle of optimality and a function 1 that captures the core of the priciple. The function was later popularized and standardized as Bellman's Equation 2 by Sutton & Barto (1998). They highlighted its recursive nature and explained that it describes the optimal value of the current state in a Reinforcement Learning Context.

In 1989, Watkins (1989) introduced in his PhD thesis Q-learning (3) which was then refined and formalized (Watkins & Dayan, 1992). They proved theoretical convergence under infinite iterations and emphasised that it does not require a transition model - it learns only from experiences.

$$f(c) = \max \left[\int_0^S F(x, y)\, dt + f(c(S)) \right] \tag{1}$$

$$V(s) = \max_{a \in A(s)} \left[R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V(s') \right] \tag{2}$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{3}$$

20 years later, Mnih (2013) combined Q-learning with convolutional neural networks (CNN) to learn directly in a high-dimensional, continous state space. The authors explain that the optimal strategy is to select the action which maximizes the **expected** state-action value of $r + \gamma Q^*(s_0, a_0)$ and using it as an iterative update, ideas derived from Bellman's Equation and Q-learning. But using the tabular RL update rules - computing the expected value table is impossible in practice, as there are 84x84 pixels **per frame** that need to be represented, and between 4 and 18 valid actions. If we take the upper bound (18 actions), there are 84x84x18 = 127.008 values in the Q table for one state (one frame). In the Atari game, there are 60 frames per second (which can be reduced using skipped frames) and for a 5 minute game, there are 60x5x60 = 18,000 frames, so the Q table would contain 228 billion (18k x 127k) values. This is a huge limitation for tabular RL.

Thus a function approximator is required and the Mnih (2013) chose a neural network with weights θ. The neural network tries to predict the value of every action at a certain state, while the Q-learning gives the reccursive nature and dictates that the action with highest value should be chosen. To update the parameters, a true value (or target y) is necessary, which was chosen as the expected reward of action a on state s (see 4). Now the loss - the difference between the true value (the reward of the action) and the predicted value (the output of the Q function with state and action as input) is computed and the network's parameters are adjusted by taking a step into the direction that minimizes that difference (Gradient Decent).

$$\begin{aligned} y &= r + \gamma \max_{a'} Q(s', a'; \theta^-), \\ L(\theta) &= \left[y - Q(s, a; \theta) \right]^2. \end{aligned} \tag{4}$$

2. Methods / Experiments

First, a naive integration of a neural network was implemented and different hyperparameters were tested for 1e5 steps. Second, best hyperparameters are used to compare the naive method to the method after adding Target Network, Experience Replay, and both for 1e6 steps.

2.1. Problem - Cartpole

The methods are applied to the cartpole problem: "A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the

cart and the goal is to balance the pole by applying forces in the left and right direction on the cart." (Farama Foundation, 2024). There are two actions - move left and move right. The observation state consist of 4 variables: Cart position, cart velocity, pole angle and pole angle velocity. The game ends when the pole is inclined at more than 24 ° or if the cart is outside of the screen. A reward of 1 is given for each state until the game ends, to a max of 500 steps.

2.2. Ablation Studies for Naive Agent

A naive agent refers to a straight-forward implementation of a neural network with Q-learning where the network predicts the Q value and its weights are updated with the most recent transitions. The naive agent starts by initializing a memory buffer and random Q-values, selects actions via epsilon-greedy, gathers experience in the buffer, periodically updates the network's weights based on (s_t, a_t, r_t, s_{t+1}) from the memory, through gradient descent (see Appendix A for pseudocode).

Update frequency refers to the number of steps (N) before the weights are updated. The last N transitions (s,a,r,s') are stored in memory, all are used to update the weights, then erased from memory.

To control for the exploration-exploitation trade-off, three hyperparameters were considered initially: fix epsilon and control for it; epsilon decay from 1 to 0.1 (inspired by (Mnih, 2013)) and control for the rate (speed) of the decay ($\epsilon_n = \epsilon_{n-1} \times$ decay_rate); linearly decay from 1 to 0.1 and control for the stop percentage (eg: decay for the first 10% of the env steps). Based on initial experiments, epsilon stop percentage was chosen as it explores the trade-off better and it's easier to understand.

The following hyperparameters (HPs) were tested (table 1): Gamma, learning rate, update frequency, epsilon stop percentage, layer size, and number of layers. In total there are 6 hyperparameters, 27 values, running 5 times each, resulting in 130 experiments. Each experiemnts runs for 10e5 environmental steps.

Table 1. Hyperparameter Configuration for Ablation Study

Hyperparameter Configuration		
Default Values		**Ablation Values**
Parameter	**Values**	**Values Tested**
update frequency	1	[1, 3, 5, 10, 50]
max steps	500	–
total steps	1e5	–
gamma	0.95	[0.5, 0.8, 0.9, 0.95, 0.99]
epsilon start	1.0	–
epsilon min	0.1	–
epsilon stop percentage	0.1	[0.01, 0.05, 0.1, 0.25, 0.5]
learning rate	0.001	[5e-4, 1e-3,5e-3,1e-2]
layer size	32	[32, 64, 128, 256]
nr layers	3	[3, 5, 10]

2.3. Experienced Replay And Target Network

The naive approach has fundamental limitations and hyperparameter optimization alone could not surpass them. Thus new methods were needed: Experienced Replay and Target Network (see Algorithm 1).

Experienced Replay. All the samples the network is learning from are correlated in the naive approach, resulting in small adjustments to the network having a huge effect on the loss, destabilizing it. This results in the network never converging. Instead of learning from the last examples, the experience $e_t(s_t, a_t, r_t, s_{t+1})$ is stored at each step and stored in a memory buffer $D_t\{e_1, \ldots, e_t\}$. A batch of experiences is sampled uniformly at random to update the network's parameters at an interval determined by update frequency.

Target Network. The naive approach uses the same network to determine the true (target) value and the predicted value, resulting in a *chasing it's own tail* situation, creating a feedback loop. Each update changes the target value so it's impossible to predict it and causes divergence and unstable training. To solve this, a target network is introduced (Mnih et al., 2015) to predict the target value; the network is updated (as a copy of the Q-network) every C steps.

Algorithm 1 Deep Q-Learning with Experience Replay and Target Network

Initialize replay memory D to capacity N
Initialize Q-network with random weights θ and target network with weights $\theta^- = \theta$
while not done training **do**
  Observe state s_t
  With probability ϵ select random action a_t, else $a_t = \arg\max_a Q(s_t, a; \theta)$
  Execute a_t, observe reward r_t, next state s_{t+1}, and done (final state)
  Store experience $(s_t, a_t, r_t, s_{t+1}, done)$ in replay memory D
  if is time to update (update frequency) **then**
    Sample random minibatch of experiences from D
    Set targets $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$ (or $y_j = r_j$ if done)
    Perform gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$
  end if
  if time to update target network (every C steps) **then**
    Update target network weights: $\theta^- = \theta$
  end if
end while

The hyperparameters were chosen based on ablations for the naive method, and the following were added regarding the new methods: update frequency for target network = 50; replay buffer size = 1e5; batch size = 16; total environment steps = 1e6. See Appendix 4.

3. Results

3.1. Naive Hyperparameter Ablation

Figures with reward and loss for 27 hyperparameter values and tables with statistics (mean, std, min, max for reward) can be found in Appendix B. They are summarize in Table 2. Findings and observations:

- **Update frequency** and **layer size** do not significantly affect average reward or standard deviation
- 0.1 best for **Epsilon Stop Percentage**, it shows a continous improvement while other values plateau after around 30% of total steps.
- The lower the Epsilon Stop, the higher the variance of the reward 1.
- 0.95 for **Gamma** seems to be the best, but not conclusive/significant, as its reward is very similar to 0.9, 0.95, 0.99
- 0.005 best for **Learning rate** - probably because of the small number of steps. It is expected as the number of steps increases, the ideal learning rate to decrease.
- The lower the learning rate, the flatter the loss function and the smaller the variance
- Three layers proved the best, sufficient for the simple task at hand. But the implementation didn't optimize for deeper networks (eg: using residual connections)

Table 2. Aggregated Summary of Best Hyperparameters
Diff 2nd / last = the difference between the best hyperparameter value and the second / last value

Hyperparameter	Value	Avg Reward	Std	Diff 2nd	Diff Last
Epsilon Stop	0.1	136.80	147.04	26.84	47.78
Gamma	0.95	120.08	119.54	15.32	30.12
Layer Size	256	136.31	113.77	8.89	13.37
Learning Rate	0.005	104.25	111.93	4.25	9.25
Layers	3	109.19	99.27	4.19	29.19
Update Frequency	1	124.81	122.32	1.81	14.81

3.2. Target Network and Experience Replay

Figure 2 shows the Average Reward and Training loss over 1e6 steps. For interpretability, two modifications were made: 1) The y-axis for the loss graph was set to log-scale as an early observation showed a high loss for Experience Replay; 2) the standard deviation is not shown for the reward graph - still, it's not easy to read and an attempt was made to smoothened the reward for clearer results (see Appendix 14).

Table 3 shows the statistical results. The average reward across all 5 runs across all episodes was recorded but it was not a reliable metric as the variance of the number of episodes in 1e6 steps was high between the four methods (naive method had on average 4659.4 episodes per 1e6 steps, while TN&ER had 11314.8 - see Appendix 5) Still, the standard deviation of that average reward over the 5 runs was recorded and proved insightful when seen as a percentage of avg reward - it increases as the std over all steps decreases. This means that even though the individual runs are more stable, different runs differ more (greater variance) from each other.

Surprisingly, the naive method performed better than Experience Replay and the TN&ER method in terms of reward; but this changes the standard deviation over all steps is considered - TN&ER has lower std (difference of 41 absolute points), meaning it is much more stable. These findings suggest that there is a trade-off between speed of learning

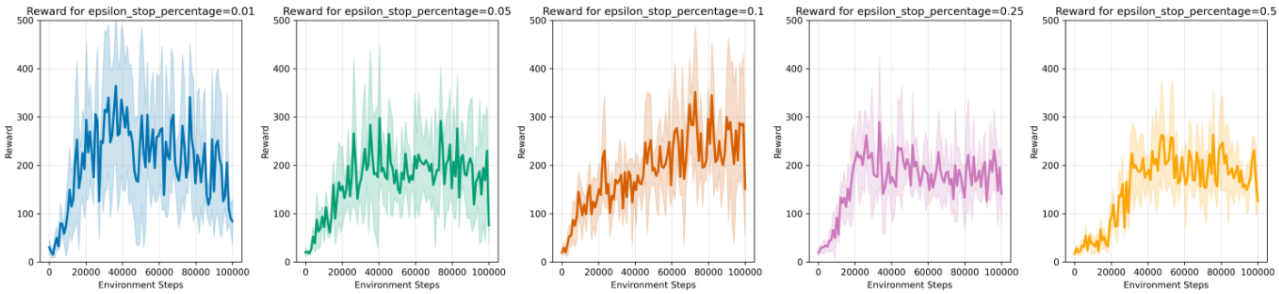

Figure 1. Epsilon Stop Percentage, 5 values for 1e5 steps

and stability. We can observe this in the Reward Graph 2, but here it seems like the difference is predominant only at the beginnig of the trading; towards the end, all methods have relatively the same std, none converging to a stable solution. We can Experience Replay's issue of convergence in the Training Loss graph, where it is stable in the first 0.4e6 steps, but then the network becomes unstable, has multiple spikes and it's not able to recover. It reaches a max loss of 10e9, an incredible high value. It is difficult to come up with an explanation at this moment, but further experiments with the replay memory size (now at 10e5 transitions) could bring more understanding. Adding the target network seems to help a lot, as TN&ER has a more stable training. Still, none of the methods tested come close to the reward performance of the Random Baseline (reward of 490 for the last 200k steps) or to the optimal performance, hinting at a possible implementation issue.

Table 3. Comparison of DQN Methods on CartPole-v1
Average Reward for the Last 200k steps; Reward for the Last Episode; Standard Deviation over all 1e6 steps; Standard Deviation of avg reward over 5 runs as percentage of avg reward

Method	**Avg Last 200K**	**Last Ep Reward**	**Std**	**Std Over Runs %**
Naive	289.21	181.63	151.10	2.2
TN	319.35	147.75	118.10	4.0
ER	246.7	113.46	114.10	5.6
TN&ER	254.49	120.41	111.93	7.7
Random Baseline	490.03	500.00	163.92	-

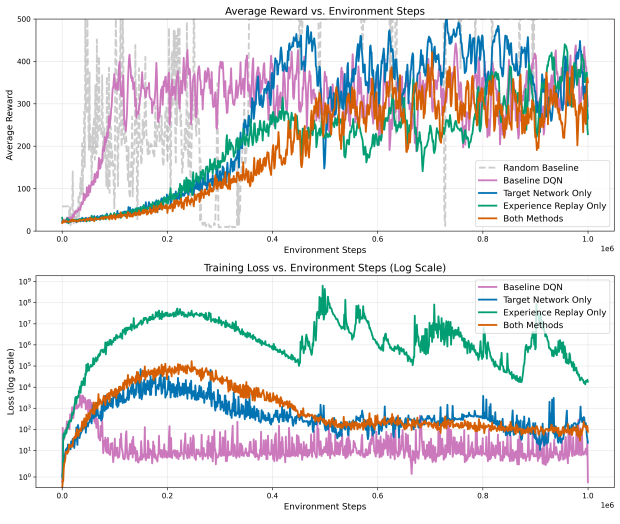

Figure 2. 4 Methods (Naive, ER, TN, ER&TN) compared Naive Agent is Baseline; Random Baseline is from the data recieved as example

4. Discussion

The hyperparameter ablations showed the effect of the hyperparameters (HPs), but the purpose was not to find the best absolute HPs, but rather to learn how they affect the reward and training loss, as there are intricate relations within the HPs tested and between the fixed HPs. For example the rank of the HPs values would change for different numbers of environmental steps. The found HPs are also problem-specific and they might serve as a good start for HP tuning for other problems, but would not be optimal.

The results are surprising as the naive approach performs well compared to the more sophisticated methods which were empirically proven to work very well (Mnih, 2013)(Mnih et al., 2015)(Hessel et al., 2018). It might hint at a possible mistake in the code implementation of the algorithm, although it was thoroughly checked against and compared with stable baseline 3 implementation (et al., 2025). Another possible explanation is the number of environmental steps - Hessel et al. (2018) point that DQN works but it's inefficient (50 million frames are used in in (Mnih, 2013)). Jari (2020) tested DQN for cartpole v1 for 2e6 steps hitting a score of 500 after 5e5 steps, and convergening at 1.2e6 steps. The combination of these two factors (implementation and number of steps) explains the unexpected results.

Compute Resources

All experiments were run on the DS lab (Leiden University), on the *vibranium.liacs.nl*. For the Naive Agent hyperparameter optimization (130 experiments), 34 were run in parallel on one GPU and took 2.41 hours. For the second part, 20 Experiments (Naive, TN, ER, TN&ER) were run with 1e6 environment steps, taking 12.09 hours.

5. Conclusion

In this paper, an empirical analysis of Deep Q-Network (DQN) variations was performed, including naive Deep Q-learning, Target Network (TN), Experience Replay (ER), and their combination (TN&ER), within the CartPole-v1 environment. Contrary to expectations from existing literature, the naive implementation performed comparably or even better in terms of reward than the sophisticated TN and ER methods, suggesting possible implementation issues or suboptimal hyperparameter selection. However, the complementary of TN and ER was observed. The experiments also revealed the effect of the hyperparameters, but it served as a learning lesson rather than an absolute best. Future work should focus on validating the correctness of implementations, exploring extended training durations beyond 1 million steps, and investigating additional improvements presented in (Hessel et al., 2018).

References

Bellman, R. E. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

et al., R. Stable-baselines3: Dqn documentation, 2025. URL https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html. Accessed: March 15, 2025.

Farama Foundation. CartPole-v1 — Gymnasium Classic Control Environment, 2024. URL https://gymnasium.farama.org/environments/classic_control/cart_pole/. Accessed: 2025-03-06.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Jari, S. Deep q networks (dqn) with the cartpole environment, 2020. URL https://wandb.ai/safijari/dqn-tutorial/reports/Deep-Q-Networks-DQN-With-the-Cartpole-Environment--Vmlldzo4MDc2MQ.

Leiden University. Data science lab. URL https://rel.liacs.nl/labs/dslab. Accessed: 2025-03-15.

Mnih, V. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998. ISBN 978-0262193986.

Watkins, C. J. C. H. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.

Watkins, C. J. C. H. and Dayan, P. Q-learning. *Machine Learning*, 8(3–4):279–292, 1992.

A. Naive Agent Q Learning Pseudocode

Algorithm 2 Naive Agent Q-Learning

Initialize memory buffer D to capacity N
Initialize action-value function Q w/ random weights θ
while not done training **do**
  Initialize state s_1
  while $t = 1$ to T **do**
    With probability ϵ select a random action a_t, else select $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute a_t, observe reward r_t and next state s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in memory buffer D
    if size of memory buffer = update frequency **then**
      update Q with memory buffer
      for (s_t, a_t, r_t, s_{t+1}) in memory buffer **do**
        $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta)$
        Gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$
      end for
      empty memory buffer
    end if
    Update state: $s_t \leftarrow s_{t+1}$
    Decay ϵ according to the decay schedule
  end while
end while

B. All Naive Experimental Results

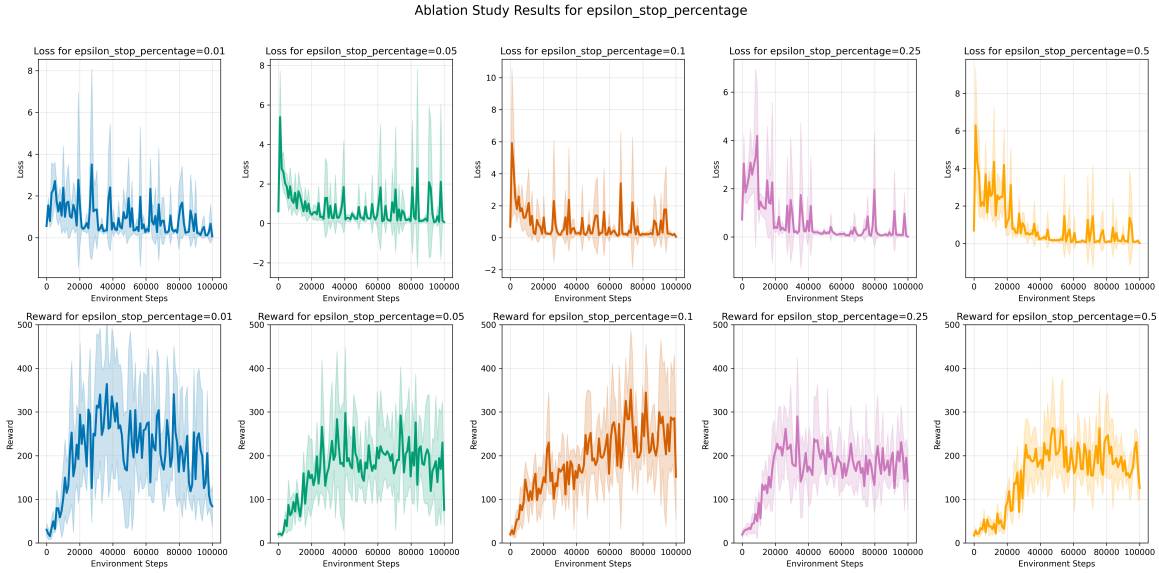

Figure 3. Visualization of reward and loss for different epsilon stop percentage values.

Value	Avg Reward	Std Dev	Min	Max	Final Avg Loss
0.01	109.96	119.42	8.00	500.00	0.023527
0.05	107.37	95.85	1.00	500.00	0.118379
0.1	136.80	147.04	8.00	500.00	0.030357
0.25	106.95	98.07	8.00	500.00	0.021179
0.5	89.02	89.23	8.00	500.00	0.007839

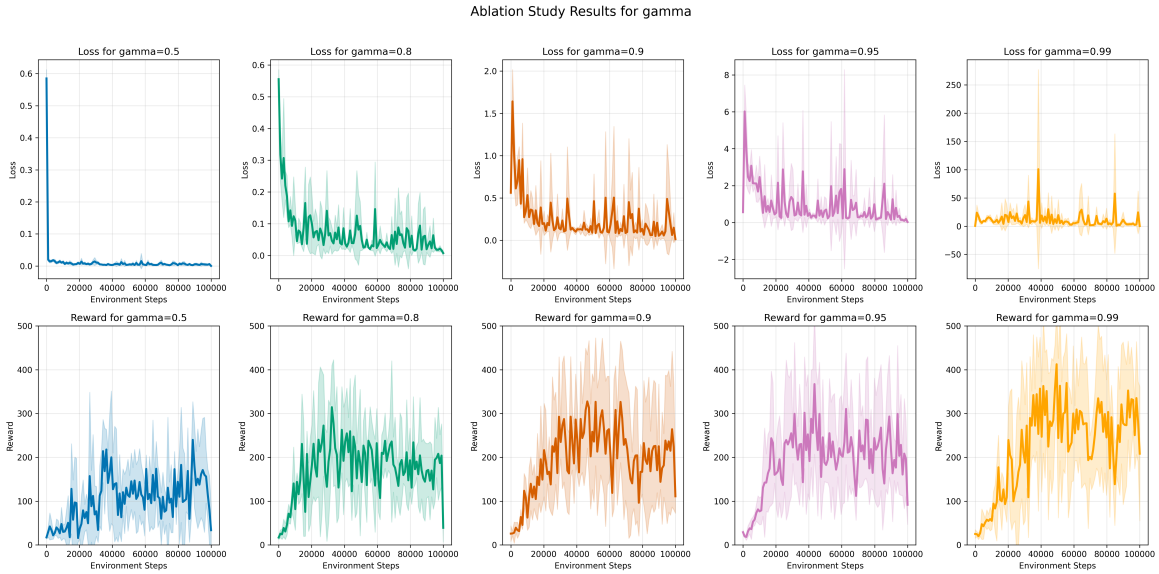

Figure 4. Visualization of reward and loss for different gamma (discount factor) values.

Value	Avg Reward	Std Dev	Min	Max	Final Avg Loss
0.5	35.10	57.71	3.00	500.00	0.000548
0.8	89.85	98.09	8.00	500.00	0.001134
0.9	103.20	96.21	6.00	500.00	0.003796
0.95	120.08	119.54	8.00	500.00	0.015981
0.99	125.44	145.06	8.00	500.00	2.337833

Figure 5. Ablation Study Results for Gamma (Discount Factor)

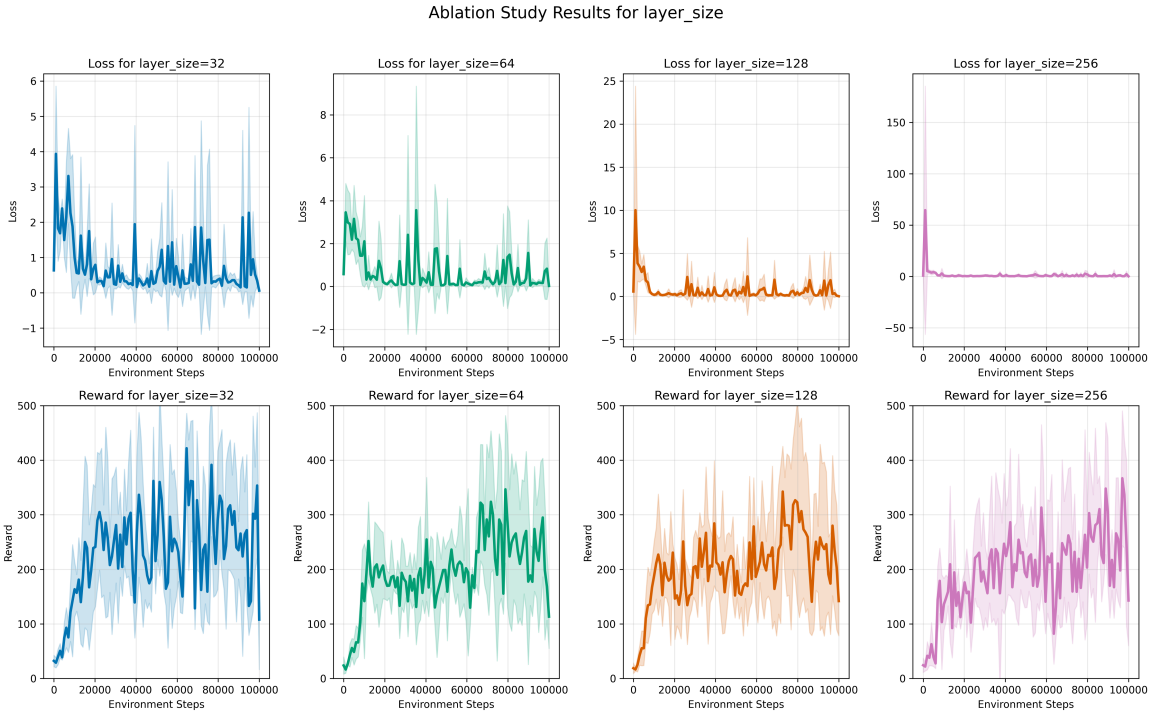

Figure 6. Visualization of reward and loss for different neural network layer sizes.

Value	Avg Reward	Std Dev	Min	Max	Final Avg Loss
32	123.30	132.03	8.00	500.00	0.054899
64	122.94	104.33	1.00	500.00	0.012351
128	127.42	104.48	8.00	500.00	0.031448
256	136.31	113.77	8.00	500.00	0.059634

Figure 7. Ablation Study Results for Neural Network Layer Size

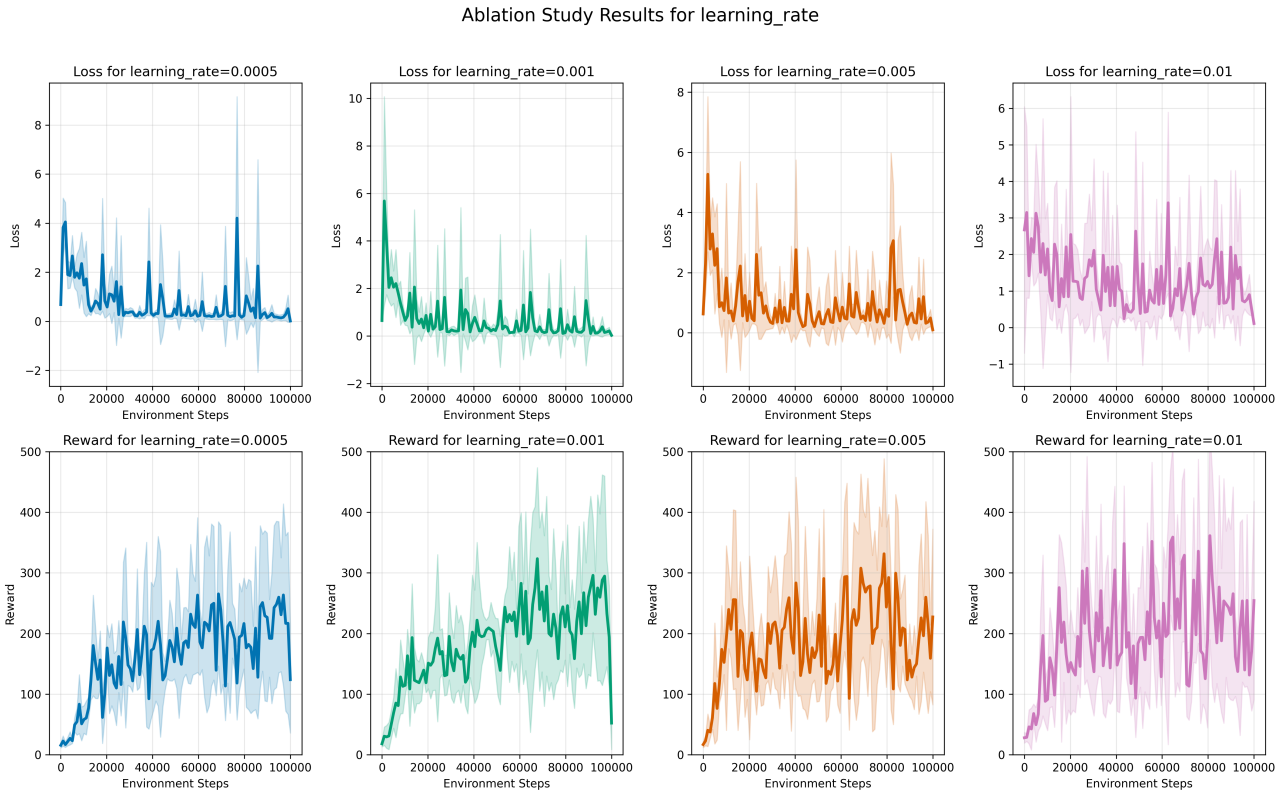

Figure 8. Visualization of reward and loss for different learning rate values.

Value	Avg Reward	Std Dev	Min	Max	Final Avg Loss
0.0005	96.26	109.02	8.00	500.00	0.018909
0.001	126.68	126.09	8.00	500.00	0.011609
0.005	104.25	111.93	8.00	500.00	0.081573
0.01	117.79	127.16	8.00	500.00	0.114144

Figure 9. Statistical results for different learning rate values.

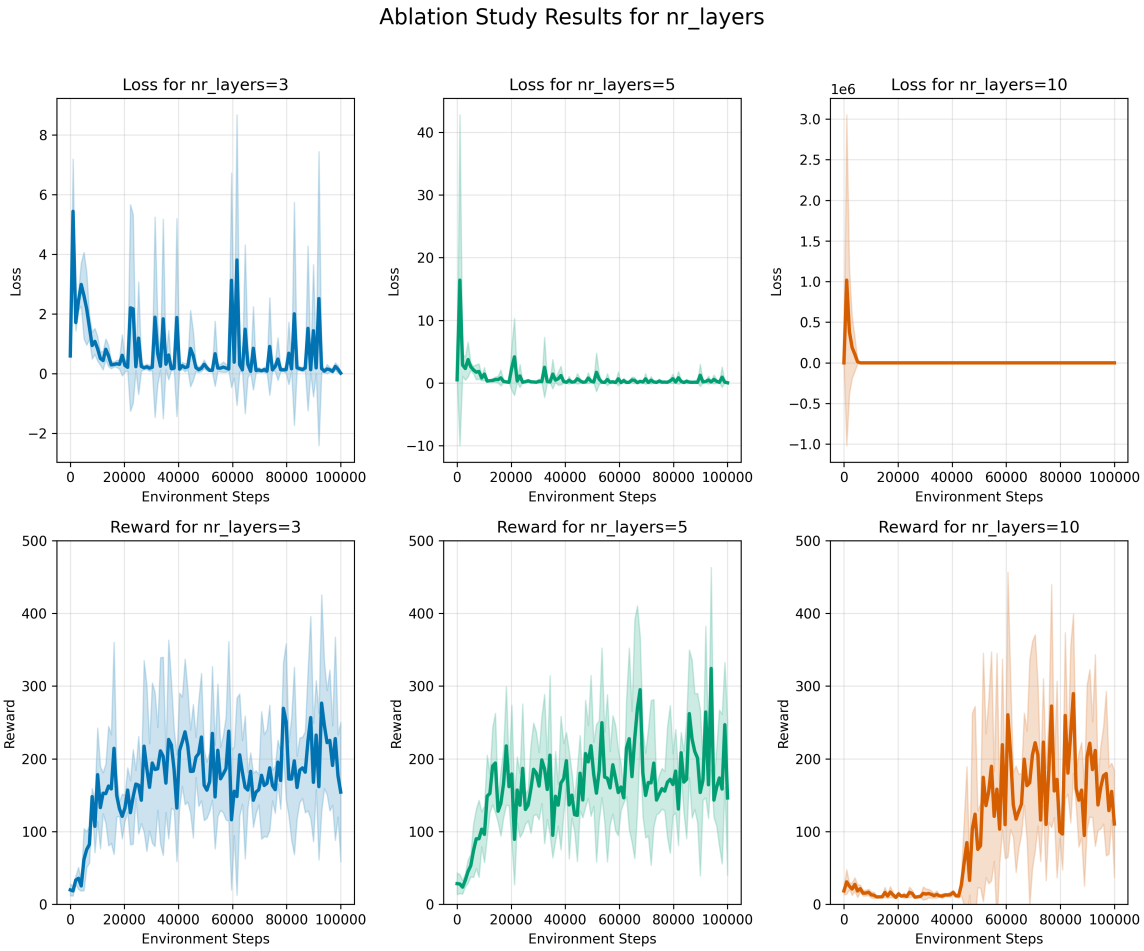

Figure 10. Visualization of reward and loss for different numbers of neural network layers.

Value	Avg Reward	Std Dev	Min	Max	Final Avg Loss
3	109.19	99.27	8.00	500.00	0.029973
5	111.56	112.69	8.00	500.00	0.027672
10	15.37	22.72	1.00	500.00	0.237704

Figure 11. Statistical results for different numbers of neural network layers.

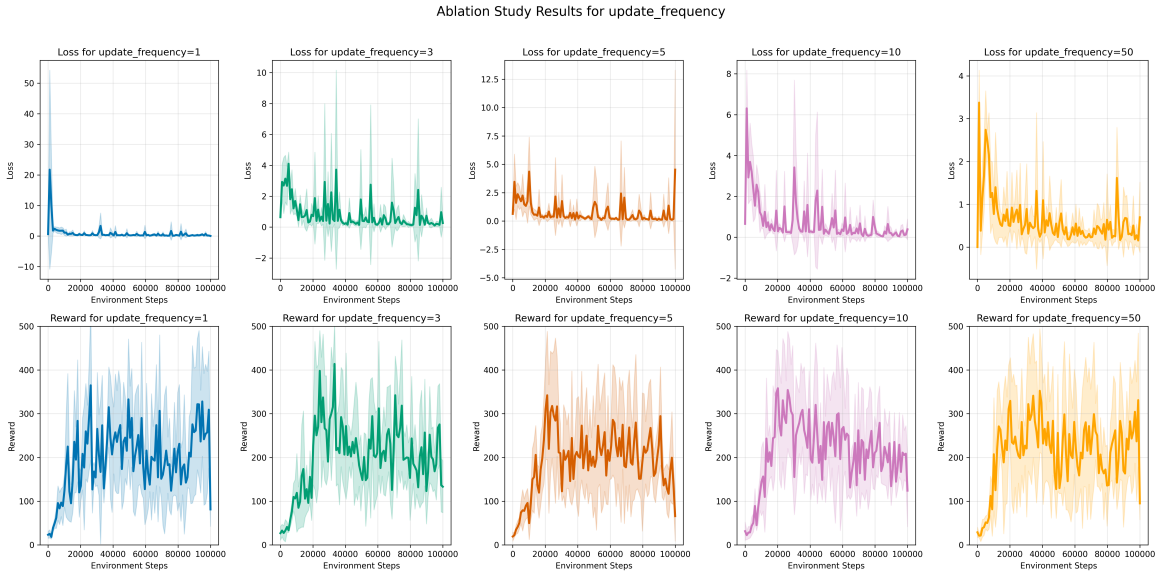

Figure 12. Visualization of reward and loss for different update frequency values.

Value	Avg Reward	Std Dev	Min	Max	Final Avg Loss
1	124.81	122.32	8.00	500.00	0.009393
3	126.01	124.80	8.00	500.00	0.070969
5	112.56	114.56	8.00	500.00	0.257894
10	125.06	129.41	8.00	500.00	0.745925
50	123.00	126.66	8.00	500.00	0.853065

Figure 13. Statistical results for different update frequency values.

C. Four Methods Comparison Extended Results

Table 4. Fixed Hyperparameters for DQN Experiments

Hyperparameter	**Value**
Update Frequency	5
Maximum Steps per Episode	500
Total Environment Steps	1,000,000
Discount Factor (γ)	0.99
Initial Exploration Rate (ϵ_{start})	1.0
Minimum Exploration Rate (ϵ_{min})	0.1
Exploration Stop Percentage	0.1
Learning Rate	0.001
Hidden Layer Size	64
Number of Hidden Layers	3
Target Network Update Frequency	50
Replay Buffer Size	100,000
Batch Size	16

Table 5. Full Statistical Results 4 Method Comparison

Method	Avg Reward	Std (Runs)	Std (Eps)	Avg Episodes	Last Ep Reward	Last 200K Avg
Baseline DQN	214.73	4.77	181.63	4659.4	173.80	289.21
Target Network Only	109.60	4.40	147.75	9139.0	196.40	319.35
Experience Replay Only	92.24	5.23	113.46	10876.2	255.40	246.70
Both Methods	88.97	6.85	120.41	11314.8	215.60	254.49
Random Baseline	384.42	N/A	163.92	N/A	500.00	490.03

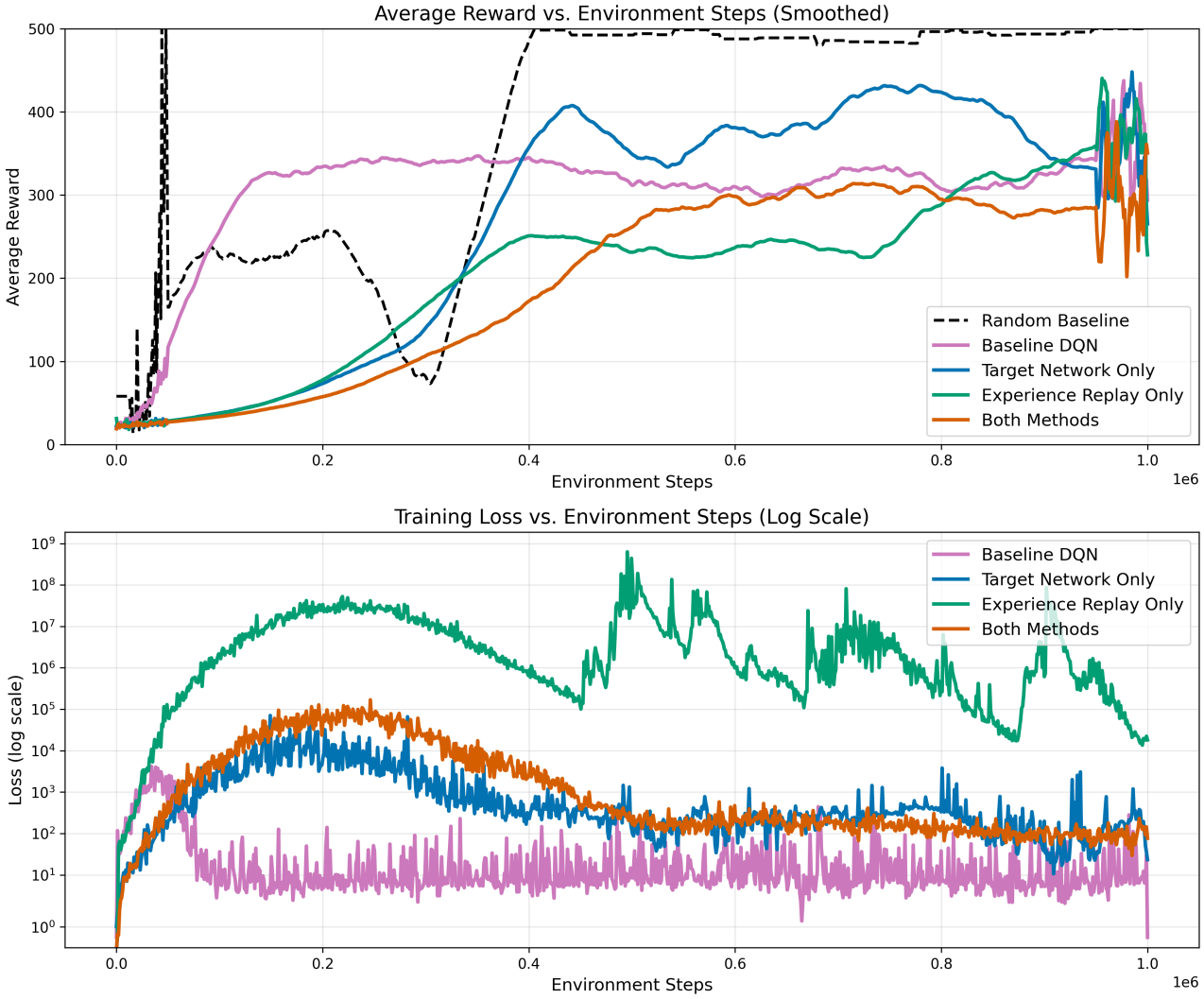

Figure 14. 4 Methods Comparison Smoothened Reward