# Reinforcement Learning Assignment 1 Report
# DQN implementation on CartPole-v1

**Kevin Bretz,** [1]

## Abstract

In this report, I will take on the first assignment in the Leiden University Computer Science Master course on Reinforcement Learning. I start by answering theoretical questions about how the loss function in Q-Learning is derived, and why we cannot use update rules as in tabular RL. Then, I implement a naive Deep Q-Network (DQN) in PyTorch, and train it on the CartPole-v1 environment. I perform an ablation study on the hyperparameters, and using the best results I implement Experience Replay and Target Network functionality into my agent. Finally, I discuss the results and the limitations of my DQN implementation.

## 1. Introduction

For this assignment, I implemented a DQN to solve the CartPole-v1 environment. DQNs were first introduced by (Mnih et al., 2013) and later improved in (Mnih et al., 2015). I implemented my DQN using PyTorch (PyTorch, 2024) and tested it on the CartPole-v1 environment from Gymnasium (Gymnasium, 2023). The theoretical foundations of DQN are well explained in (Plaat, 2022).
DQNs combine Q-learning with neural networks, allowing them to handle complex state spaces that traditional tabular methods cannot process.

The CartPole-v1 environment provides a good testing ground for this investigation. It's a simple control problem where an agent needs to balance a pole by moving a cart left or right. The environment's simplicity lets me focus on understanding how different DQN components affect learning.

I first perform an ablation study to find the best hyperparameters for my implementation. Then, I test how experience replay and target networks impact the performance, both individually and combined. This helps understand when these enhancements are actually beneficial.

[1]LIACS, Leiden University, Leiden, The Netherlands. Correspondence to: Kevin Bretz,

*Proceedings of the $42^{nd}$ International Conference on Machine Learning*, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).

## 2. Theory

### 2.1. Q-Learning Loss Function

To understand how DQNs work, we need to look at the Q-learning update rule first. In regular Q-learning, we update Q-values using:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{1}$$

For DQNs, we use a neural network to approximate Q-values instead of a table. This means we need a loss function to train the network:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2\right] \tag{2}$$

Here, $\theta$ are the network parameters, $\theta^-$ are the target network parameters, and $D$ is the replay buffer.

### 2.2. Deep Q-Network Architecture

We can't use the same update rules as in tabular Q-learning for two main reasons:

1. The state space is continuous, so we can't store all possible state-action pairs in a table
2. Updating one Q-value affects others because they share network parameters

This is why we use gradient descent and other techniques to stabilize training.

### 2.3. Pseudocode

Algorithm 1 presents the DQN implementation with experience replay and target networks:

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

Initialize Q-network $Q(s, a; \theta)$ and target network $Q(s, a; \theta^-)$ with random weights
Initialize replay buffer $D$ with capacity $N$
**while** Total steps $< 10^6$ **do**
  **for** episode = 1 to M **do**
    Initialize state $s_1$
    **while** not Terminated or Truncated **do**
      Select action $a_t$ using $\epsilon$-greedy policy from $Q(s_t, a; \theta)$
      Execute $a_t$, observe reward $r_t$ and next state $s_{t+1}$
      Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
      Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
      Set target $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-)$
      Perform gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$
      Every $C$ steps: Update target network $\theta^- \leftarrow \theta$
    **end for**
  **end for**

# 3. Experiments

## 3.1. Environment Setup

The goal is to program a system that uses a DQN to estimate the best possible action to take for each possible state that an agent finds itself in.
For this experiment, I use the CartPole-v1 environment from the Gymnasium toolkit (Gymnasium, 2023). The agent must balance a pole on a cart by moving the cart left or right. The environment is considered solved when the agent can balance the pole for 500 consecutive time steps.
In this implementation, I have a seperate class for the DQN and the agent, which get called in a seperate file.
The DQN class creates a basic neural network using PyTorch (PyTorch, 2024), with an input layer consisting of 4 neurons (one for each state variable), and an output layer consisting of only 2 neurons, for both possible actions (left and right), as well as a variable amount of hidden layers and neurons per layer.
The agent class contains the environment, DQN as well as target network, and the replay buffer.
Each training is run for a total of 1 million steps, as well as repeated 5 times and averaged to get a more stable result.

For the first experiment, a hyperparameter ablation study, I will test the effect of the learning rate, discount factor, number of DQN layers and neurons, as well as epsilon-decay and the update to data ratio on the training performance of the DQN by testing out 3 different values per parameter. A graph of the average and standard deviation of all 5 training runs will be presented, and the results discussed.

After this, I will enable the Experience Replay and Target Network functionality in the agent, and perform 5 trainings with only ER active, 5 with only TN active, and 5 with both active, while adopting the best hyperparameters from the ablation study. I will then present the results of these trainings all visualized in a single graph and discuss the results.

## 3.2. Results

An optimal training curve would first and foremost show a steady increase in the average reward over time, until every single run achieves the reward limit and the average reward matches the maximum reward of 500. Secondarily, the time it takes to reach this maximum reward would be as short as possible.
With this in mind, the results should now be inspected on the following pages.

The results show that the best performance for the naive model occur with a

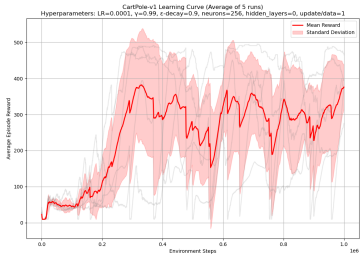
1. **Epsilon decay of 0.9**
2. **Learning rate of 0.0001**
3. **Discount factor of 0.99**
4. **Number of hidden layers of 0**
5. **Neurons per layer of 256**
6. **Data to update ratio of 1**

For the second experiment, the results show that the agent performs similarly with an experience replay than when no enhancement techniques are being applied at all. However, all variations with the target network enabled seem to lead to worsened results, with a slight improvement in initial learning speed and reaching an average of around 300 reward per episode, but then dipping and consistently floating at around 200 reward per episode.

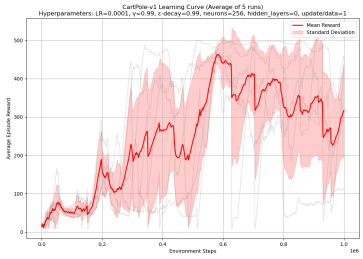
# 4. Discussion

My ablation study showed some interesting results about how DQNs work on CartPole-v1. A fast epsilon decay (0.9) worked best, probably because the environment is simple enough that we don't need much exploration. This makes sense since there are only two possible actions and four state variables.

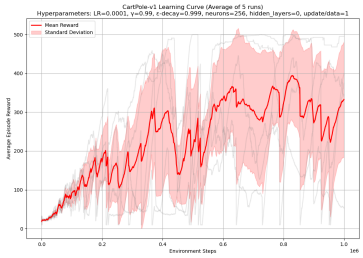
The learning rate of 0.0001 gave the best results - higher values made the training unstable, while lower values learned too slowly. The discount factor of 0.99 worked well, which is what most people use in reinforcement learning.



(a) epsilon decay = 0.9

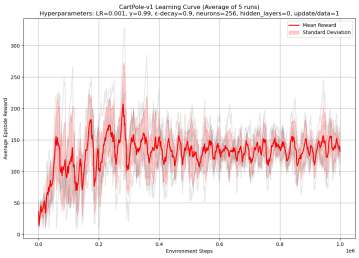


(b) epsilon decay = 0.99

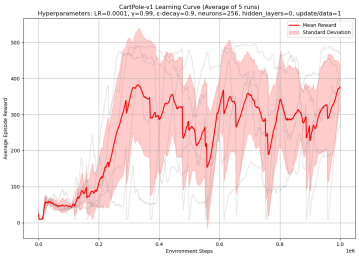


(c) epsilon decay = 0.999

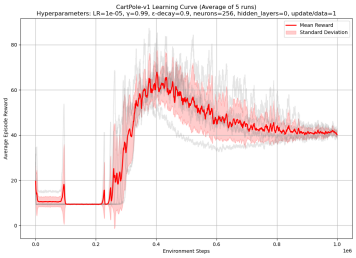
*Figure 1.* Impact of epsilon decay



(a) learning rate = 0.001

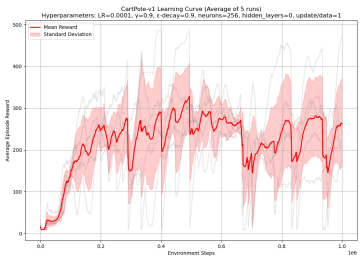


(b) learning rate = 0.0001

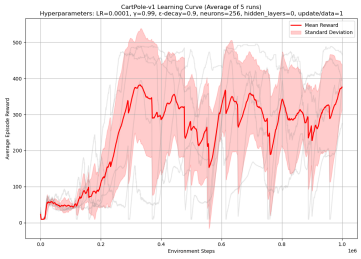


(c) learning rate = 0.00001

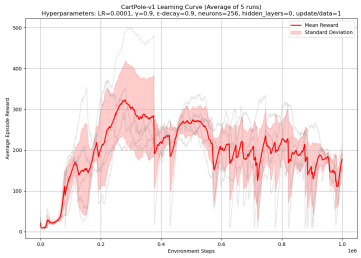
*Figure 2.* Impact of learning rate



(a) discount factor = 0.9



(b) discount factor = 0.99



(c) discount factor = 0.999

*Figure 3.* Impact of discount factor

The most surprising finding was that a network without hidden layers performed better than deeper ones. This suggests that CartPole-v1 might be simple enough that a linear approximation works fine. Adding more layers just made it harder to train.

Testing experience replay and target networks gave unexpected results:

1. Experience replay didn't help much, probably because CartPole-v1 is deterministic
2. Target networks actually made things worse, maybe because they slow down learning
3. Using both together wasn't helpful either

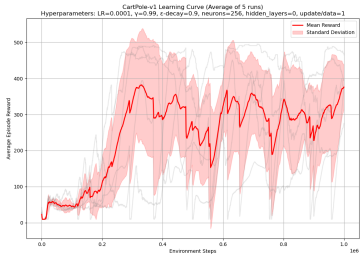
This shows that techniques that usually help in DQN might not always be necessary, especially for simple environments.

Limitations of this study include the focus on a single environment and the relatively short training duration. Future work could explore these techniques across multiple environments to better understand when each enhancement provides the most benefit.

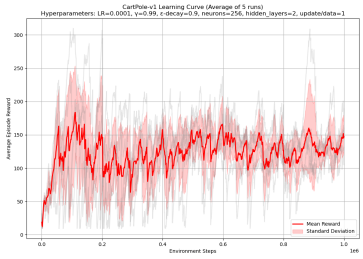
## 5. Conclusion

My implementation showed that DQN enhancements aren't always helpful. For CartPole-v1, a simple network without hidden layers worked better than more complex setups. The main findings were:

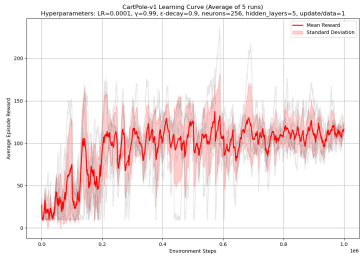
1. Simple networks can work fine for simple environments
2. Experience replay didn't help much in this case
3. Target networks actually made performance worse
4. Fast exploration (-decay = 0.9) worked best



(a) hidden layers = 0

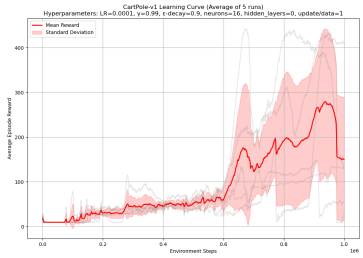


(b) hidden layers = 2

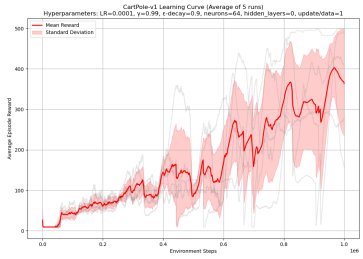


(c) hidden layers = 5

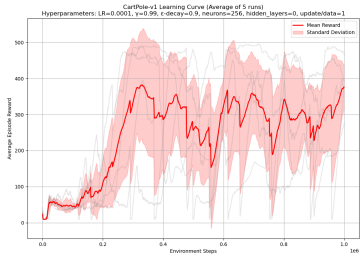
*Figure 4.* Impact of hidden layers



(a) neurons = 16

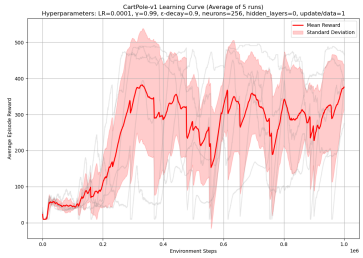


(b) neurons = 64

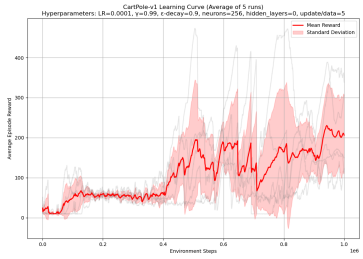


(c) neurons = 256

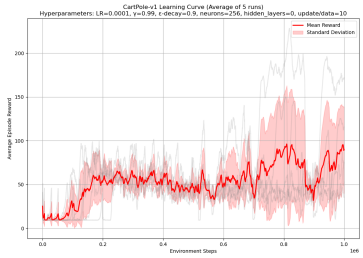
*Figure 5.* Impact of neurons per layer



(a) ratio = 1



(b) ratio = 5



(c) ratio = 10

*Figure 6.* Impact of update to data ratio

While experience replay and target networks are usually considered important for DQNs, my results suggest we should think about whether we actually need them for simpler tasks. Future work could test this on more complex environments to see when these enhancements become necessary.

My agent successfully learned to balance the pole, but more importantly, I learned when different DQN components are actually useful. This could help make better choices when implementing DQNs for different problems.

## References


Gymnasium. Gymnasium documentation. https://gymnasium.farama.org/, 2023. Accessed: 2024-03-18.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. 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.

Plaat, A. *Deep Reinforcement Learning*. Springer, 2022. ISBN 978-981-19-0930-4. doi: 10.1007/978-981-19-0931-1.

PyTorch. Pytorch documentation. https://pytorch.org/docs/stable/index.html, 2024. Accessed: 2024-03-18.

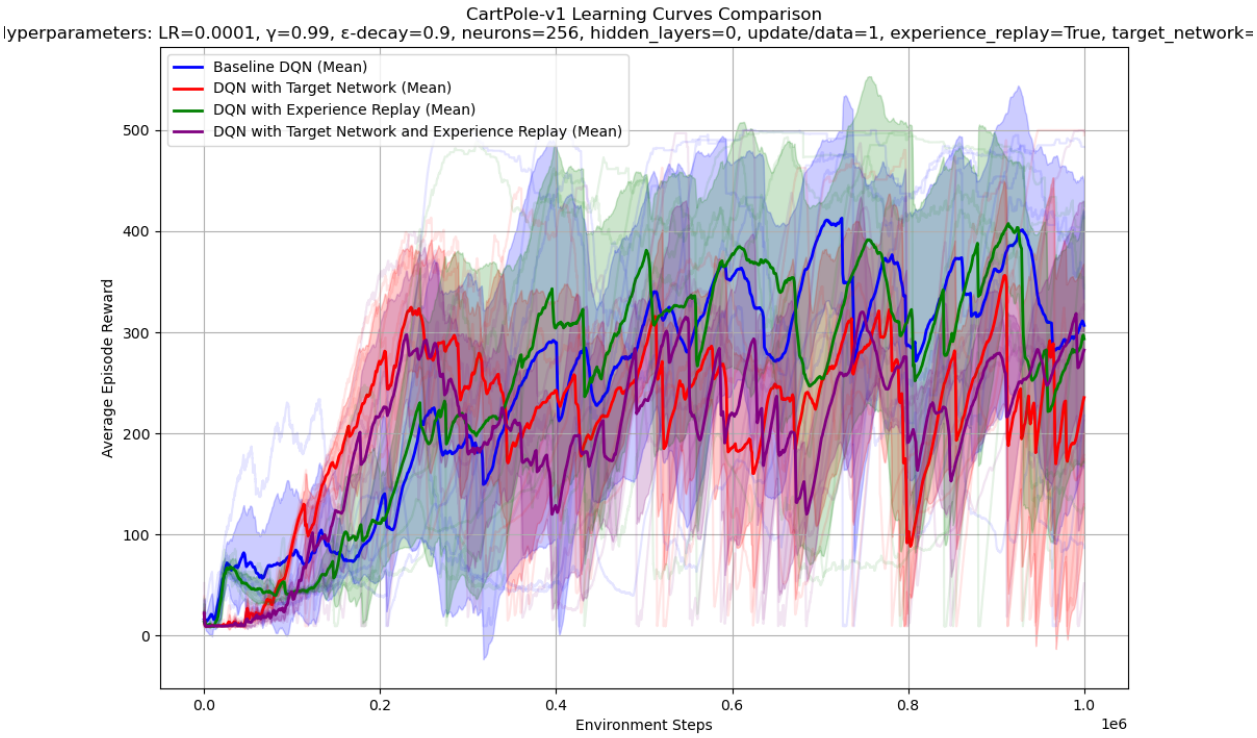



*Figure 7.* Performance of all models comparison