



Project Report

# Deep Q-Learning

Models of Higher Brain Function

Summer Semester 2024

Kevin Ferreira, kevin.ferreira@campus.tu-berlin.de

Reihaneh Taghizadegan, reihaneh@bccn-berlin.de

31.07.2024

@Version 1.0

# Table of Contents

# 1. Introduction

**Reinforcement learning** is a branch of machine learning where an agent learns to make decisions by performing actions in an environment to maximize rewards. It involves a dynamic interplay between an agent and the environment, requiring him to **explore** and **exploit** to improve his performance over time.

The aim of this project is to improve a basic reinforcement learning model through iterative improvement. Starting from a baseline, the goal is to identify key improvements by implementing various strategies and evaluating their effectiveness through a series of experiments.

This process will include adjusting hyperparameters, modifying the training loop, incorporating advanced techniques such as experiment replay and target networks, and testing various loss functions.

# 2. Problem statement

Reinforcement learning is an effective approach for training agents to perform tasks in structured environments. In this project, two distinct environments created using **OpenAI** Gym are studied: the *Linear Track* and the *Deadly Gridworld*. Each environment poses unique challenges and requires a thoughtful approach to state representation and action selection.

Indeed, **state representation** is crucial as it defines how the environment is perceived by the agent: a good state representation captures all the relevant information needed for the agent to make optimal decisions while an inadequate one can lead to poor performance due to either **missing important details** or **overwhelming the agent with unnecessary information**.

## a. Linear Track

The Linear Track consists of a linear grid of $3 \times 32$ pixels where an agent (represented in red) can navigate within walls (gray) while collecting rewards in the form of coins (yellow) or a diamond (turquoise): a coin represents a reward of +100 while a diamond represents a reward of +10 000. The goal of the agent is to maximize his final reward in **less than 50 moves**. In addition, once a set of rewards has been collected, the game stops. For that, the agent can perform three possible actions, denoted as $A = \{left, right, stay\}$.

The environment state can be represented using four binary pixel arrays of size $32 \times 4$. Each array focuses on a specific aspect of the environment:

- ➔ Array 1: Represents the agent's position (1 for occupied, 0 for empty).
- ➔ Array 2: Represents the presence of coins (1 for coin present, 0 for absent).
- ➔ Array 3: Represents the presence of the diamond (1 for diamond present, 0 for absent).
- ➔ Array 4: Represents empty spaces (1 for empty space, 0 for occupied by wall or agent/reward).

Then the completed representation is an array $s \in R^{3\times32\times4}$ that captures the complete visual information of the environment, including the agent's location, coins, and the diamond.

Figure 1: Linear Track Environment at initial position

One might assume that the **maximum reward** occurs when all but one of the coins and the diamond are collected, representing a total reward of +10 700, in a minimum of 40 moves. However, **this sequence of actions is unlikely**, since the agent has to go all the way to the left, only to return directly to the right. Collecting a reward only changes the state representation by one byte, not enough for a simple model to make any difference. So, let's assume that the maximum reward is to go straight for the diamond (+10 000). On the contrary, in the **worst-case scenario**, when the agent obtains no coins or diamonds after 50 moves, his total reward will be 0.

## b. Deadly Gridworld

The Deadly Gridworld is a more complex grid environment where the agent must navigate in a grid of $6 \times 14$ pixels, still collecting rewards but also avoiding poisons (brown) and a patroller walking from top to bottom and back (green). Here, a coin also represents a +100 reward and a poison -10. The agent's objective is also to maximize his final reward in less than 100 moves while avoiding the patroller, which in the event of a collision ends the game. In that case, the agent has four possible actions $A = \{up, left, right, down, stay\}$.

In that case, the environment state can be represented using five separate binary pixel arrays of size $6 \times 14$. Each array focuses on a specific aspect of the environment (

agent, coins, poison, patroller, empty space). Then, the state of the environment can be represented by a pixel array $s \in R^{6\times14\times5}$.

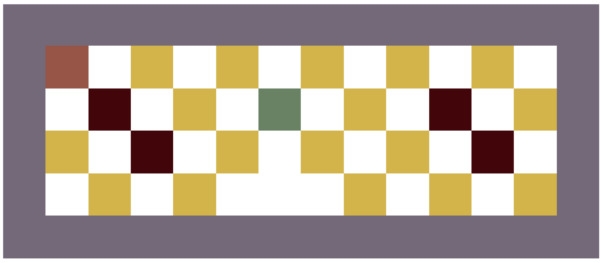

Figure 2: Deadly Gridworld Environment at initial position

For this environment, the **maximum possible reward** is when all the coins (+1 700) are collected in at least 34 moves. On the contrary, if in the **worst-case scenario** the agent collects the two poisons near the initial position without obtaining any coins after 100 moves, his total reward will be -20.

# 3. Baseline Implementation

The **Neural Network architecture** employed for the Deep Q-Learning is depicted in the connected layers in the figure 3:

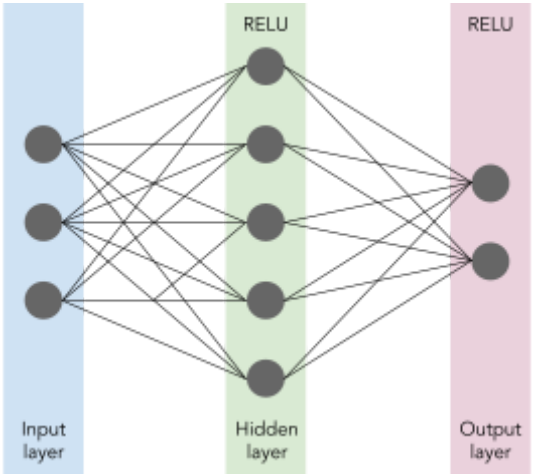


Figure 3: Composition of the Neural Network

It is composed of three main stages:

- An Input Layer that accepts the environmental state as a flattened vector, acting as the initial data entry point for the neural network.

- A fully connected Hidden Layer featuring 128 hidden units whose purpose is to capture complex relationships and patterns in the input data. Activation functions, specifically Rectified Linear Units (ReLU), are applied to introduce non-linearity, enabling the model to learn and represent more complicated functions.
- An Output Layer outputs a vector with a length equal to the action space, where each neuron corresponds to a potential action that the agent can take. The value outputted by each neuron represents the estimated value of taking that action given the current state as input to the network.

The first loss function chosen for training the model was the Mean Squared Error (MSE), which measures the average of the squares of the errors between predicted action values and target values.

A critical step during the training is to ensure that the target values for the MSE loss are detached from the graph of computations that track gradients. This detachment is vital because the target values function as fixed benchmarks against which the predictions of the model (the Q values) are compared. *If they were not detached, these target values would also be updated during the gradient descent steps according to the computed errors, leading to incorrect learning signals.* In essence, by detaching the target values, we prevent them from influencing the gradient calculations, avoiding a situation where the model tries to chase a moving target, and thus ensuring more stable and reliable learning. If this step is omitted, the agent would receive inconsistent feedback and likely converge to suboptimal policies, exhibiting erratic behavior and poor performance in decision-making tasks.

Moreover, the fundamental formula that guides the training of our model is the update of the Q-value, which is computed as follows:

$$Q(s_t, a_t; \theta_k) \leftarrow Q(s_t, a_t; \theta_k) + \alpha(t_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_k) - Q(s_t, a_t; \theta_k))$$

where $\theta_k$ are the parameters of the neural network at iteration k, $\alpha$ is the learning rate, $\gamma$ is the discount factor, $r_{t+1}$ is the reward at time t+1, and a' represents all possible actions.

At the start of the agent's deployment, performance metrics such as average return per episode and step per games were recorded. These metrics serve as a benchmark for measuring the learning progress and overall effectiveness of the agent.


Figure 4: Results of the baseline for the Linear Track and the Deadly Grid

While the agent in the Linear Track gets the maximum rewards, in the Deadly Grid, he fails to maximize his return and the game ends mostly after taking the maximal number of actions. In the rest of the report, we will discuss possible improvements to maximize the rewards of both environments as well as reducing the convergence times.

# 4. Methodology

Once the baseline had been established, we set out to optimize it by modifying parameters and adding features. These methods aimed to improve the learning process, enhance performance, and address specific environmental challenges.

We started with the basic algorithm explained in the previous paragraph. Then, we iteratively implemented several updates to enhance its performance: *Basic Experience Replay, Prioritized Experience Replay, Target Network, Loss Functions, Discount Parameters, Exploration Strategies, Neural Network Architecture.*

To ensure consistency, agents were initialized with the same random weights for neural networks during all the tests. Each agent was trained for a fixed number of episodes, with performance evaluated at regular intervals to monitor progress and effectiveness.

For each episode, data such as state transitions, actions taken, rewards received, returns and the number of steps taken to end the game were collected and stored in a JSON file. To test the model, we averaged the returns over 50 test games to ensure robustness in performance evaluation. The improved models were compared against the baseline in terms of the defined performance metrics to measure the impact of the enhancements. *For reasons of graphic representation, all diagrams of results for the Linear Track (respectively Deadly Grid) environment use a sliding average over 20 (respectively 200) games.*

# 5. Iterative Testing and Improvements

## a. Experience Replay Buffer

Experience Replay Buffer addresses the issue of sample inefficiency by storing a history of experience tuples $(s, a, r, s')$, where $s$ is the current state, $a$ is the action taken, $r$ is the reward received, and $s'$ is the next state. The Replay Buffer has a limited capacity $N_{replay}$, with older transitions being discarded as new ones are added. *The implemented ReplayBuffer class allows the agent to store a dictionary of transitions, each consisting of the state, action, reward, next state, and done flag.*

The buffer was configured with a capacity of 1 000 transitions and a batch size of 128, following the specifications in the provided instructions. Referring back to the parameters table and the code implementation provided, we chose a learning rate $\eta = 0.001$. This value sets a moderate pace for learning, aiming for a balance between sufficiently quick adaptation to acquired knowledge and avoidance of overshooting the minimum of the loss function which could result in unstable training. Also, we tried to implement a Prioritized Experience Replay, a technique that improves the agent's training by sampling according to the magnitude of the temporal difference error—transitions.

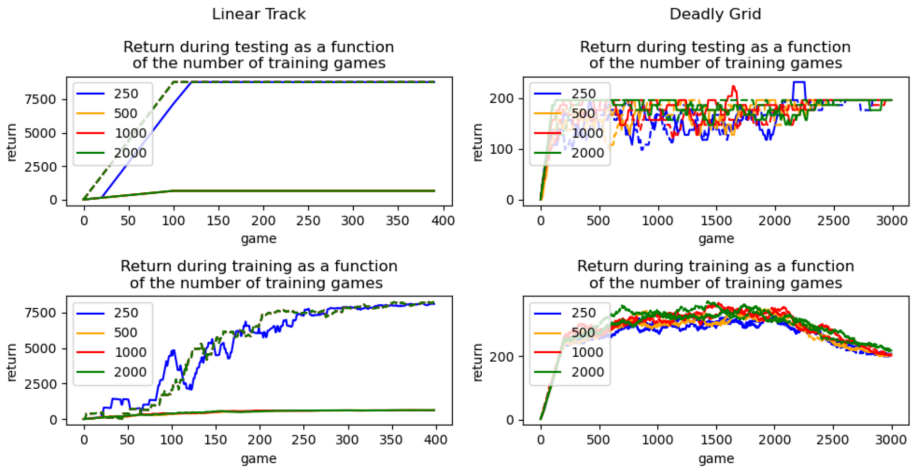


Figure 5: Results for different capacities of the replay buffer *(continuous lines)* and PRE *(non-continuous lines)*

What one can see in the figure 5 is that with a replay buffer, the convergence is smoother than for the baseline. Besides, whatever the capacity, the Prioritized Replay Buffer always gives better results leading to a convergence to the maximum returns. Indeed, while

the Experience Replay Buffer samples the experiences randomly, the Prioritized one gives more importances to the estimated q-value with a high error compared to the real one..

To conclude, with a buffer, the agent can sample mini-batches of experience, avoiding the rapid forgetting of past experiences and smoothing out the learning over a wider distribution of experiences.

## b. Target Networks

To make our Deep Q-Learning model more stable and help it converge better, we added a **target network**. The target network has the same architecture as the main network but with a separate set of slowly updating parameters, denoted as $\theta^{-}$. The update rule for the target values incorporates the target network to compute the Q-values for the next state $s_{t+1}$, leading to the following modified Bellman update formula:

$$y^{k}_{t,\,DQN} = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^{-})$$

The target network's parameters are updated to match the main network's parameters every $C = 500$ update steps, thereby ensuring consistent and more stable targets for each iteration of training. Also, for consistency, it is important to initialize the weight of the target neural network with the same weight of the network.

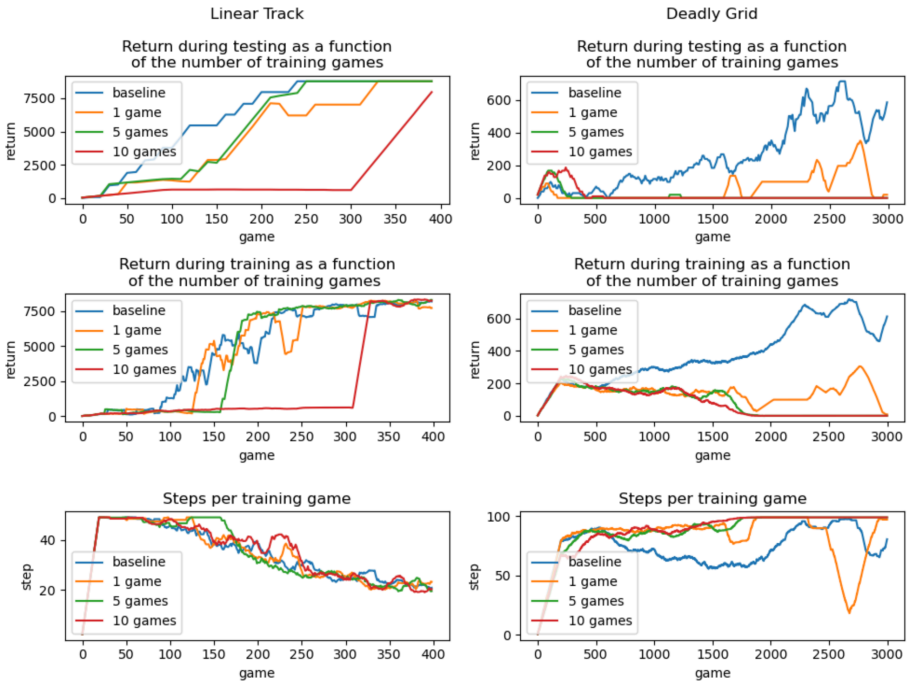


Figure 6: Results with differents update frequency of the target network

As can be seen, although the terminal state with target networks is similar to the network without them, it takes longer to learn and converge. However, it allows for a more gradual and stable approach to updating the target values used in the TD error computation, leading to less volatile learning curves and enhancing the performance of our agent. Target networks address the issue of moving target values, a common challenge in training deep Q-networks where rapid changes in policy can lead to divergence or oscillatory behaviors.

### c. Loss Function Comparison

To refine the learning process and address potential issues with sensitivity to large temporal difference errors, a comparison between the Mean Squared Bellman Error (MSBE) and the smooth L1 Huber loss function was implemented. The Huber loss function is represented by the following formula:

$$\mathcal{L}_{SmoothL1} = \mathbb{E}_{s_t,a_t,r_{t+1},s_{t+1}}[\begin{cases} 0.5(Q(s_t,a_t;\theta_k) - Y_k^t)^2, & \text{if } |Q(s_t,a_t;\theta_k) - Y_k^t| < 1 \\ |Q(s_t,a_t;\theta_k) - Y_k^t| - 0.5, & \text{else} \end{cases}].$$

This formula calculates the expected value of a loss that is quadratic for small differences between predicted Q-values and target values when the absolute error is less than 1 and linear for larger differences. In our implementation with PyTorch, the *torch.nn.SmoothL1Loss* function was employed to compute this loss, and the training process was updated accordingly. The results of this implementation were then compared to the previous MSE-based training. After integrating the smooth L1 Huber loss into the training routine, the loss values for each step were recorded alongside the agent's performance metrics.


Figure 7: Results of the MSE and L1 Hubber loss

As it can be seen in the Figure 7, the agent will learn how to reach the maximal returns faster with a MSE loss than a L1 Huber loss function.

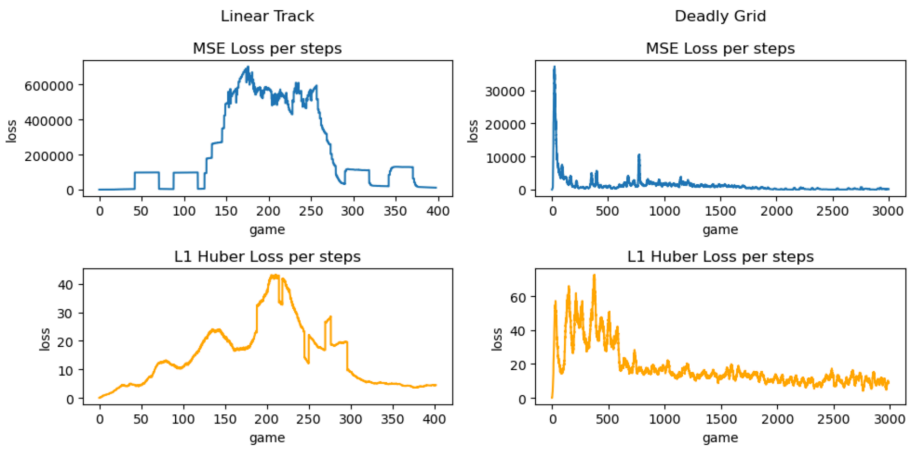


Figure 8: MSE and L1 Hubber loss

Indeed, in Figure 8, we can see that Huber loss L1 varies less than MSE: at each step, the improvement is smaller. It is then less sensitive to outliers and provides stability in training, improving the robustness of learning by not overly penalizing large prediction errors, which can be caused by noise transitions.

## d. Discount parameters influence

The discount factor γ plays a role in determining how future rewards are valued relative to immediate rewards, influencing the agent's decision-making process by balancing short-term gains against long-term benefits. In this section, we explore different interpretations and the impact of the discount factor on agent behavior by varying its value for the two environments. Following, one can see the cumulative rewards as well as the step per training for the differents value of the discount factor:

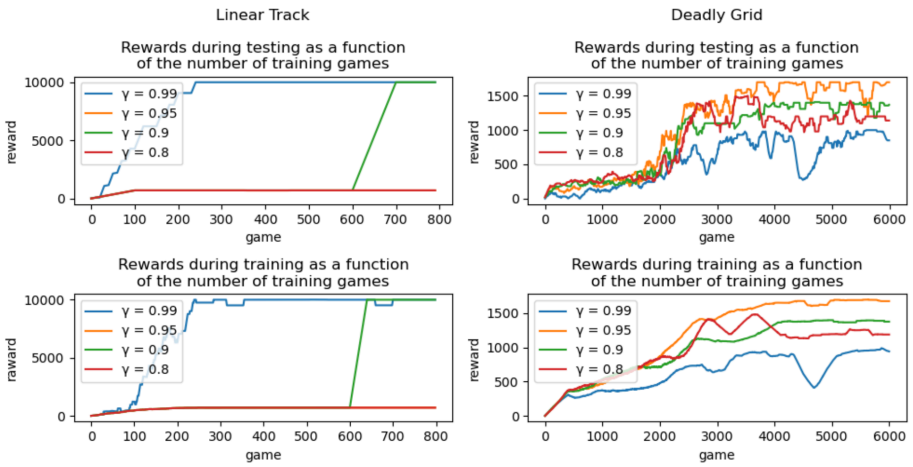


Figure 9: Results with differents discount value

As we can observe in the result and by re-playing some games, the discount factor influences convergence highly. This factor can be seen as a **planning factor**. Depending on the value of γ, the agent will consider long-term rewards to be more or less important than immediate rewards. With a high γ, future rewards are weighted highly, making them almost as important as immediate rewards. This weighting is beneficial when the environment requires a long-term strategy, as is the case on the linear track, where the best reward may be obtained after several moves. With a low γ, immediate rewards are more important than future rewards. This approach is advantageous in environments where immediate actions have a significant impact on success, as in the Deadly Grid where immediate action can end the game.

Thus, as expected, for the **Linear Track** environment, a high discount factor γ=0.99 leads to rapid convergence towards maximum rewards, whereas with γ=0.8 the agent does not learn. For the **Deadly Grid**, where both immediate and long-term actions are important, the agent achieved the best convergence with a discount factor of 0.95. This balance enabled the agent to evaluate future rewards while giving sufficient importance to immediate rewards, enabling it to navigate and make effective decisions in a complex environment. However, with a discount factor of 0.99, the agent took longer to converge and often failed to find the optimal strategy. The overestimation of future rewards in this environment led to over-cautious behavior, preventing the agent from effectively managing immediate threats.

### e. Exploration strategy

Initially, we implemented a baseline exploration strategy using **ϵ-greedy with linear decay**. This approach starts with a high exploration rate $\epsilon = 1$ and gradually decreases it to

0 over time. The idea is to initially explore the environment extensively and progressively shift towards exploitation as the agent becomes more confident in its learned policy.

To improve upon the baseline, we experimented with various exploration strategies. First, we tried the same ϵ-greedy with exponential decay. In this strategy, the exploration rate ϵ decays exponentially over time. We tested two different decay rates $\exp(10^{-5})$ and $\exp(10^{-3})$ This approach allows the agent to quickly reduce exploration at a fast rate initially and then slow down the decay as training progresses, maintaining a small level of exploration even in later stages. This helps prevent the agent from getting stuck in suboptimal policies by occasionally exploring new actions.

Moreover, the softmax policy, also tried, used a probabilistic approach to select actions. The probability of selecting an action a in state s is given by

$$P(a|s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a_i} \exp(Q(s,a_i)/\tau)}$$

where τ controls the level of exploration.


Figure 10: Results of the differents policies

One can see that for the Linear Track, the best strategy is the ϵ-greedy with an exponential decay with $\exp(10^{-5})$ rate converging faster to the optimal solution in 119 games. Indeed, the environment is straightforward, with limited action choices. Then, the agent does not need a lot of exploration to collect all the valuable information about state transitions and rewards. As training progresses, the reduced exploration rate allows the agent to focus on exploiting the learned strategies, refining its policy to maximize rewards. This balance helps the agent converge to an optimal policy without getting stuck in local optima.

In the Deadly Gridworld environment, our experimentation yielded nuanced results. Infact, while the fastest convergence is made by the softmax policy in 1500 games, it only converges toward a local maximum: the agent discovers a safe but not the most rewarding path and, due to the nature of the softmax policy, stick to this path, preventing it from exploring potentially more rewarding but initially riskier strategies.

However, the ϵ-greedy with an exponential decay with $\exp(10^{-3})$ takes more time to converge but leads to a higher final return. The exploration phase is longer due to the decay rate, which ensures that the agent tries a wide range of actions before settling into exploitative behavior: it helps the agent avoid local maxima and identify more optimal strategies.

The difference in performance between the softmax policy and the ϵ-greedy strategy with exponential decay highlights the importance of exploration in reinforcement learning, particularly in complex environments like Deadly Gridworld. While the softmax policy provides quick convergence, the ϵ-greedy strategy converge to more optimal strategies, resulting in higher final returns

## f. Neural Network

Now, we investigated the impact of using shallow neural networks with different numbers of hidden layers on the performance of reinforcement learning agents in the Linear Track and Deadly Gridworld environments.

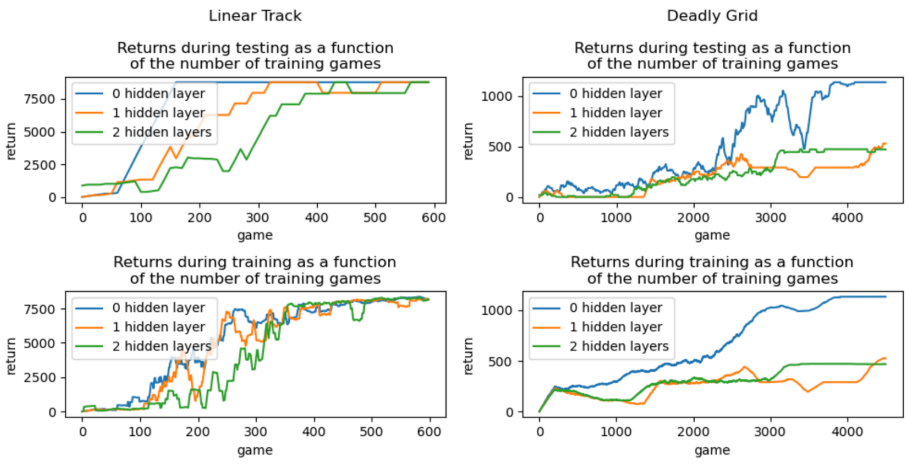


Figure 11: Results for different numbers of hidden layers

Surprisingly, we found that using a shallow network without hidden layers led to faster and, in some cases, better convergence.

For both environments, while the agent converged fast using a network without hidden layers, it was slow with two hidden layers, and in some cases, the added complexity did not lead to significant performance improvements, potentially causing overfitting.

So, both environments demonstrated that networks without hidden layers achieved faster and, in some cases, better convergence. Indeed, without hidden layers, the network functions as a linear model, which is easier to train and less prone to overfitting. For both environments, the critical state-action relationships were sufficiently linear, making the additional complexity of hidden layers unnecessary.

Now, let's examine the impact of varying the number of hidden units in the neural networks on the performance and convergence speed of the agents.

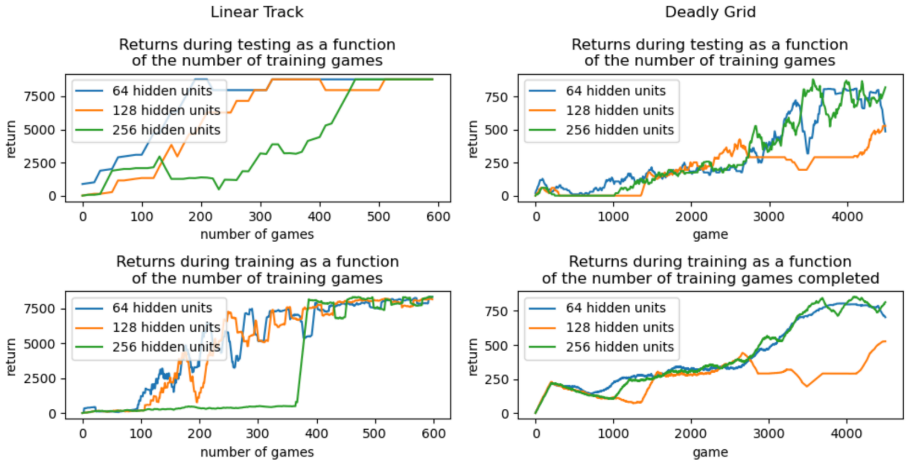


Figure 12: Results for different numbers of hidden units

Our findings highlight that simpler models, with fewer hidden units, often lead to faster and sometimes better convergence compared to more complex models.

Indeed, for the Linear Track environment, a network with a single hidden layer and 64 hidden units is optimal. It offers the fastest convergence while maintaining high performance. Increasing the number of hidden units does not improve the final performance but does slow down the convergence due to the added complexity. On the other hand, for the Deadly Gridworld environment, networks with 64 and 256 hidden units are optimal, providing both the fastest convergence and the best performance.

In conclusion, the simpler the model, the faster the convergence, as there are fewer weights to modify. What's surprising, however, is that simple models converge as well or better than more complicated ones. This may be due to the lack of iterations that allow the more complex model to converge.

# 6. Conclusion & limitation

In conclusion, we implemented a pipeline in order to improve the convergence of the baseline. What we have observed is as following:

- The use of Experience Replay Buffer smooths the agent's returns. Sampling the experiences using a prioritizing scheme also led to reducing the time of the convergence.
- While applying the Target Network had a similar terminal state with the network without it, it converged and learned slower. However, it enhances the learning process by making it smoother.
- The loss function has to be chosen according to the number of possible noisy states in the environment: while the Mean Square Error will lead to faster convergence, the Huber L1 Loss can handle more noise.
- The discount factor $\gamma$ influence the importance of the future rewards. If the best reward is obtained after several actions without rewards, this value has to be high.
- The selection of the exploration strategies has to be made according to the environment. The more complex the environment, the more the agent will have to explore.
- Adding layers and/or neurons to our network makes it more complex, leading to slower convergence, but also enabling us to manage more complex environments.

The results obtained are environment-specific: generalization of the results to different environments with different characteristics and complexities is therefore difficult. In addition, the experiments were carried out mainly on CPUs due to limited access to GPU resources. This limitation affected training speed and the ability to scale up experiments. The use of GPUs could significantly reduce training time and enable more in-depth tuning of hyperparameters and experiments especially for the Deadly Grid environment.

# Sources

[1] Ferreira Kevin & Taghizadegan Reihaneh (2024). **Deep Q Learning.** GitHub repository *(private)*. https://github.com/KFerrei/Deep_Q_Learning

[2] Paszke Adam & Towers Mark (2024). Reinforcement Learning (DQN) Tutorial. PyTorch tutorials. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html