Departamento de Computação/UFS

Optimization of Reward Wrappers in the Implementation of the Deep Q-Networks (DQN) Algorithm in the Atari Freeway Game

Otimização de Wrappers de recompensas na Implementação do Algoritmo Deep Q-Networks (DQN) no Jogo Freeway do Atari 2600

Heitor Saulo Dantas Santos [1]; Itor Carlos Souza Queiroz [1]; Lanna Luara Novaes Silva [1]; Lavínia Louise Rosa Santos [1]; Rômulo Menezes De Santana [1]

[1] *Departamento de Computação (DCOMP)*
Universidade Federal de Sergipe (UFS)
Av. Marechal Rondon, s/n– Jardim Rosa Elze– CEP 49100-000
São Cristóvão – SE– Brazil

heitor.santos@dcomp.ufs.br, itor_carlos@academico.ufs.br, lannaluara@academico.ufs.br, lavinialouise@academico.ufs.br, rmsantana@dcomp.ufs.br

This study investigates the optimization of reward wrappers in the implementation of the Deep Q-Networks (DQN) algorithm for the Atari 2600 game *Freeway*. Four DQN-based agents were trained: two using convolutional neural networks (CNN and CNN-R) and two with multilayer perceptrons (MLP and MLP-R), where CNN-R and MLP-R employed customized reward systems. The goal was to overcome the limitations of the original reward system, which only rewards complete street crossings, aiming to accelerate and make learning more efficient. The developed reward wrapper introduced intermediate rewards for progress and penalties for collisions, using game RAM data for precise state and event detection. The results showed that the customized system improved agent efficiency, although challenges remain in achieving adaptive behavior similar to that of a human player. For future work, the adoption of strategies such as multi-agent learning and recurrent neural networks (RNNs) is suggested to capture more complex temporal patterns and enhance decision-making.
Key-words: Reinforcement Learning, Deep Q-Network, Freeway, Reward.

1. INTRODUCTION

The field of training intelligent agents in digital games has been explored since the emergence of these games. Over the years, new techniques and models have been developed, including Reinforcement Learning and Neural Networks. The Atari 2600 Video Computer System is a video game console released in 1977 by Atari, featuring various games that have become a valuable resource for training models. In 2013, DeepMind introduced Deep Q-Network (DQN), applying Deep Learning to train agents in Atari 2600 games such as *Breakout* and *Pong* [1]. Given this context, the objective of this article is to present the training data of an intelligent agent designed to play the game *Freeway* on the console, using DQN, Convolutional Neural Networks (CNN), and Multilayer Perceptron (MLP) to optimize the reward system.

The selected game for this study was *Freeway*, in which the objective is to control a chicken crossing a road linearly while avoiding automobiles. A known issue with *Freeway*'s reward system is that its mechanics hinder Reinforcement Learning (RL). RL relies on rewards and penalties to learn the optimal policy [2]; however, in the game, the chicken only receives points after fully crossing the road and does not incur penalties in its score when colliding with a vehicle. As a result, the reward is delayed since it depends on a sequence of correct actions, and during training, the scoring model does not reflect this path effectively. Nevertheless,

Freeway operates in a deterministic and static environment, meaning there are specific patterns to be exploited. Acknowledging this, the experiments presented in this article aimed not only to explore more direct and intuitive training model alternatives but also to develop a customized reward system that addresses the identified challenges.

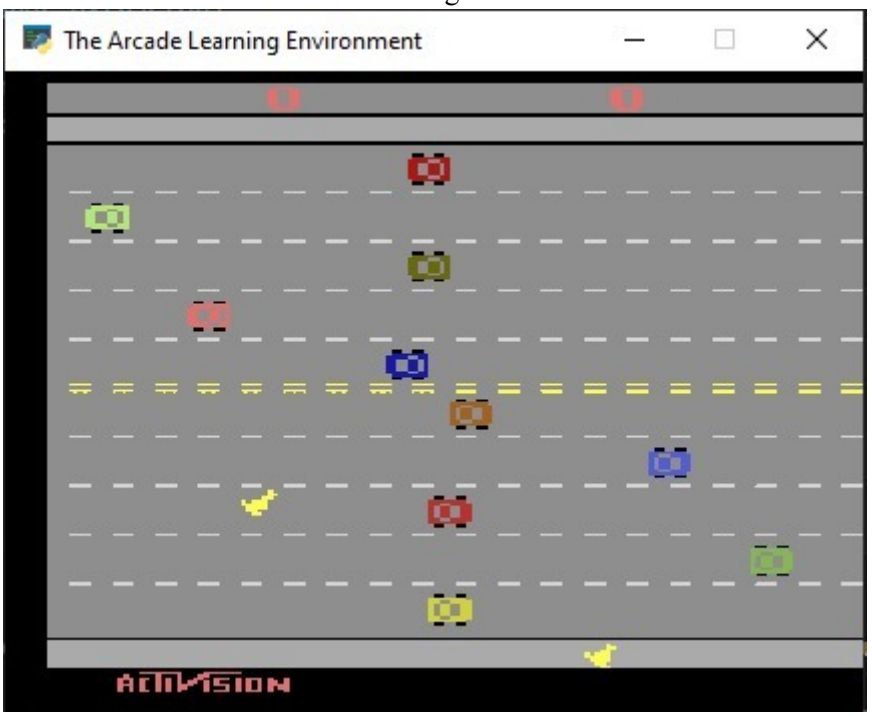

Figure 1: Environment of Freeway

The chosen approach involved using two Deep Q-Network (DQN) models with different policies for comparison and a more thorough performance analysis. The first policy is a fully connected neural network using Multilayer Perceptron (MLP), while the second policy employs Convolutional Neural Networks (CNN). The developed reward system was implemented and applied to both policies, and the experimental results will be presented throughout the paper.

The following section presents the detailed methodology and implementation specifications that supported the experiments. Section 3 describes the data and specifications of the conducted experiments. Section 4 presents the results, comparisons between the models, and a detailed analysis of the obtained solutions. Finally, the experimental conclusions and possible future improvements are discussed.

2. METHODOLOGY

The methodology used was based on the training of four intelligent agents: DQN with a CNN policy and the original reward system (**CNN**), DQN with an MLP policy and the original reward system (**MLP**), DQN with a CNN policy and a customized reward system (**CNN-R**), and DQN with an MLP policy and a customized reward system (**MLP-R**). To facilitate reference to the agents throughout the article, they will be identified using the highlighted acronyms. The following sections will present a description of each agent, the details of the customized reward system, an overview of the frameworks and libraries used, and finally, a description of the hardware employed for training.

2.1 Agent Specifications

All agents were implemented using the Deep Q-Network (DQN) architecture, which combines Q-learning with deep neural networks to approximate the Q-value function,

estimating the expected future reward for each state-action pair. Training is based on the Bellman equation, minimizing the difference between the predicted Q-value and the target Q-value [3]. The implementation was done using the Stable-Baselines3 library, which includes techniques such as experience replay (storing past transitions for stability) and ε-greedy (balancing exploration and exploitation) [4].

Two neural network policies were used in the DQN: Convolutional Neural Network (CNN) and Multilayer Perceptron (MLP). CNN was chosen for its ability to process visual data, such as the frames of the Freeway game, extracting spatial patterns directly from pixels. The implemented architecture is NatureCNN, proposed by Mnih et al. (2015) [3], consisting of three convolutional layers (32 filters 8x8, 64 filters 4x4, and 64 filters 3x3, with strides 4x4, 2x2, and 1x1, respectively), followed by a dense layer of 512 neurons and a final linear layer to map actions (move up, move down, or stay still). All layers use the ReLU activation function.

MLP, being simpler than CNN, maps pre-processed observations (such as the agent's position and distance to cars) directly to actions. Its architecture consists of two fully connected layers (64 neurons each, with ReLU) and a final linear layer for actions. Although it also works for images, it is a more efficient policy when the input is not visual [5].

During agent training, two reward systems were tested: the original reward system from the Freeway game and a customized reward system, which will be explained later in more detail. In general, this new version includes penalties for collisions and intermediate rewards for progress in the correct direction. These two types of systems were used to compare agent performance based on these scores, aiming to identify how to accelerate the agents' convergence to more efficient policies.

2.1.1 DQN with CNN Policy and Original Reward System (CNN)

This agent was the first to be developed. It utilizes DQN with the standard CNN policy, where the agent initially relies on randomness to identify environmental patterns and begin learning. Below is the pseudocode for the model along with the parameters used.

```
# Create the Freeway environment using Gym Atari
# Initialize the model with the following parameters
ModeloDQNCNN(
    Learning Policy: Cnn Policy
    Training Environment: Freeway environment
    Learning Rate: 0.0005
    Replay Buffer Size: 100000
    Steps Before Learning Starts: 1000
    Training Batch Size: 32
    Discount Factor: 0.99
    Target Network Update Interval: 1000
    Training Frequency: every 4 actions
    Output Detail Level: 1
)
save_model(file)
train_model(total_steps)
```

2.1.2 DQN with CNN Policy and Customized Reward System (CNN-R)

Like the previous agent, this one also uses DQN with a CNN policy, but with a customized reward system developed during the experiments. This system will be presented in detail in Section 3 - Experiments. A vectorized environment was also used for optimization, which will be detailed further in section 2.4. Below is the pseudocode for the model, including the parameters used and the customized reward system.

```
# Create the Freeway environment using Gym Atari
```

```
      FreewayRewardWrapper(environment)   # Apply the customized reward
system

      # Initialize the model with the following parameters
      ModeloDQNCNN(
          Learning Policy: Cnn Policy
          Training Environment: Freeway environment
          Learning Rate: 0.0003
          Replay Buffer Size: 50000
          Steps Before Learning Starts: 50000
          Training Batch Size: 64
          Discount Factor: 0.78
          Target Network Update Interval: 1000
          Training Frequency: every 4 actions
          Exploration Factor: 0.12
          Final Exploration Factor: 0.04
          Output Detail Level: 1
          Training Device: GPU, if available
      )

      train_model(total_steps, reward_callback)   # Train with callback
to store the average reward per episode
      save_model(file)
```

2.1.3 DQN with MLP Policy and Original Reward System (MLP)

This agent was implemented using a Deep Q-Network (DQN) architecture with a policy based on a Multilayer Perceptron (MLP). The original game reward system was adopted. The pseudocode for this agent follows the same structure and parameter values as the CNN agent described in Subsection 2.1.1, with the only difference being that the learning policy is changed to "MlpPolicy".

2.1.4 DQN with MLP Policy and Customized Reward System (MLP-R)

This agent also used DQN with an MLP policy but with the customized reward system that was developed during the experiments and will be detailed further later. The pseudocode for this agent follows the same structure and parameter values described for the CNN-R agent in subsection 2.1.2, with the only change being the learning policy set to "Mlp Policy.". Like CNN-R, a vectorized environment was also used.

2.2 Description of Frameworks and Libraries

For training, Python 3.13.0 was used as the programming language. Several libraries and frameworks were utilized during the implementation. There was no need to use a predefined Freeway dataset, as the agent was trained interactively, learning through trial and error within the environment. Below are the libraries and frameworks used:

a) **gymnasium [6]:** A library used to create and interact with reinforcement learning environments; it provides pre-built Atari game environments.

```
import gymnasium as gym # Importing the library
```

b) **stable_baselines3 [4]:** A framework implementing deep reinforcement learning (Deep RL) algorithms; it provides DQN and its CNN and MLP policies.

```
!pip install stable_baselines3  # Installation
from stable_baselines3 import DQN  # Importing DQN
from stable_baselines3.dqn import CnnPolicy
from stable_baselines3.dqn import MlpPolicy
```

c) **ale_py [7]:** Provides the interface for Atari games, enabling gymnasium to use them

```
import ale_py
```

d) **numpy, time, os, matplotlib.pyplot:** Python libraries used as support during implementation.

e) **torch**: Provides support for tensor operations and GPU-accelerated computation.

2.3 Description of the Hardware Used

To better analyze the performance of the trained intelligent agents, different hardware configurations were used. Throughout the article, the machines will be referenced according to the specifications listed in Table 1.

Table 1: Hardware Specifications

	Hardware		
Identificador	**RAM**	**CPU/GPU**	**Os/Environment**
Machine 1	16 GB	Intel Core i5-9300H	Windows 10
Machine 2	12 GB	12 GB (VRAM)	Google Colab
Machine 3	16 GB	Apple Silicon M1 (MPS)	MacOS

2.4 Use of the Stable-Baselines3 Framework for Environment Vectorization

As mentioned in section 2.1, agents using the customized reward system were implemented with a vectorized environment. For this approach, the classes VecFrameStack and DummyVecEnv from the Stable-Baselines3 framework were implemented. Vectorized Environments are a technique used to stack multiple independent environments into a single environment. Instead of training a reinforcement learning (RL) agent in only one environment per step, this approach allows it to be trained simultaneously in N environments per step. As a result, the actions sent to the environment are represented as an N-dimensional vector, which also applies to observations, rewards, and episode termination signals (dones). Vectorized environments are essential for applying wrappers used in techniques such as frame-stacking and normalization. Additionally, when using this approach, environments are automatically reset at the end of each episode. Thus, the observation returned for the i-th environment, when done[i] is true, will be the first observation of the next episode rather than the last observation of the recently completed episode. If it is necessary to access the actual final observation of the terminated episode—i.e., the one associated with the done event generated by the underlying environment—it can be retrieved through the terminal_observation key present in the info dictionaries returned by VecEnv [8]. The code execution was observed to be approximately 10 times faster compared to solutions without environment vectorization and without applying the DummyVecEnv constructor. Additionally, it is possible to override and customize methods from these classes.

As previously explained, using vectorized environments allows multiple instances of the same environment to be executed in parallel, optimizing the sample collection process during training. Thus, each interaction step covers multiple observations, reducing the total

training time. Construction methods such as make_vec_env or directly DummyVecEnv create these parallel environments in a simple way. When integrated with DQN, PPO, or A2C models from the stable_baselines3 library, they facilitate synchronous or asynchronous data sampling, enabling better hardware utilization. Vectorization also allows callbacks to obtain statistics from simultaneous episodes, assisting in the analysis and monitoring of the agent in complex scenarios. Each environment generates rewards and states that are replicated for the network, optimizing batch training usage and making the process more robust and scalable.

3. EXPERIMENTS

As mentioned, the experiments were conducted with each agent running on the three machines described in section 2.3 Hardware Description. The following sections will detail each metric used and the performance comparisons between executions. Next, the rewards used, the parameters affecting the training, and the process utilizing RAM positioning for the customized reward will be detailed.

3.1 Reward Value Used

During training, reward values can be greater than zero—indicating a positive reward—or less than zero, indicating penalties for a poor action. Each of these values influences the agent's learning and performance as it trains in the game environment. Agents that used the game's original reward system only receive +1 when crossing the street, whereas the developed reward system assigns values to other actions. The identified actions and their respective implemented rewards are shown in Table 2 below:

Table 2: Rewards

Id	Rewards				
	Action Description	Value			
		CNN	MLP	CNN-R	MLP-R
Action1	Chicken moved forward and did not collide	0	0	+0.25	+0.5
Action 2	Chicken moved forward and returned, with collision	0	0	-0.95	-0.95
Action 3	Returned (with or without collision)	0	0	0	0
Action 4	Successfully crossed	+1	+1	+10	+3.5
Action 5	Stayed still	0	0	-0.035	-0.50

3.2 Training Parameters

Training parameters are essential configurations that guide the learning process of models, directly influencing the efficiency, stability, and final performance of the agents. Below are the main parameters used:

a) **learning_rate:** Defines the magnitude of the adjustments made to the neural network weights during training, balancing the speed and stability of convergence.
b) **buffer_size:** Determines the capacity of the replay memory, which stores past experiences for sampling during training, promoting data diversity.
c) **learning_starts:** Specifies the number of initial steps in which the agent collects experiences without updating the network, ensuring a sufficient initial buffer for sampling.

d) **batch_size:** Defines how many experiences are sampled from the buffer for each network update, influencing training efficiency and stability.
e) **gamma (discount factor):** controls the importance of future rewards relative to immediate ones, being essential for calculating the expected value of actions.
f) **target_update_interval (intervalo de atualização da rede alvo):** Defines how often the target network is updated, helping stabilize training by reducing the volatility of Q-value estimates.
g) **train_freq (frequência de treinamento):** Determines how many environment steps are taken between each network update.
h) **exploration_fraction (exploration fraction) and exploration_final_eps (final epsilon value):** Control the ε-greedy exploration policy, defining the gradual reduction of the exploration rate over time.
i) **verbose (verbosity level):** Adjusts the amount of information displayed during training, facilitating process monitoring.
j) **total_timesteps (number of steps):** Total number of steps in the environment during training. It defines the agent's exposure to different situations, influencing the robustness of the learned policy. A higher number allows for broader exploration but increases computational cost. A lower number may limit learning but is more resource-efficient.

These parameters were carefully configured and tested with the goal of optimizing the agents' performance in the game Freeway. The parameters, already presented in the agents' pseudocodes in section 2.1 Agent Specifications, are detailed to facilitate the understanding of the impact of the implemented values in both versions of the systems, providing a clearer analysis of their functions and effects on training.

3.2.1 Agents with the Original Reward System (CNN and MLP)

In the implementation of both agents using the original reward system, the same parameter values were defined to compare the CNN and MLP policies under identical training conditions. The learning_rate was set to 0.0005, a relatively low value that allows for smooth weight adjustments in the network, ensuring stability during learning. The buffer_size of 10,000 defines a large replay memory capable of storing a significant amount of experiences, which helps diversify samples and prevent overfitting. The learning_starts of 1,000 ensures that the agent collects sufficient data before starting training, and the batch_size of 32 balances efficiency and stability in network updates. The gamma of 0.99 prioritizes future rewards, encouraging the agent to adopt long-term strategies. The target_update_interval of 1,000 steps determines the frequency of target network updates, reducing the volatility of Q-value estimates. The train_freq of 4 updates the network every four steps, maintaining a balance between exploration and learning. Finally, the verbose of 1 enables moderate monitoring of the training progress. These parameters were chosen to promote stable and efficient training for both policies.

3.2.2 Agents with a Custom Reward System (CNN-R and MLP-R)

Just like in the agents using the original reward system, the parameters for agents with a custom reward system were kept the same for both policies (CNN-R and MLP-R). The learning_rate of 0.0003 was chosen to allow faster learning compared to the previous value, adapting to the custom rewards. The buffer_size of 50,000 increases the replay memory capacity, storing more experiences and improving sample diversity. The learning_starts of 50,000 ensures that the agent collects a significant amount of data before starting training. The batch_size of 64 increases the number of experiences used in each network update, promoting greater learning stability. The gamma of 0.78 reduces the importance of future rewards, reflecting the prioritization of short- and medium-term custom rewards. The target_update_interval, train_freq, and verbose follow the same standards and values as the

agents with the original reward system. The exploration_fraction of 0.12 and exploration_final_eps of 0.04 adjust the ε-greedy exploration policy, reducing the exploration rate more quickly to focus on already learned strategies. These parameters were configured to optimize the performance of agents with custom rewards.

3.3 Custom Reward and RAM Positioning

Initially, the models were trained without reward Wrappers, using only the score provided by the game for each successful crossing of the road. This approach produced inconsistent results, as the agent relied solely on random exploration to maximize its return, without considering collisions with cars as a negative event. Subsequently, various improvements were made to the Freeway reward system.

The custom reward system utilizes the study of collision positions and RAM positioning to map the game states and adjust reward patterns accordingly. This approach was inspired by a repository that analyzes RAM positioning in Atari games [8]. The collision position refers to the location where the chicken collides with a vehicle and moves back (Action 2). This section will present in detail the process of developing this approach. Below is the pseudocode for better understanding:

```
FreewayRewardWrapper():
    initialize reward and player position to 0
    function reset(current environment state):
        initialize RAM #self.env.unwrapped.ale.getRAM()
        initialize last score from ram[103]
        initialize last chicken position from ram[14]
        clear current episode rewards
        return current environment state
    function reward(ram, action):
        initialize chicken position from ram[14]
        initialize score from ram[103]
        initialize cars with the position range of cars in RAM
        reward = 0
        lane position = verify_car_lane()
        if chicken position > last position # moved up
            reward = reward + 0.25
                    if (check_collision(cars, lane position) or
check_collision_with_action and current position < last position)
            reward = reward - 0.95 # collision detected
        if current position = last position
            reward = reward - 0.035 # remained stationary
   # amplify score impact
        if current score > last score
            reward = (score - last score) * 10
        update previous states (score and position)
        return reward
```

The main logic for reward evaluation is centered in the FreewayRewardWrapper class, where the verify_car_lane function maps the RAM position (ram[14]) to one of the specified traffic lanes, and the check_collision and check_collision_with_action functions penalize actions that fail to avoid collisions or do not contribute to scoring progress. The customization of the reward system reinforces positive advancements (such as moving up the screen) and discourages stagnation and collisions. When the agent scores (ram[103] increases), the reward is amplified, balancing motivation for correct movements and penalties for incorrect decisions.

The first attempt at collision detection focused on observing the agent's actions and position. If the agent moved backward without an explicit command to do so, the event was interpreted as a collision. Using RAM data was essential for obtaining the agent's y-position and

the x-coordinates of the cars. However, every time the agent crossed the street and returned to the starting point, this mechanism mistakenly computed the event as a collision, penalizing correct actions. After analyzing these inconsistencies, more precise collision detection methods were examined, identifying the agent's x-position at a specific address (25) in RAM, which allowed correlating the agent's location to the y-position of each highway lane. This way, the exact moment when collisions occur could be detected more reliably. The combination of these strategies resulted in a more efficient detection system, improving the model's training by balancing progress and penalties more robustly.

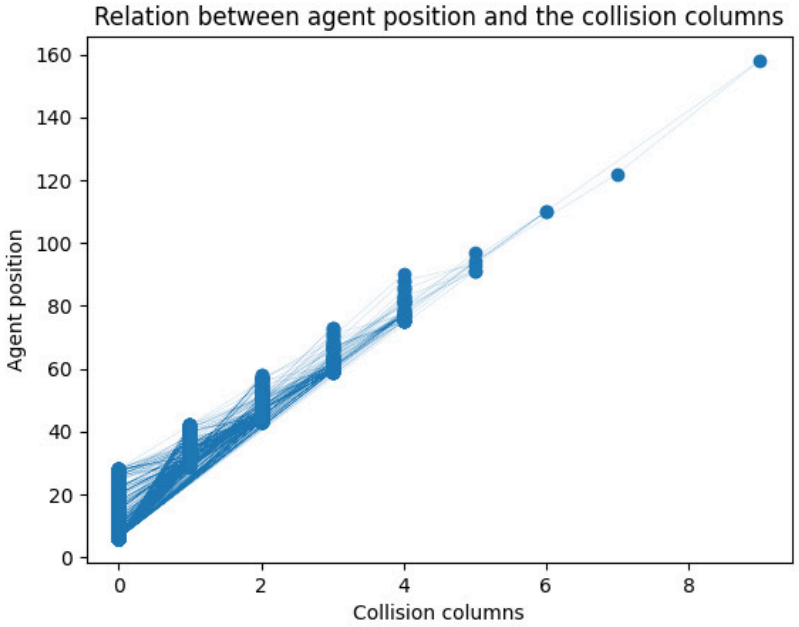

Figure 2: Graph of the agent's position during collisions

4. Results and Discussion

4.1 Execution Results of Models on Each Machine

Table 3 presents the execution results of training on different machines, where *p* represents the number of steps executed and *t* the total training time. For consistency and comparison, the tests in the table below were standardized to 100,000 steps. Machines labeled as "without test" indicate that the corresponding agent was not tested on that machine.

Table 3 Model on each Machine

Agent	Execution of Agents					
	Machine 1		Machine 2		Machine 3	
	p	t	p	t(s)	p	t(s)
CNN	100.000	14760	100.000	563	100.000	5400
MLP	100.000	11232	100.000	394	100.000	1200
CNN-R	*without test*	*without test*	*without test*	*without test*	100.000	1080
MLP-R	*without test*	*without test*	*without test*	*without test*	100.000	43

4.2 CNN Performance Result

The DQN with a CNN policy and the original reward, when executed on machines with 100,000 steps, had execution times as shown in Table 3. The duration was within the expected range, considering each machine's hardware specifications. The chicken moved forward, collided, returned, and crossed the street. After some training time, it was observed that it became smarter and crossed the street more frequently; however, it eventually returned to making the same mistakes as at the beginning of the training. The maximum score, visually evaluated using the original reward and the game's scoring system, was 24 points. In the following graph, the evolution of rewards during training in each episode is presented. It is observed that at the beginning, when the agent is learning, the reward is 0, but between episodes 20 and 30, it starts learning quickly and shows a growing trend in performance, although it still declines at times due to the errors mentioned above. The orange line represents the average of the rewards obtained.

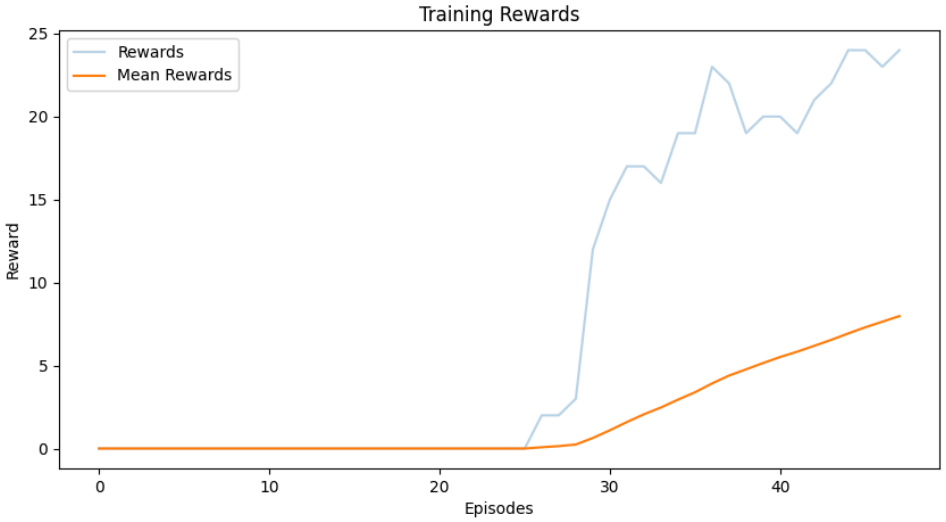

Figure 3: Reward Graph in Standard CNN Training

4.3 CNN-R Performance Results

The DQN with a CNN policy and a customized reward, when executed with 600,000 steps, did not achieve the expected results. Adjustments were made to the rewards and penalties, along with modifications to the exploration rate, yet the agent learned to use only action (moving forward) until crossing the street. Despite multiple collisions during this process, the agent was able to score (fully cross the street) considerably well. The following graph shows the evolution of rewards during the training of CNN-R. The blue line represents the rewards per episode, with significant variation over time, and the orange line indicates the average rewards, which gradually increase until stabilizing around the 50th episode.

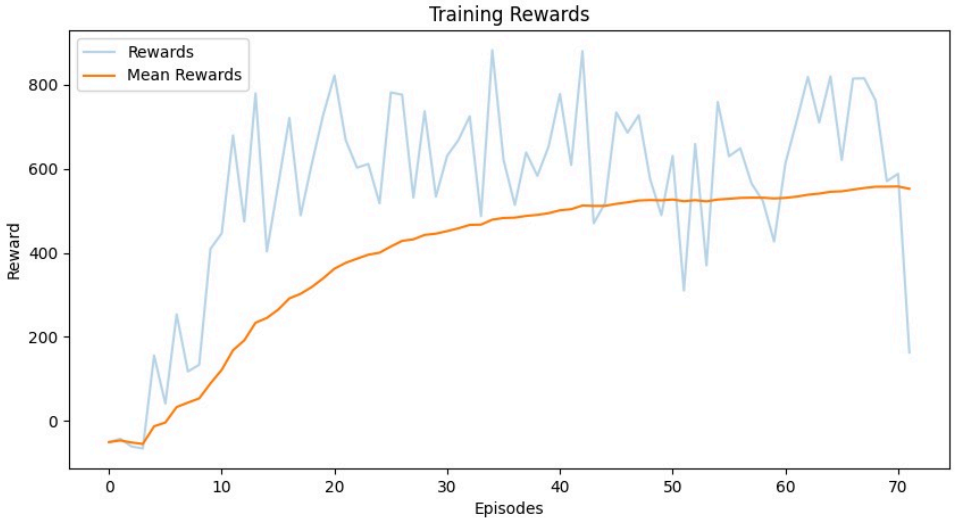

Figure 4: Reward Graph in Customized CNN training

4.4 MLP Performance Results

The DQN with an MLP policy and the original reward, when executed with 100,000 steps, had a training time of 6 minutes and 34 seconds on Machine 1 and 2 hours and 53 minutes on Machine 3. The duration was within the expected range, considering each machine's hardware specifications. The chicken behaved similarly to the CNN training but trained considerably faster on both machines. Other executions also demonstrated this increase in speed when training with MLP. The maximum score, visually evaluated using the original reward and the game's scoring system, was 6 points.

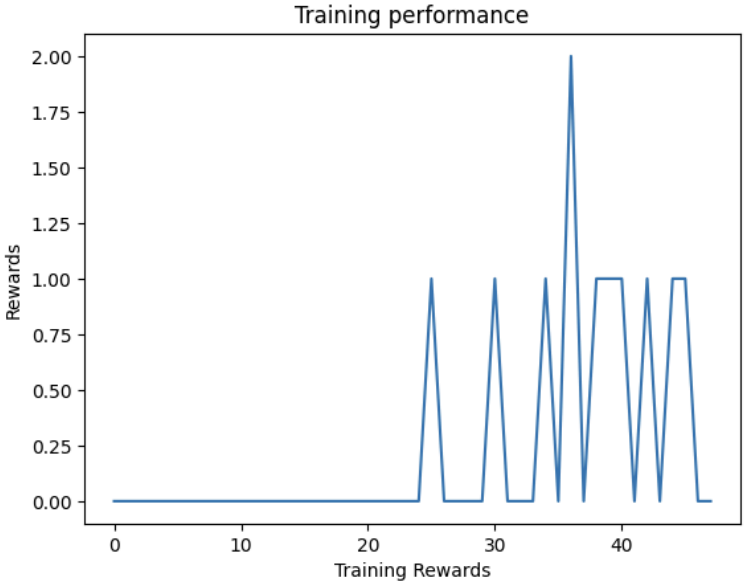

Figure 5: Reward graph in standard MLP training

4.5 MLP-R Performance Results

The DQN with an MLP policy and a customized reward, when executed with 100,000 steps, had a training time of 43 seconds on Machine 4, using vectorized environments to speed up the training process. The duration was within the expected range, considering each machine's hardware specifications. However, the agent's behavior was not successful: it moved up and

down multiple times but barely left its place and did not cross the street. The following graph displays the rewards during the training of MLP-R. The blue line shows abrupt variations in the final episodes, while the orange line (average rewards) exhibits gradual growth until stabilizing. Despite the late improvement, the agent struggled to learn, with performance below expectations.

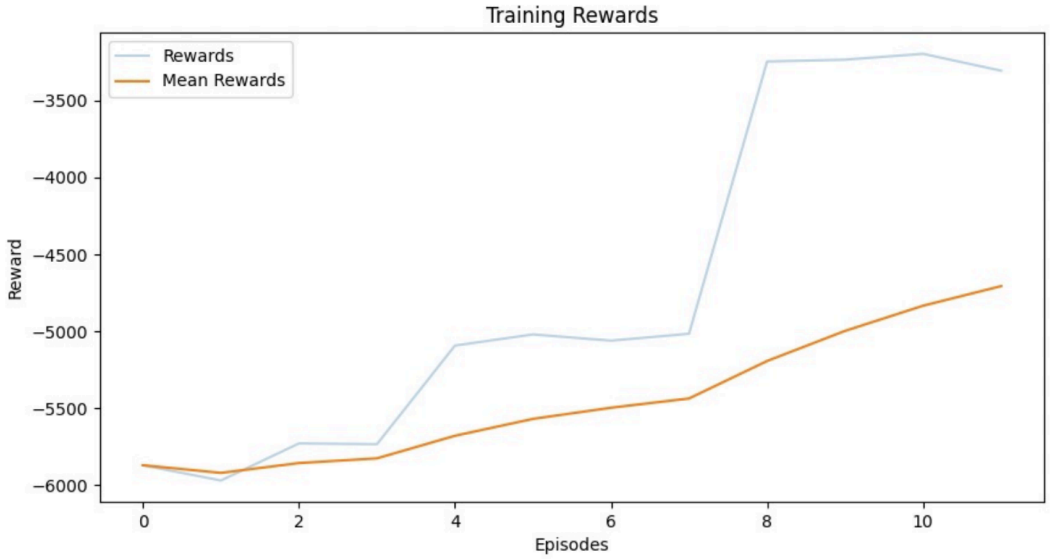

Figure 6: Reward Graph in MLP Customized Training

5. Conclusion

The experiments conducted demonstrated that modifying the reward system had a significant impact on the agent's learning in the Freeway game. Implementing the custom Reward Wrapper accelerated the learning process by providing more granular feedback on the agent's actions. However, the results indicate that while reward adaptation improved training efficiency, considerable challenges remain in achieving behavior similar to that of a human player.

The main difficulty encountered was balancing rewards and penalties to encourage a more natural and efficient strategy. The tests showed that, in some cases, agents developed suboptimal behaviors, such as persistently advancing without considering collisions or hesitating to take strategic actions. This limitation reinforces the need for fine-tuning reward weights and possibly combining different reinforcement learning techniques, such as hierarchical learning or imitation learning.

Additionally, analyzing the RAM of the Freeway game revealed valuable insights into the game's internal dynamics, enabling the exploration of alternative approaches. Using this data to infer richer and more detailed states could be a promising path for improving agent performance. In the future, more advanced feature extraction techniques and reward engineering strategies may be employed to enhance the adaptability and efficiency of agents in different scenarios.

The execution times of the experiments also highlight the impact of the modifications. In the standard scenario, the CNN architecture took 563 seconds and the MLP took 394 seconds on Machine 2, while on Machine 3, the times were 5400 seconds for CNN and 1200 seconds for standard MLP. With the modified reward system, the CNN took 1080 seconds, and the MLP took only 43 seconds. These results suggest that modifying the rewards and optimizing computation through vectorization not only affected the agents' behaviors but also significantly impacted the computational efficiency of model training, especially for the MLP, which showed a drastic reduction in execution time.

For future work, it is suggested to investigate strategies such as multi-agent learning, where different agents cooperate or compete to improve learning efficiency, as well as the application of recurrent neural networks (RNNs) to capture more complex temporal patterns. Another possibility is to test hybrid architectures that combine DQN with advantage-based methods (A2C, PPO), which could result in more robust and efficient policies. Since Freeway naturally involves competition between two players to score points more efficiently, this approach would be a more natural and robust way to improve the agent's performance.

Finally, this study reinforces the importance of adapting reward systems in reinforcement learning, highlighting both the benefits and challenges of this approach. The insights gained provide a valuable starting point for future research aimed at enhancing the ability of reinforcement learning agents in games and other complex domains.

6. BIBLIOGRAPHIC REFERENCES

[1] MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; et al. **Playing Atari with Deep Reinforcement Learning.** [s.l.: s.n.], 2013. Disponível em: <https://arxiv.org/pdf/1312.5602>.

[2] Sutton, R. S., & Barto, A. G. (2018). **Reinforcement Learning: An Introduction** (2nd Edition). MIT Press. Capítulo 1: The Reinforcement Learning Problem (pp. 1-20)

[3] MNIH, VOLODYMYR, et al. "**Human-level control through deep reinforcement learning.**" nature
518.7540 (2015): 529-533.

[4] Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations — Stable Baselines3 2.6.0a1 documentation. Readthedocs.io. Disponível em: <https://stable-baselines3.readthedocs.io/en/master/>. Acesso em: 21 fev. 2025.

[5] MOREIRA, S. Rede Neural Perceptron Multicamadas. Disponível em: <https://medium.com/ensina-ai/rede-neural-perceptron-multicamadas-f9de8471f1a9>.

[6] Gymnasium Documentation. Farama.org. Disponível em: <https://gymnasium.farama.org/index.html>. Acesso em: 21 fev. 2025.

[7] ALE Documentation. Farama.org. Disponível em: <https://ale.farama.org/index.html>. Acesso em: 21 fev. 2025.

[8] atari-representation-learning/atariari/benchmark/ram_annotations.py at master · mila-iqia/atari-representation-learning. GitHub. Disponível em: <https://github.com/mila-iqia/atari-representation-learning/blob/master/atariari/benchmark/ram_annotations.py>. Acesso em: 21 fev. 2025.