Barcelona School of Economics

Reinforcement Learning Final Project

Car Racing Agent

Ferran Boada Bergadà | Julian Romero | Simon Vellin

Contents

June 30, 2025

1 Introduction

1.1 Purpose of Study

For our final project, we conducted an empirical evaluation of Reinforcement Learning (RL) methods applied to a nontrivial car racing environment. The goal was to explore the practical challenges and performance of different RL approaches in visually complex tasks.

To this end, we used the Gymnasium Python library, specifically the CarRacing-v3 environment. CarRacing-v3 is a visually rich, continuous control task in which an agent must learn to drive a car efficiently along procedurally generated tracks.

Essentially, we compare the empirical performance of two RL approaches:

- A classic tabular Q-Learning agent with a discretized state space, based on downsampling and binning pixel information.
- A Deep Q-Learning (DQN) agent utilizing a Convolutional Neural Networks(CNN) based architecture, trained end-to-end on raw visual inputs.

For the classic tabular implementation, we relied primarily on the Gymnasium environment for simulation and applied different custom discretization strategies.

For the DQN implementation, we relied on Stable-Baselines3 (SB3), a widely-used library offering reliable and well-tested RL algorithm implementations in PyTorch. SB3 provides convenient tools for training and evaluating agents with deep neural networks, including CNNs for processing visual observations.

Throughout these experiments, we aim to analyze the strengths and limitations of each method, and reflect on the practical application of concepts seen in the course.

1.2 Recap of Relevant RL Concepts

We focus on RL methods covered during the course, which form the foundation for our project. These include:

- **Value Function**: Estimating expected returns from states or state-action pairs, guiding the agent to maximize long-term rewards.

$$V(s) = \mathbb{E}[G_t | S_t = s] \tag{1}$$

  where $V(s)$ is the value of state s, and G_t is the expected return from time t.

- **Q-Learning**: A classic tabular method that uses discretized state spaces to learn action-value estimates, updated iteratively using the Bellman equation.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2}$$

  where $Q(s,a)$ is the action-value, α is the learning rate, r is the reward, γ is the discount factor, and s' is the next state.

- **Deep Q-Learning (DQN)**: Extends Q-Learning by employing CNNs to approximate Q-values directly from raw pixel inputs.

$$Q(s, a; \theta) \approx r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{3}$$

where $Q(s, a; \theta)$ is the Q-value approximated by a neural network with parameters θ, and θ^- are the target network parameters.

2 Environment and Action State Space

As mentioned previously, we use the `CarRacing-v3` environment from the Gymnasium library. This environment is a visually rich, top-down car racing task with procedurally generated tracks each episode, as displaeyd in 1. It provides high-dimensional RGB pixel observations (96x96x3) and supports both continuous and discrete action spaces.

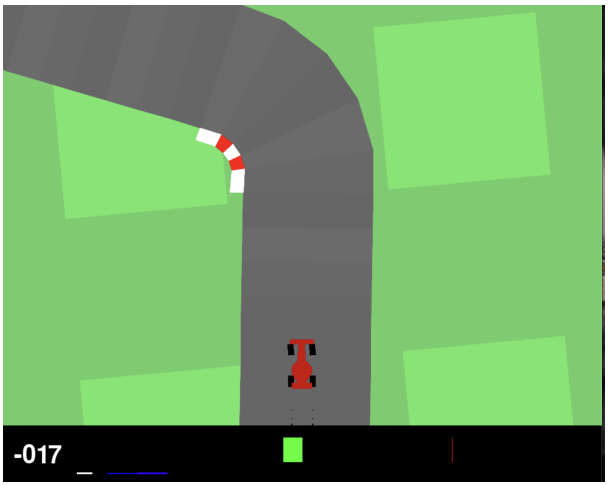


Figure 1: Environment

- **Observation space**: A 96x96x3 RGB image representing the top-down view of the car on the track.
- **Action space**:
  - *Continuous:* 3 actions — steering ([-1, 1]), gas, and brake.
  - *Discrete:* 5 actions — do nothing, steer right, steer left, accelerate, brake.
- **Rewards**: Negative reward per frame (-0.1), plus positive reward for visiting track tiles (+1000/N).
- **Episode ends** when the car goes off the playfield.

For our experiments, we consistently use the discrete action space consisting of five actions, *0: do nothing, 1: steer right, 2: steer left, 3: accelerate, 4: brake.*

We choose the discrete action space primarily to simplify the action selection process, enabling a more direct and fair comparison between classic tabular Q-Learning and Deep

Q-Learning methods. Both these algorithms traditionally assume discrete action spaces, so restricting the environment accordingly avoids additional complexities introduced by continuous control, such as the need for policy gradients or actor-critic methods.

The main challenges in this environment arise from the high-dimensional RGB pixel input, which complicates direct state representation and value estimation. We begin with a simple tabular Q-Learning approach, where the key difficulty is to effectively map these continuous, high-dimensional observations into a manageable discrete state space without losing critical information. Building on this, we then advance to a more sophisticated Deep Q-Learning method, where the focus shifts to automatically learning meaningful feature representations directly from the raw pixels.

2.1 The reward function

The reward function in `CarRacing-v3` is designed to encourage the agent to visit all track tiles efficiently. At each timestep, the agent receives a small negative penalty per frame to promote faster completion, and a positive reward proportional to the number of unique track tiles visited. Formally, the cumulative reward at frame t is:

$$r_t = \frac{1000}{N} \times T_t - 0.1 \times F_t$$

$$R = 1000 - 0.1 \times F$$

where:

- T_t: number of unique track tiles visited up to frame t
- N: total number of track tiles on the track
- F_t: number of frames elapsed up to time t
- R: total reward at the end of the episode (when all tiles are visited)

3 Tabular Q-Learning

We started with a simple and limited approach, using a tabular Q-Learning agent. At its core, Q-Learning maintains a Q-table, where the agent stores its learning by interacting with the environment and observing the resulting rewards.

> For example, when the agent encounters a curve and tries various actions (keep straight, brake, turn right or left), after many interactions it learns that higher rewards come from turning into the curve's direction.

This Q-table serves as a lookup table encoding the expected future rewards (Q-values) for every possible state-action pair. After training, the agent reduces exploration and starts exploiting by selecting the actions that maximize its long-term reward.The main

limitation of this approach is that, while effective in environments with small, discrete state spaces, its practicality diminishes rapidly when faced with high-dimensional, continuous observations like in our case.

Here, the environment provides raw pixel observations as RGB images with resolution $96 \times 96 \times 3$. Each pixel has 256 possible intensity values per channel, resulting in a theoretical state space size of $256^{27,648}$. Such an astronomical number of states makes explicit representation and storage of the Q-table infeasible with current computational resources. The main challenge, then, is to reduce this complexity to a manageable form useful for the agent to learn from.

Moreover, the action space itself can complicate learning. In its continuous form, actions are three real-valued components: steering (from -1 full left to +1 full right), gas, and braking. While this allows fine-grained control, it exponentially increases the action space complexity, making naive Q-Learning intractable. This is why we keep the action space discrete.

3.1 State Space Simplification

To address this, we coded two discretization functions for the observation space. The simplest approach downsamples the image to 8×8 resolution using interpolation, converts it to grayscale by averaging the three color channels, and then discretizes the pixel values by dividing them into bins. This is illustrated in Figures 2 and 3.

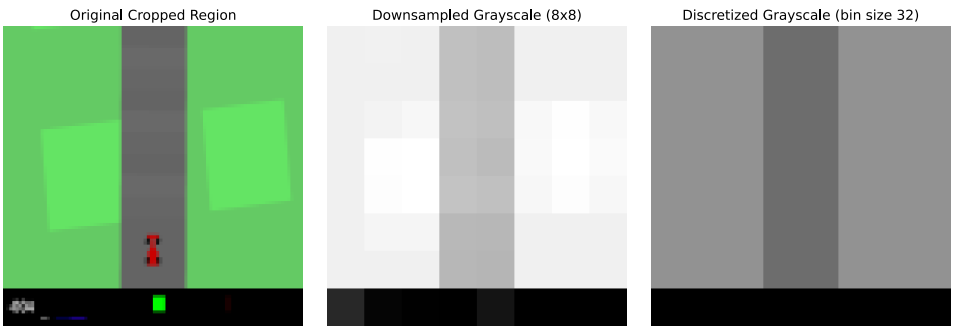


Figure 2: Grayscale discretization with bin size 32.

This sacrifices fine details for computational tractability, and we analyze the trade-off between this simplification and the agent's performance. This reduces the original problem from $96 \times 96 \times 3 \times 5$ actions to $8 \times 8 \times 1 \times 5$ actions in the most reduced case.

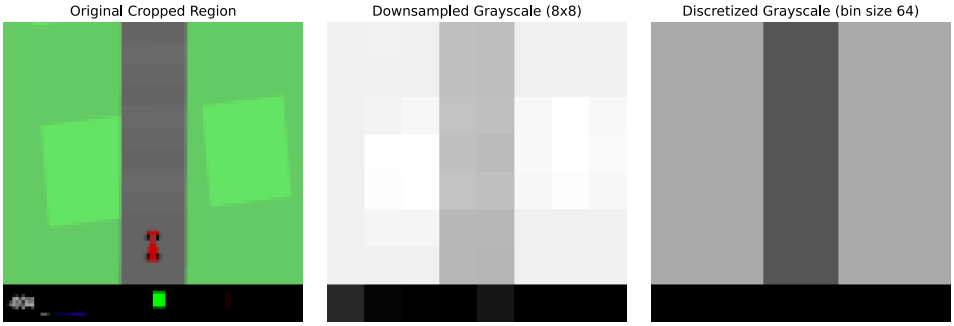


Figure 3: Grayscale discretization with bin size 64.

The second approach preserves the color (RGB channels) by first downsampling to 8×8 and then discretizing each channel separately, flattening the values into a one-dimensional tuple representing the state. The illustrations for this are represented in Figures 4 and 5.

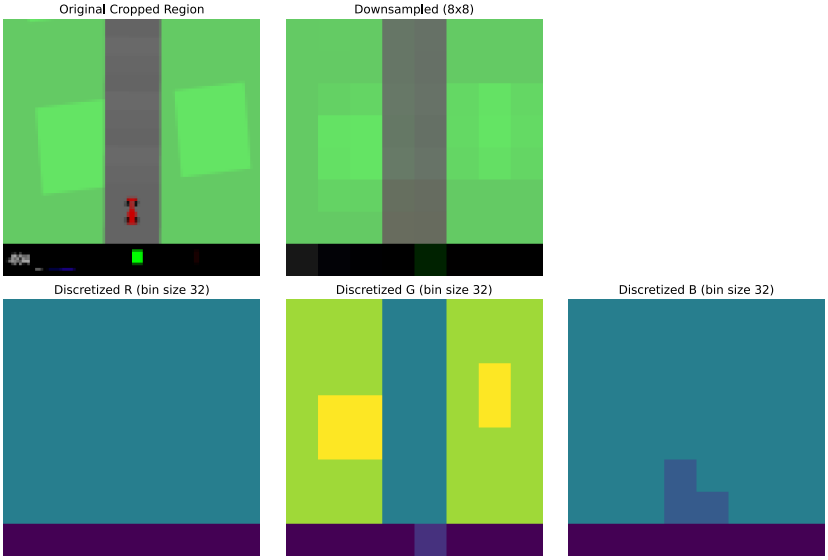


Figure 4: RGB discretization with bin size 32.

Alternatively, a grayscale approximation by averaging the color channels after downsampling and prior to discretization further reduces dimensionality. This compact discrete representation enables use of a finite Q-table, avoiding the need to explicitly handle every raw image.

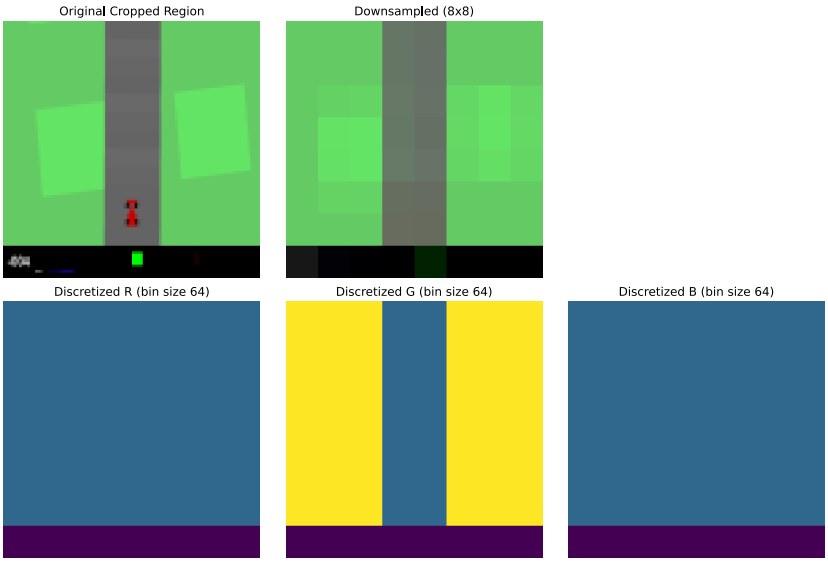


Figure 5: RGB discretization with bin size 64.

While these discretization strategies make tabular Q-Learning technically feasible for image-based environments, they introduce inherent limitations. The loss of visual detail may obscure subtle but important cues, constraining the agent's performance ceiling. Additionally, though reduced, the state space remains large, and the Q-table grows rapidly with finer discretizations or increased bin counts.

3.2 Exploration-exploitation (Epsilon-Greedy Policy)

To balance exploring new actions and exploiting learned knowledge, we implemented a simple epsilon-greedy policy where the agent selects a random action with probability ϵ, and the best-known action otherwise:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon \end{cases}$$

$$\epsilon_{t+1} = \max\left(\epsilon_{\text{final}},\ \epsilon_t - \frac{2}{n_{\text{episodes}}}\right), \quad \epsilon_0 = 1.0, \quad \epsilon_{\text{final}} = 0.1$$

where:

- a_t is the action chosen at time t,
- s_t is the current state (discretized observation),
- $Q(s_t, a)$ is the estimated Q-value for taking action a in state s_t,
- ϵ is the exploration rate (decaying over time).

The agent explores by choosing a random action with probability ϵ, and exploits the best-known action otherwise. Epsilon decays linearly from 1.0 to 0.1, shifting from exploration to exploitation as learning progresses, gradually shifting the agent from exploration towards exploitation as it learns more about the environment.

This balance between exploration and exploitation is crucial in reinforcement learning to avoid getting stuck in suboptimal policies and to ensure the agent learns an effective strategy.

3.3 Results

Figure 6: Performance of various Q-learning agents under different state space discretization strategies.

The above figure shows that grayscale discretization with 32 bins (RGB: False, Bins: 32) consistently outperforms other configurations across all key metrics: average reward, median reward, maximum reward, and reward stability. This setup offers the best trade-off between reducing the state space and maintaining strong learning performance.

Grayscale inputs significantly reduce the dimensionality of the observation space, allowing for a much smaller and more manageable Q-table. Our intuition is that this simplification helps the agent generalize more effectively, as reflected in its higher and more stable rewards. While using fewer bins reduces visual detail, it prevents excessive fragmentation of the state space, which is known to hinder learning in tabular settings.

In contrast, retaining RGB channels, especially with 64 bins—greatly expands the state space. This leads to poor generalization, lower rewards, and increased variance. These agents struggle with Q-table sparsity and exhibit unstable learning dynamics.

In summary, the grayscale 32-bin configuration provides a computationally efficient state representation without sacrificing the essential information needed for effective learning. It strikes the optimal balance between Q-table size and input granularity.

Finally, training time grows approximately linearly with the number of episodes. However, none of the agents show clear signs of convergence, suggesting that even the best-performing setup has not fully stabilized (converged) within the tested episode range.

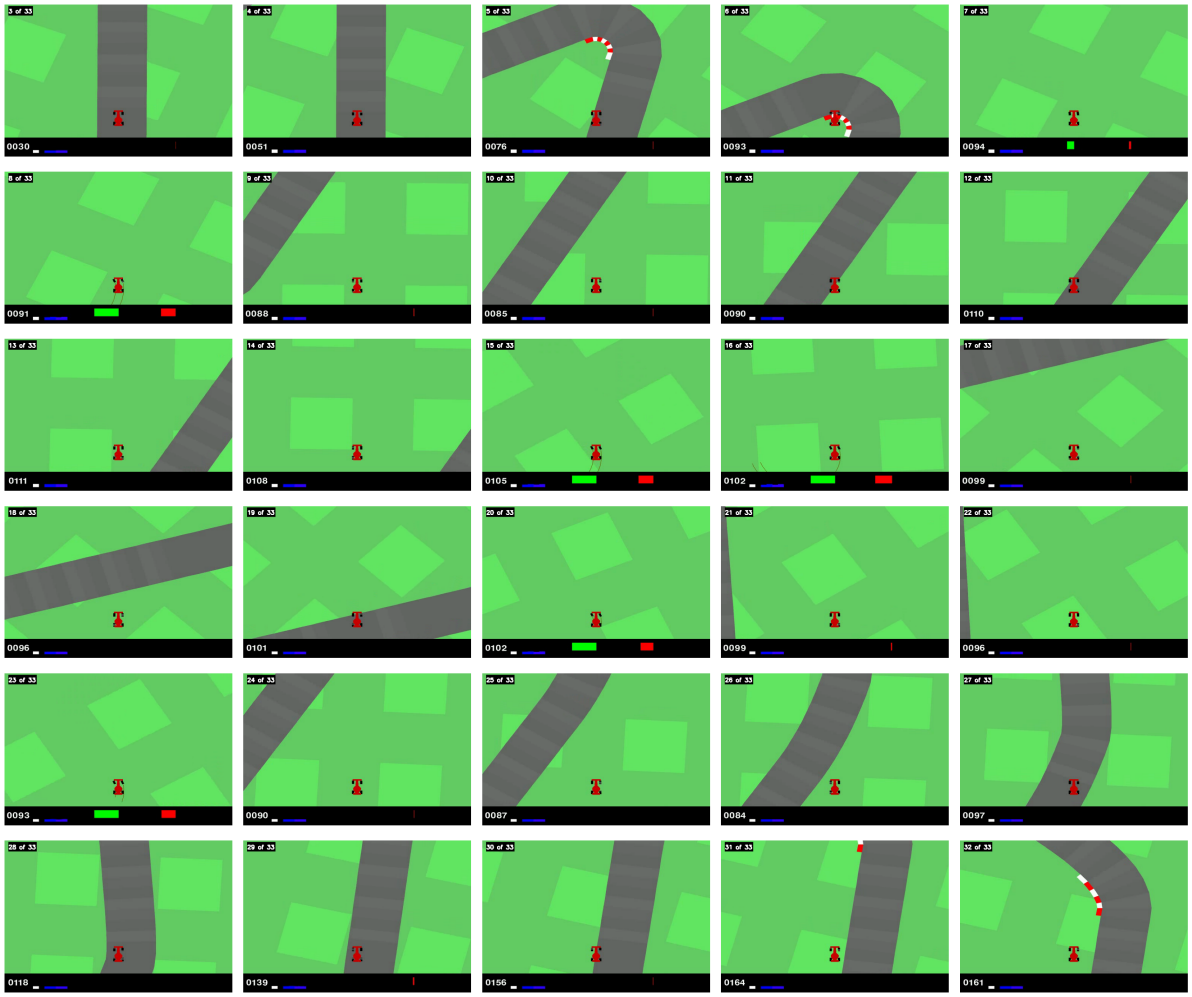


Figure 7: Performance of best Q-learning agent on real environment. Frames of video

Figure 7 shows a sequence of video frames resulting from the training process of the best-performing tabular agent (bin size 32, grayscale, 50K iterations). The frames are presented in chronological order from left to right and top to bottom.

As observed, the car is capable of driving straight and recognizing upcoming curves. In the last frame of the first row, it becomes evident that the agent is not yet able to complete the turn in a single maneuver. However, in the subsequent frames (second row), the car successfully returns to the track, demonstrating a basic level of corrective behavior.

On the one hand, it is encouraging to see that, despite the simplicity of the tabular Q-Learning approach, the agent is able to interact meaningfully with the environment. On the other hand, these results also highlight the limitations of this method, especially in handling complex scenarios such as continuous turns.

4 Deep Q-Learning (DQN)

The limitations of tabular Q-Learning with the large 96 by 96 by 3 RGB pixel observations, even when reduced to an 8 by 8 grid with bin sizes of 32 or 64, as seen earlier, require a better approach.

4.1 Justification

Traditional Q-Learning with tables becomes impractical for environments like CarRacing-v3, and we have learned that modern methods relies on deep neural networks to simplify the environment representation.

Thus, we have used the same environment but Deep Q-Networks (DQN), which estimate values for actions without listing every possible state (i.e. learning directly from raw pixels without manual feature design).

DQN uses CNNs to work with raw pixel data, estimating action values directly, helping the agent learn from large visual inputs without the state space growing too big. CNNs pull out key features like edges and curves from the 96 by 96 by 3 images, avoiding manual setup, which is great for CarRacing-v3's visual nature. However, the choice of this method also comes with its own set of challenges, mostly a consequence of the increased complexity of the neural network architecture and the training process. Unlike tabular methods, DQNs require careful tuning of multiple hyperparameters, and potentially also the customization of the NN itself (which we didn't carry out for this project).

Figure 8: CNN raw input

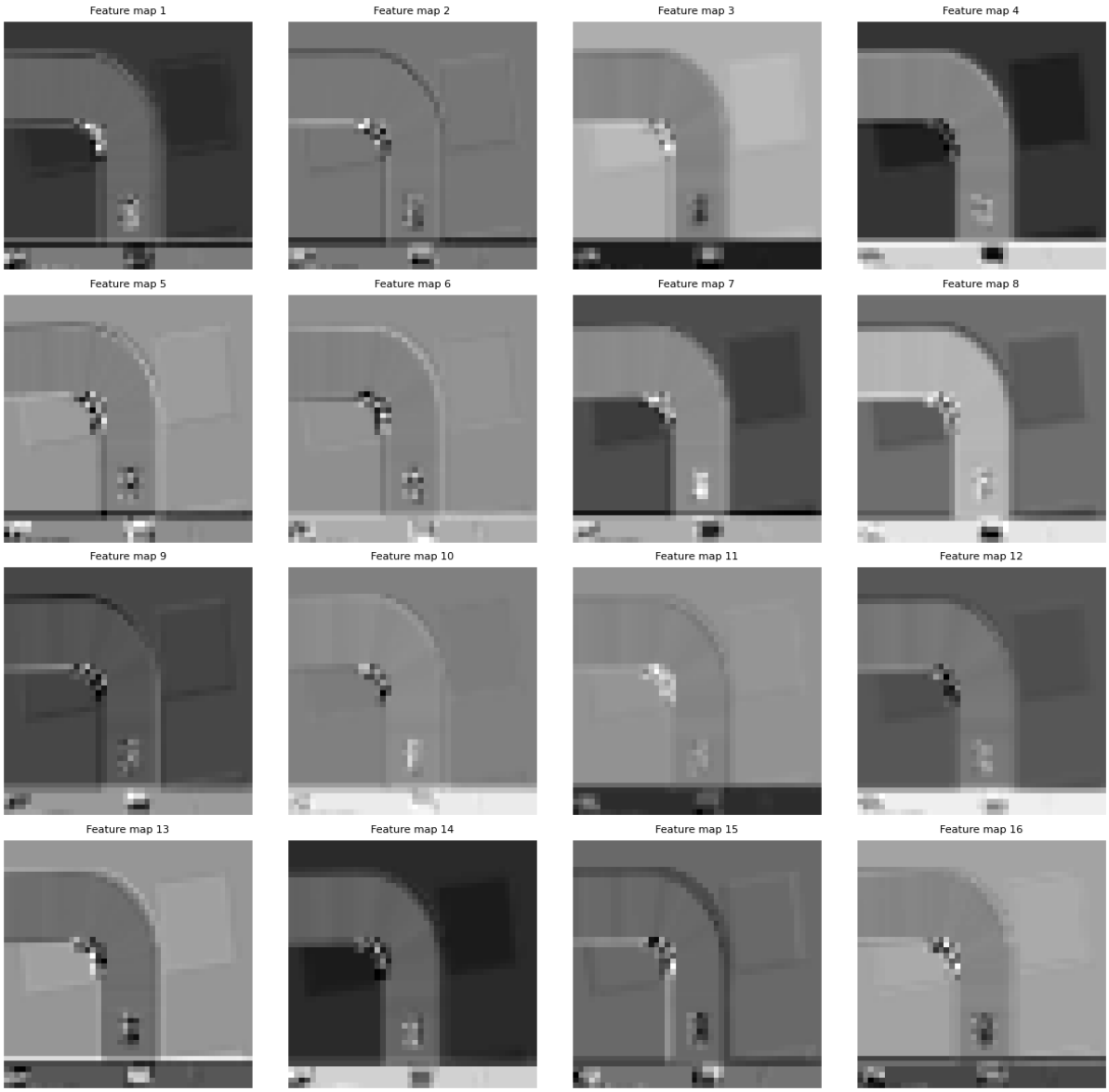


Figure 9: Illustration of CNN 16 neurons input layer abstraction of the raw input from enviroment

Benefits:

- **CNN feature extraction**: CNNs automatically find features like track edges and car position from the 96 by 96 by 3 RGB frames, reducing the need for image discretization and scalable as the input dimensionality increases (unlike the tabular case).
- **Representation power of neural networks**: In contrast to the tabular setting—where the Q-value for every state-action pair must be stored explicitly—neural networks approximate the Q-function continuously over the state space. This allows generalization across similar observations. Compared to linear approximators, neural networks (especially convolutional ones) can represent complex, nonlinear relationships between raw pixel inputs and action values.

- **Frame stacking**: Four frames are stacked using *VecFrameStack* (Stable-Baselines3) to show movement, helping the CNN understand better the dynamics of racing.
- **Replay buffer, target network**: A replay buffer stores past actions for random review (more sample-efficient than the previous method), and the target network updates every 1,000 steps for stability.

4.2 Training details

All models tested have been trained using the DQN algorithm, implemented using the Stable-Baselines3 DQN class with the default 'CnnPolicy', with training runs from 50k timesteps (for faster testing) to 500k timesteps, with the following hyperparameters:

- learning rate: 0.0003
- buffer size: 50,000
- batch size: 128
- gamma: 0.99
- target update interval: 1,000 steps
- train frequency: 4 steps
- learning starts: 5,000 steps
- gradient steps: 1
- Exploration: linear decay from $\epsilon = 1.0$ to $\epsilon = 0.05$ over 10%-20% of training (like in the previous section)

Regarding the *timesteps* parameter mentioned, it corresponds to the value passed to "total timesteps" in the DQN.learn() method. In detail, it controls the total number of environment interactions (i.e., the number of transitions (X_t, A_t, R_t, X_{t+1}) collected). However, the number of Q-network parameter updates (K in Sutton Barto's Algorithm 5.5) is lower, since updates occur every 4 `train_freq` steps in our configuration, and only after a warm-up phase defined by `learning_starts`. Therefore, the number of gradient update steps K is given by:

$$K = \frac{\texttt{timesteps} - \texttt{learning\_starts}}{\texttt{train\_freq}} \times \texttt{gradient\_steps}$$

During training, evaluation is conducted every 5,000 timesteps over 3 episodes in a reset environment, and the best model is saved. We use *EvalCallBack* from Stable-Baselines3. Then we plot the average rewards of each evaluation over training timesteps.

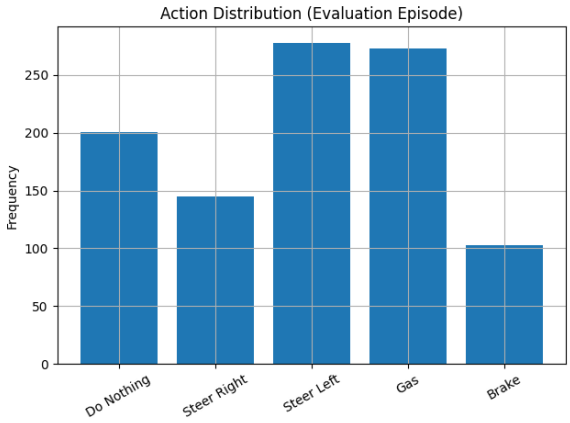
4.3 Reward engineering

After conducting initial training experiments with various parameter combinations (using up to 50,000 timesteps), we observed that the agents were able to improve their behavior compared to a tabular Q-learning baseline. However, their learning process was often unstable and relatively slow. In particular, the agents frequently failed to maintain proper track alignment or orientation, resulting in erratic driving and frequent departures from the track. This raised an important question for us: *could the quality and speed of learning be improved by modifying the default reward function of the environment?*

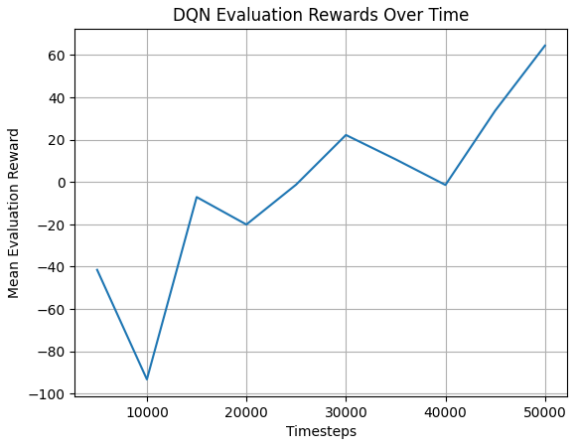
To explore this, we experimented with three additional reward configurations, each corresponding to a different model variant. These were implemented by overriding the environment's `step` method to inject additional reward shaping logic:

- **Model A (Baseline)**: Used the default CarRacing-v2 reward function, which includes penalties for frame duration and partial alignment with track direction, but does not explicitly reward forward velocity.
- **Model B**: Added a clipped velocity bonus to the reward, making high-speed progress more explicitly beneficial ($+0.01 * speed$).
- **Model C**: Built upon Model B by penalizing the `do_nothing` action, which was frequently overused and led to inefficient or stalled behavior. (-0.1)
- **Model D**: Extended Model C with a small penalty for combining steering and high (Box2D) velocity, aiming to discourage sharp turns or "drifting" behaviors that often led the agent off-track. (-0.01)

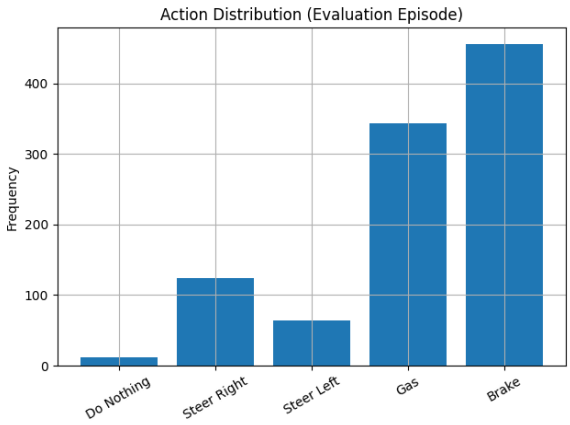
Each model was trained using the same number of timesteps, allowing for a controlled comparison of learning efficiency and behavior. After training, we evaluated the agents on a fixed test track and analyzed the distributions of their actions across an entire episode, inorder to understand how the modified reward structures influenced the learned policies and action preferences.



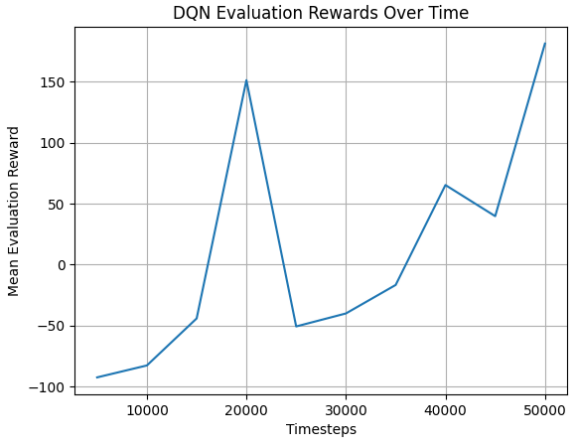
(a) Action distribution for Model B during evaluation.



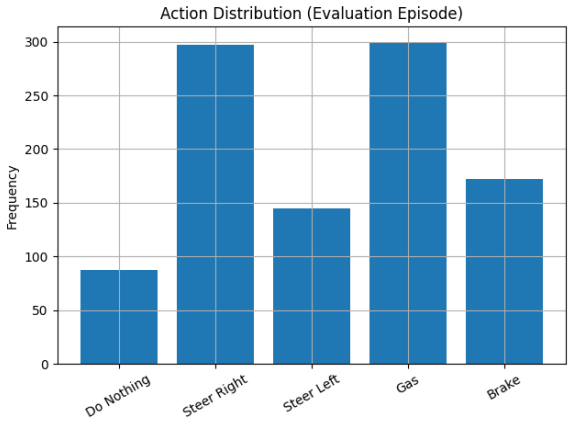
(b) Mean evaluation reward over training timesteps.



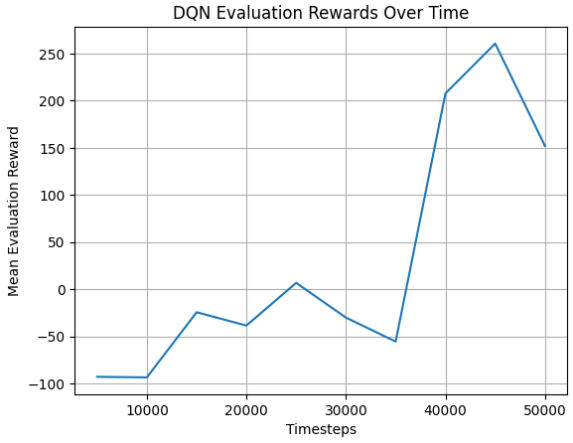
(a) Action distribution for Model C during evaluation.



(b) Mean evaluation reward over training timesteps.



(a) Action distribution for Model D during evaluation.



(b) Mean evaluation reward over training timesteps.

First of all, for this particular setting we observe that the best mean rewards are increasingly better across models; despite changing the default rewards (and therefore the upper bound of 1000 total rewards), since C and D are built upon B, both C and D only include penalties, and D is built upon C in turn, we can assure that the improvement in (relative) rewards is at least the one observed.

However, even though it seems that rewards refinement is positively affecting the learning, after analysing the test videos it was hard to visualize a clear improvement with respect to the baseline model, probably due to the relatively low timesteps used (50k), which don't give the agent enough time to learn.

As for the action distributions, it seems that at least agent C effectively learns an efficient policy (avoiding to choose "do-nothing" actions, and probably replacing them by "brake" sequences), but it's hard to interpret actions in D without visual assessment; according to the video, the agent still struggles in sharp corners.

An interesting direction for future research could be to replicate the comparison of reward

model variants (Models A–D) using a significantly larger number of training timesteps, and additionally, evaluating the models on a carefully designed test track environment.

4.4 Final Results

To conclude the study, we selected the best-performing model architecture based on prior experiments and trained it using the same parameter configuration but with the default CarRacing reward function restored, in order to maintain consistency with the original environment definition. This time, training was extended to 500,000 timesteps. The results show a substantial improvement in both performance and driving behavior. The agent is able to reach mean evaluation rewards exceeding 800 at certain checkpoints, demonstrating not only improved learning but also significantly smoother, more stable, and precise driving along the track.

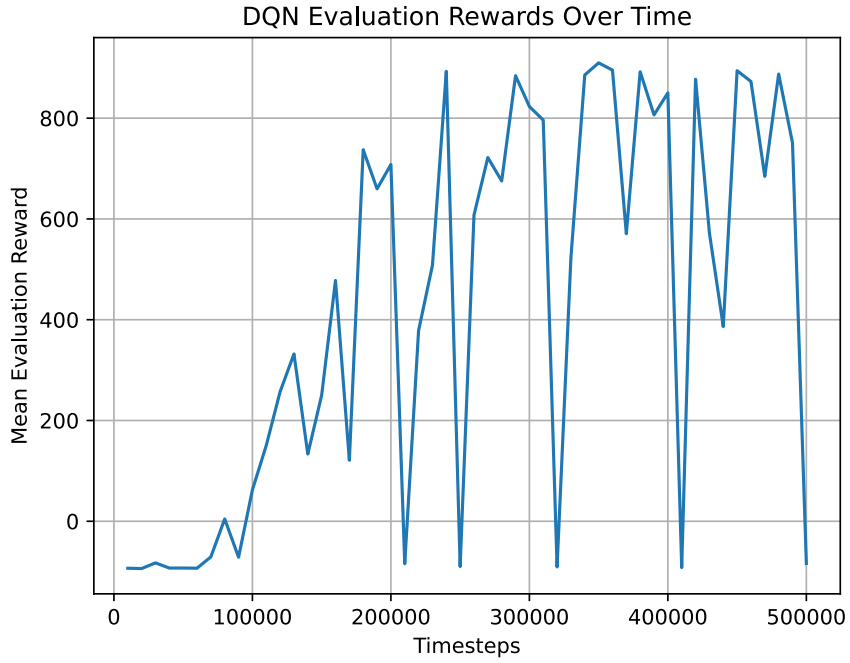


Figure 13: Training curve for DQN-CNN agent

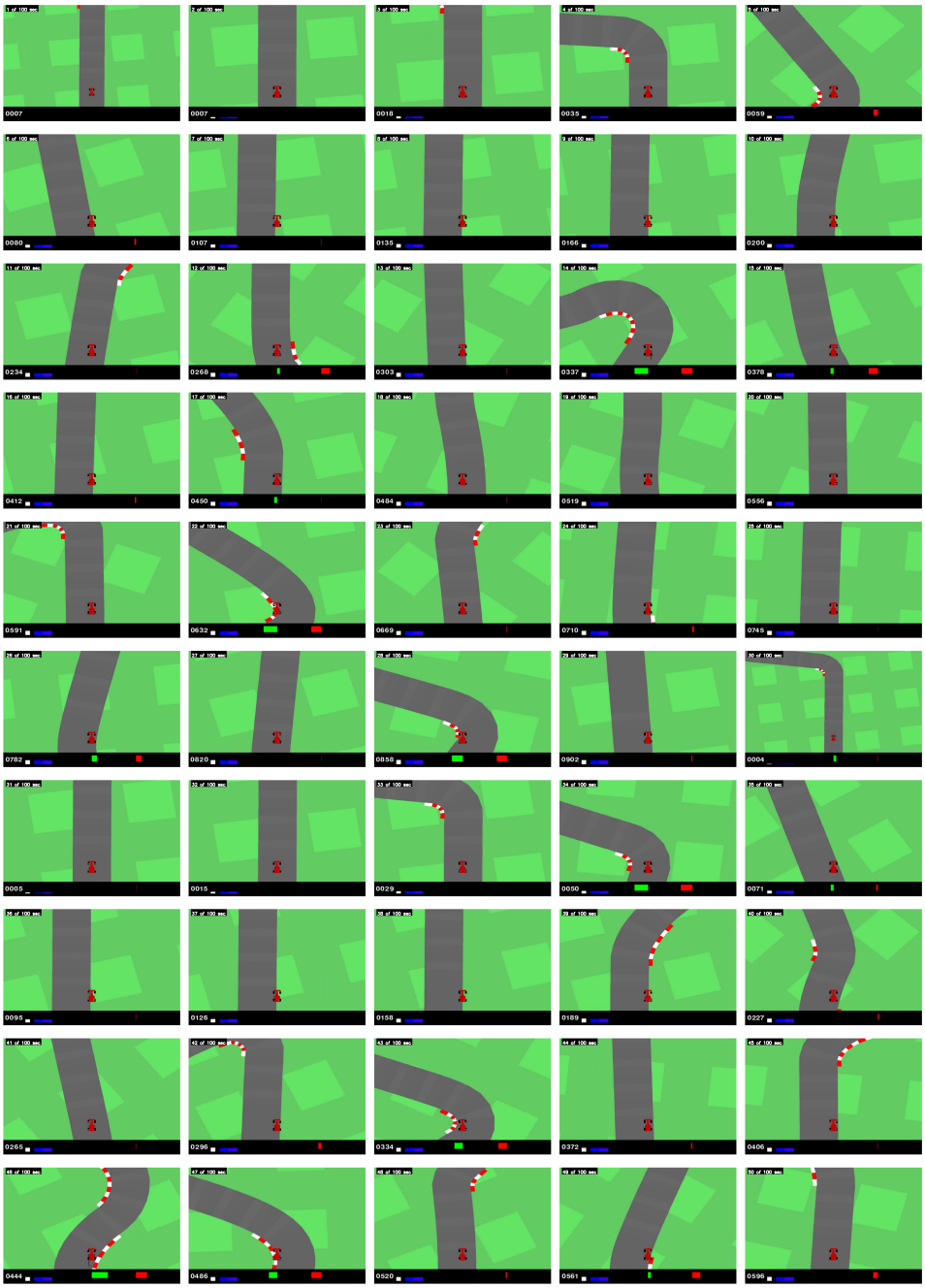


Figure 14: Performance of best DQN-CNN agent on real environment. Frames of video

Figure 14 shows a sequence of frames analogous to the one shown earlier in Figure 7. As we can observe, the car now mostly stays within the track limits, with a fast pace and stability. Unlike in all previous attempts, now the agent also performs optimally in sharp corners (like in image at 3rd row, 4th column), and doesn't go through the grass in chicanes (like in last row, 1st column).

5 Conclusions

This project offered a comprehensive, hands-on exploration of some of the reinforcement learning concepts covered in class in a visually complex (and fun) Gymnasium environment like CarRacing. We began by implementing a classic tabular Q-learning agent and tested various discretization strategies to reduce the dimensionality of pixel-based inputs. While we were able to achieve limited learning and even some corrective behaviors on curves, the approach was fundamentally constrained by the lack of explanatory power of the inputs. These findings highlight the limitations of using traditional tabular methods in high-dimensional state spaces and motivate the need for more scalable solutions in these kind of visual control tasks.

Building on this, we successfully transitioned to a Deep Q-Network (DQN) framework using convolutional neural networks to process the raw image data. We evaluated the effects of different reward engineering strategies to observe if certain shaping components, such as velocity bonuses and action penalties, could influence learning speed and action efficiency. Although short training runs (50k timesteps) showed modest improvements, longer training (500k timesteps) with the default reward yielded the most stable and impressive results. The final model achieved mean evaluation rewards exceeding 800 and exhibited consistently smooth, accurate driving. These results confirm the superiority of deep RL methods in handling complex visual environments.