ML Project Report

Pisano Raffaele
pisano.1959863@studenti.uniroma1.it

1 Introduction

This report presents the development and evaluation of a machine learning agent for solving the Gymnasium 'CarRacing-v2' environment using both Q-learning and Deep Q-Networks (DQN). The project aims to explore state discretization methods, and apply reinforcement learning (RL) techniques to train a agent capable of solving the car racing task.

1.1 Environment

The CarRacing-v2 environment is a reinforcement learning (RL) task designed for agents to learn how to control a car in a top-down racing scenario. The agent must navigate a generated racetrack using visual input (RGB images) and control signals. Each episode begins with a randomly generated track, ensuring variety in learning scenarios across training.

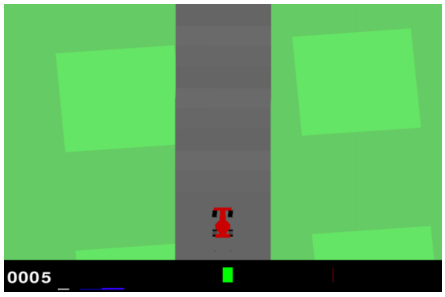

Figure 1: Example of classic scenario

1.2 Observation Space

The agent perceives the environment through a 96x96 pixel RGB image (Figure 2). This top-down view captures the car and surrounding racetrack, with key features such as track boundaries and curves. The observation also includes indicators displayed at the bottom of the screen, providing the following data (Figure 3):

- **True speed:** Displays the car's current speed.
- **Four ABS sensors:** Shows the status of the Anti-lock Braking System.
- **Steering wheel position:** Displays the current orientation of the steering wheel.
- **Gyroscope:** Provides information on the car's angular position and velocity.

These visual information can play an essential role in helping the agent learn how to navigate the track effectively.

Figure 2: Output of the envirorment: 96x96 RGB Image

Figure 3: indicators

1.3 Action Space

The environment offers two types of action spaces: the ***Continuous*** one and the ***Discrete*** one. I decided to use the discrete action space because it simplifies the action and state space, making it more manageable for algorithms like Q-learning. By limiting the number of possible actions, the agent can explore and learn faster, reducing computational complexity. While continuous actions provide finer control, the discrete approach strikes a balance between efficiency and performance, making it easier to implement and optimize in this environment. In the discrete action space the agent has five predefined actions to choose from:

- **Do nothing:** Maintains the current state without steering or accelerating.
- **Steer left:** Moves the steering wheel to the left.
- **Steer right:** Moves the steering wheel to the right.
- **Gas:** Accelerates the car.
- **Brake:** Slows down or stops the car.

1.4 Rewards

The reward function is designed to encourage the agent to cover as much track as possible while penalizing inefficiency. Specifically:

- The agent receives a small *penalty of -0.1* for every frame it remains on the track to encourage faster lap completion.
- For every track tile visited, the agent *earns +1000/N points*, where N is the total number of tiles on the track. For example, if the agent completes the race in 732 frames, it earns a reward of 1000 - (0.1 * 732) = 926.8.
- A severe *penalty of -100* points is given if the car goes far off the track and leaves the playfield, which also terminates the episode. (Figure 4)

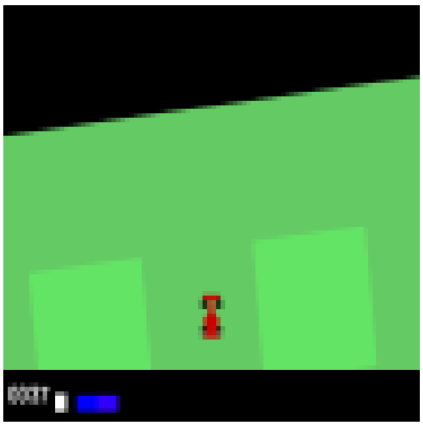

Figure 4: Car going far off the track and leaving the green playfield

1.5 Starting State

At the beginning of each episode, the car starts at rest, centered on the road, ready to navigate the track.

1.6 Episode Termination

The episode ends after 1000 steps or earlier if the car drives far off the track, exiting the playfield.

1.7 Domain Randomization

The environment supports domain randomization for more robust training, each episode begins with a new, randomly generated racetrack, making it difficult for the agent to overfit to specific track layouts.

1.8 Control Dynamics

An essential feature of the environment is the physics of the car.
As it is a powerful rear-wheel drive vehicle, the agent must learn to handle it carefully. For instance, pressing the accelerator while simultaneously turning can lead to loss of control, making the car skid or spin.
Learning the balance between speed and maneuverability is key to achieving high rewards and completing the track efficiently.

2 Q-learning

Q-learning is a reinforcement learning algorithm that aims to learn an optimal policy by

estimating the Q-values for each state-action pair.
The agent updates these Q-values iteratively based on the rewards received after taking actions and transitioning between states.

2.1 State problem

Applying Q-learning to an environment like CarRacing-v2 presents significant challenges, primarily due to the complexity of handling high-dimensional state spaces. In this case, the state is represented by 96x96 RGB images of the track, which makes storing and processing all possible states highly inefficient.

A key issue arises from the fact that even small visual differences in the environment, such as variations in the colors of the terrain or track tiles (light gray vs. dark gray), can result in the algorithm perceiving these as entirely different states, even if the car's speed, steering, and position remain the same.
This results in a massive and redundant state space, making it impossible to efficiently store or update Q-values for every possible state-action pair.
Additionally, the memory and computational costs of dealing with such high-dimensional inputs are prohibitive, slowing down learning considerably.

To address these challenges, I determined it was essential to capture key aspects of the environment rather than using raw image data. Specifically, I represented:

- The ***curvature of the track***, represented with two angles, one directly in front of the car and one slightly ahead of it, providing insights into the track's direction in the short and medium term.
- The ***current speed*** of the car.
- The ***current steering*** angle of the car.
- The ***car's position*** relative to the center of the track.

These values were obtained through a series of image transformations, which are detailed later in the report.

By summarizing the state as a list of numerical values rather than saving a full image, I significantly reduced the state space, leading to increased efficiency in both memory usage and computational processing.

2.1.1 Track Curvature Extraction: Step-by-Step

To accurately capture the curvature of the track in front of the car, I applied several image processing steps aimed at simplifying the image information into a couple of angles representing the track direction.
Here's how I approached this task:

- ***Extracting the Region of Interest:*** The first step is to focus only on the part of the image that is relevant for determining the track curvature. I achieve this by extracting a specific section of the image:

  ```
  zoom = state[20:60, 22:74]
  ```

  This step crops out the bottom section of the image, which contains irrelevant information for obtaining the angeles such as the car itself, current spedd, steering wheel position, ABS indicators, etc. It also slightly trims the sides of the image to focus purely on the track directly ahead of the car, increasing the efficiency of the subsequent image processing steps. Starting point in Figure 5, Zoom in Figure 6.

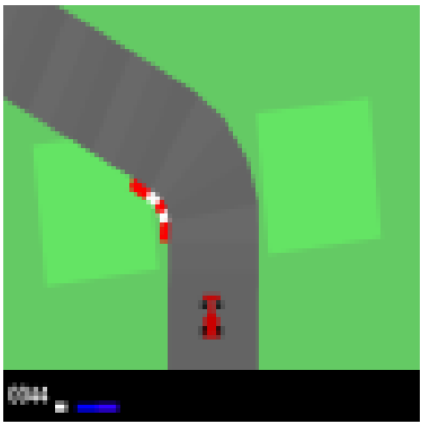

Figure 5: Original Image

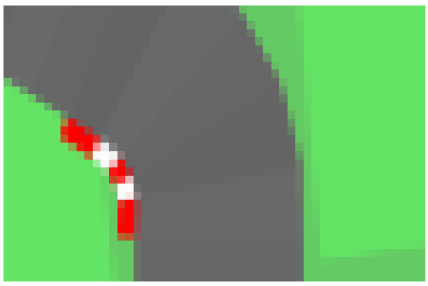

Figure 6: Zoom of interest

- ***Color Transformation:*** To simplify the image further, I applied a transformation to convert the track into a binary image where the road appears black, and everything else appears white. This transformation is crucial for extracting the track's shape and curvature from the state.
  I used the function ***'highlight_track'*** to implement several color masks that help filter out non-track areas based on color properties:
  - *White Pixel Mask:* Removes pixels where all the RGB values are high filtering out white pixels like those on the curbs. (Figure 7)

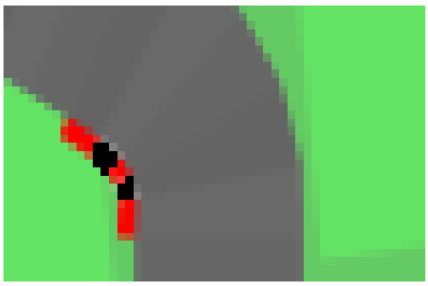

Figure 7: White curbs removed

  - *Color Difference Mask:* Removes pixels where the RGB channels differ significantly, which helps eliminate red curbs and all the green from the surroundings. (Figure 8)

After applying the masks, the remaining pixels represent the track, as its gray color does not trigger any of the filtering masks. All pixels covered by the masks

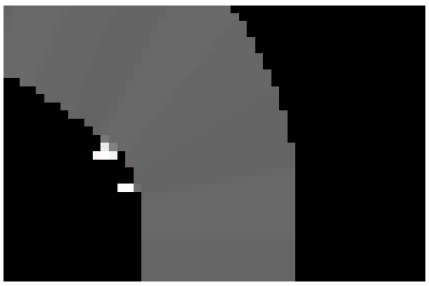

Figure 8: Removed Green and Red

are set to white, and the remaining track pixels are turned black. This conversion results in an image where the track is entirely black, and everything else is white. (Figure 9)

Figure 9: Final Result

- ***Edge Detection:*** After the Color Transformation, I detected the edges of the track, which provided clear boundaries for calculating the curvature. For this task, I used the Canny Edge Detector, a well-known algorithm for edge detection in images. It works by identifying areas of rapid intensity change, which often correspond to edges in an image. It applies two threshold values to identify strong and weak edges, retaining only those that are significant. (Figure 10)
- ***Track Direction Calculation:*** Once the edges of the track were detected, I calculated the track direction using ***two angles***, one for the section of the track immediately in front of the car and another for a section further ahead. These angles help the agent understand the curvature of the track both in the near and mid-

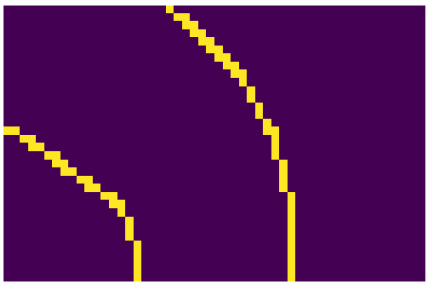

Figure 10: Edges

term. Here's how I approached the calculation:

- ***Identifying Track Edges:*** I focused on two key areas of the image, for the Near-term direction I examined the row immediately in front of the car and the row 14 pixels further up in the image, picking for each of the two rows the left and right track borders positions [l1, r1; l2, r2].
  For the Mid-term direction I examined a row approximately halfway up the image and a row 14 pixels above that midpoint, again picking as before the track borders positions.

- ***Calculating the Track Direction:*** Using the borders position i computed the angle with:

```
angle_left = np.arctan2((l2 - l1), (14))
angle_right = np.arctan2((r2 - r1), (14))
track_direction = (angle_left + angle_right) / 2.0
```

  for both Near-term direction and Mid-term direction, having at the end 2 tracks directions both used as part of the final state saved in the Q-table.
- ***Discretizing the Angle:*** To further simplify the state representation, I discretized the calculated angle by rounding it to the nearest multiple of 0.05. For instance:
  If the calculated angle is 0.02, it becomes 0.05. If the angle is 0.06, it rounds to 0.1.

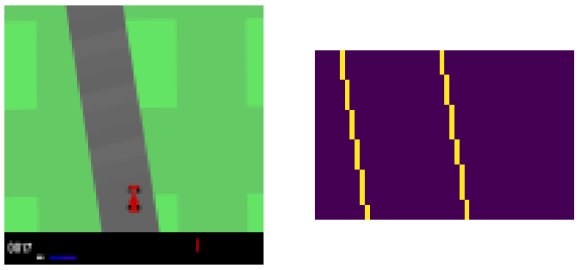

Figure 11: Angles computed: (0.2, 0.15)

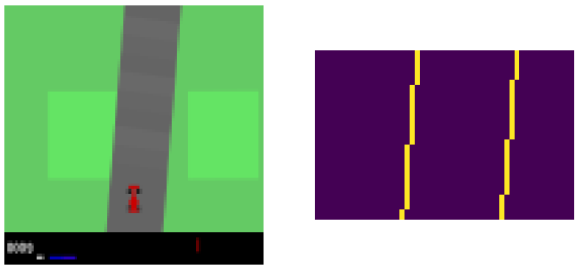

Figure 12: Angles computed: (-0.1, -0.05)

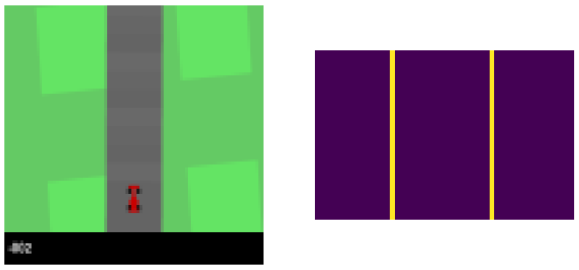

Figure 13: Angles computed: (0.0, 0.0)

- ***Special cases:*** The track direction calculation process assumes that the edges of the track are well-defined. However, due to the curvature of the track, it's not always possible to detect two borders at the same height, especially when approaching sharp turns. Here's how these special cases are managed:

  - ***More than Two Borders Detected:*** When multiple edges are detected due to sharp bends or complex track shapes, the two edges are selected as the rightmost edge closest to the car and the leftmost edge closest to

the car. This approach ensures that the car is guided based on the most relevant edges, keeping the track direction calculation accurate even in complex track segments.

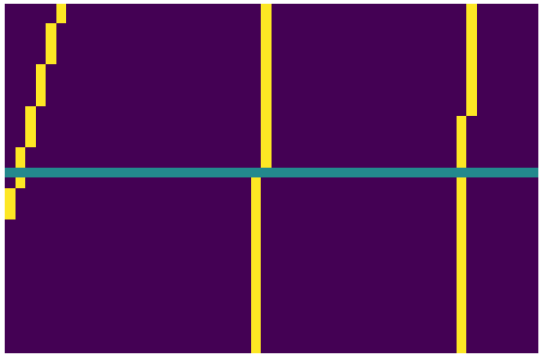

Figure 14: In blue the height considered; Angles= (0.0, 0.0)

- ***Only One Border Detected:*** If only one border is visible, it indicates that the car is approaching or is in a sharp curve. The handling is as follows:
  If there's a border entering the right side of the image, this typically means the car is approaching a sharp right turn. A high negative angle (e.g., -0.8 or -0.9) is set to guide the car right.
  If there's a border entering the left side of the image, this usually signifies a sharp left turn. A high positive angle (e.g., 0.8 or 0.9) is used to steer left.

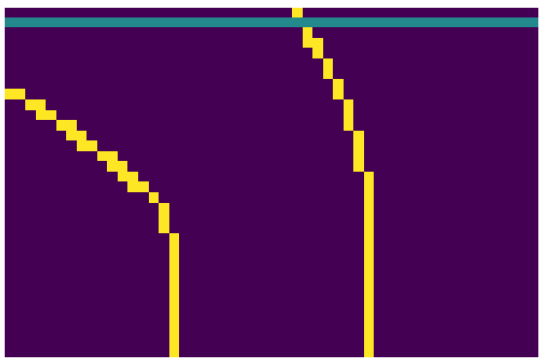

Figure 15: Angles = (0.2, 0.8)

If neither a left nor right border enters a side of the image, then it is checked where's the track with respect to the only one edge detected. if the track is on the left it means that it is curving to the left, right otherwise.

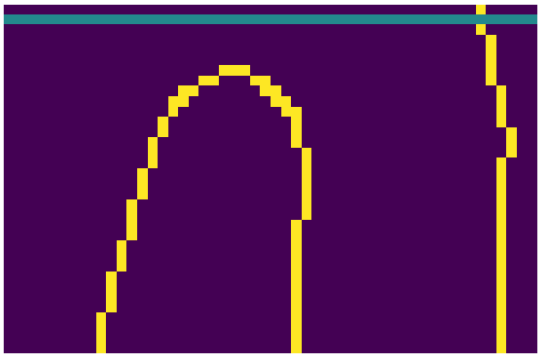

Figure 16: Angles = (-0.15, 0.8)

2.1.2 Speed Extraction

For determining the car's current speed, it wasn't feasible to simply count the number of acceleration or braking actions, as these actions affect the car differently. For instance, a single braking action reduces the speed far more than a single acceleration action increases it. Therefore, I used the speed indicator provided directly in the state image.

The speed is visually represented through a series of small squares at the bottom of the screen, which gradually become brighter as the car's speed increases. To calculate the speed, I zoomed in on the specific area containing these squares and used color thresholds to count how many of them exceeded a certain brightness level. The brighter the square, the faster the car is moving.

I mapped this information to a speed scale ranging ***from 0*** (the car is stationary) ***to a maximum of 20***. This cap of 20 was chosen because at higher speeds, the decision-making process and the car's behavior become relatively similar, so they can be effectively represented by a single state. This approach simplified the problem and avoided excessive complexity without losing critical information about the car's speed.

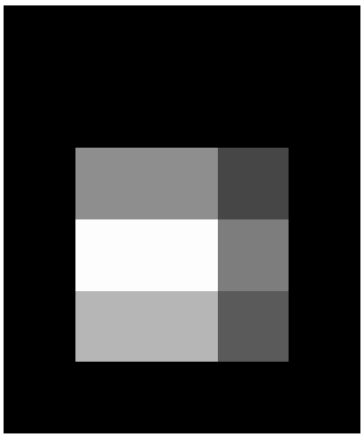

Figure 17: Speed indicator

2.1.3 Steering Angle Calculation

To determine the steering angle of the car, I used an approach similar to the one for speed calculation, focusing on the specific visual indicator present in the state image. The steering angle is represented by green pixels that light up on the dashboard: if the green pixels appear to the right of the center of the steering indicator, it means the car is turning right, and if they appear to the left, the car is turning left.

I zoomed in on the specific area of the image where the steering indicator is located and created a mask to detect green pixels. Based on the number of green pixels present on either side of the image, I inferred the steering angle.

I discretized the steering into 15 possible angles: 7 to the left, 7 to the right, and 1 for driving straight ahead (zero angle).

Figure 18: Steering indicator (max left)

2.1.4 Position Calculation

I developed a method to identify whether the car is ***on the left side*** of the track, ***on the right side***, ***in the center***, or if it has gone ***off the track*** entirely, this allows the agent to receive feedback on car's position without storing complex image data.

Starting from the black track (Image 9) the code check the pixel color at the middle of the image (that's where the car is).
If the pixel is black, the car is still on the track ***(0)***. Then if the car is on the track but closer to the edges (based on a 3-pixel distance), the position is labeled as ***left (-1)*** or ***right (1)***, depending on whether it is closer to the left or right side. If the pixel in the middle of the car is white but it is near an adge (3-pixel tolerance) the car is still inside otherwise it means the car is off the track. ***(-2)***

It also enables quick detection of when the car is off the track, providing a crucial information for the reward structure in Q-learning.

2.1.5 Final State Representation

The final state, used in the Q-learning table, is a ***tuple of five key elements***:

- *First Angle:* Track curvature just ahead (-0.9 to 0.9).
- *Second Angle:* Track curvature further ahead (-0.9 to 0.9).
- *Speed:* Current velocity (0 to 20).
- *Steering Wheel:* Steering direction (-7 to +7).
- *Position:* Car's position on the track (-2: off-track, -1: left, 0: center, 1: right).

2.2 Q-Learning Loop

In the 'Car-Racing-v2' environment, each episode lasts for 1000 steps, or less if the car goes far off the track. The loop uses a classical ***Epsilon-Greedy-Policy*** for choosing next action.

2.2.1 Parameters

I experimented with various configurations adjusting the following parameters:

- ***Alpha*** (Learning Rate)
- ***Gamma*** (Discount Factor)
- ***Epsilon-decay*** (Exploration Probability decay)

I also tested an approach where epsilon increases over time, encouraging more exploration at the end of the episode. Due to the complexity of the 'Car-Racing-v2' environment, each approach required a minimum of episodes from ***3000 to 4000 episodes*** to produce noticeable learning.

2.2.2 Optimal performances issue

While simplifying the state representation to a tuple of five key elements improved efficiency and reduced state space, it also generalized some scenarios that were slightly different. This led to good but not perfect learning, as certain subtle distinctions between states were lost, making it harder for the agent to achieve optimal performance.

2.2.3 Early-stop mechanism

Typically, the environment terminates after 1000 steps or when the car moves too far off the track. However, I modified this by introducing a condition where, if the car goes off the track (detected by my custom function), the reward is set to -100, and the episode ends. This change significantly sped up learning by eliminating unnecessary steps and storage for episodes where the car strayed off the track, forcing the agent to stay on course and maximize rewards more efficiently.

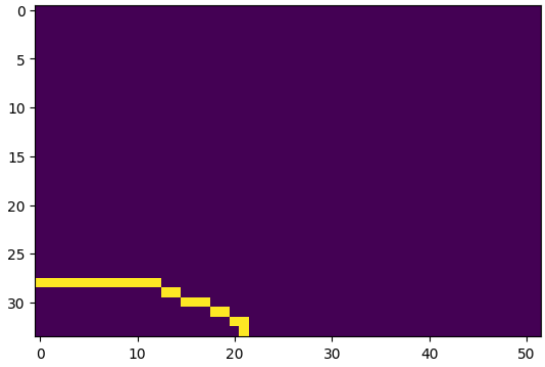

Figure 19: Edges of the track when the code consider the car out

2.2.4 Handling the Initial Zoom Phase

During the first 45 steps of each episode, the environment outputs a zooming image, transitioning from an overhead view to the appropriate scale for gameplay.
Since it was impossible to discretize the state during this zoom phase using my predefined approach, the Q-learning loop was modified. For the first 45 steps, the Q-table is not updated, and action 0 (do nothing) is executed. From step 45 onward, learning and Q-table updates proceed as normal, allowing the agent to interact with a stable, fully zoomed-in environment.

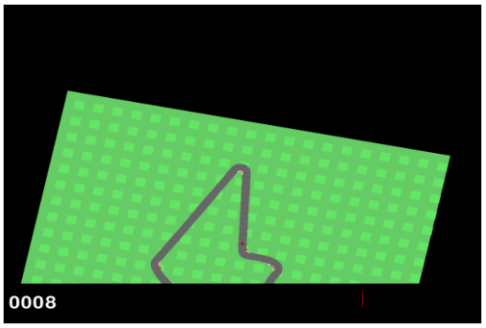

Figure 20: First image returned by the env.

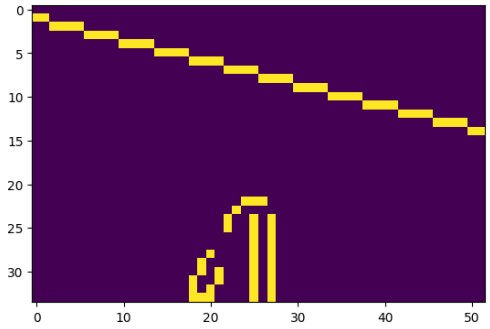

Figure 21: First image's edges

2.3 Results

2.3.1 Evaluation without Early Termination

Notice that in the evaluation phase, the custom penalty of -100 for going off-track is not applied, and the episode does not terminate prematurely if the car leaves the track.
Instead, the default termination and reward mechanism of the original environment is preserved. This decision ensures that the evaluation results remain more aligned with the original benchmark, providing more meaningful and comparable outcomes.

2.3.2 Best configuration

In the following results, the epsilon decreases linearly starting from a value of 1 and at each episode it is multiplied with a ep-decay factor, With a minimum value of 0.15
The reward shown is the average over 15 episodes, executed using the final trained Q-table. A reference reward obtained using random actions across 15 episodes averages around ***-56***.
All the configurations have trained for 3000-4000 episodes.

-	Gamma	Alpha	Ep-dec	Rew.
v1	0.99	0.15	0.9992	15.67
v2	0.97	0.15	0.9992	6.68
v3	0.995	0.15	0.9992	-95.5
v4	0.995	0.15	0.9993	-31.7
v5*	0.99	0.15	0.9993	6.10

Table 1: Results

In the v5* i used a different approximation for the angles: I discretized the calculated angle by rounding it to the nearest multiple of 0.02. and not as the previous cases where the angles is rounded to the nearest multiple of 0.05.

3 DQN

Deep Q-Network (DQN) is an extension of Q-learning that leverages deep neural networks to approximate the Q-values for large or continuous state spaces, where traditional tabular Q-learning becomes impractical, like in the 'Car-Racing' environment. Instead of maintaining a Q-table, DQN uses a neural network to map states to action-value pairs.

In DQN, the network is trained using experience replay and a target network, which stabilize the learning process. Experience replay stores past transitions, randomly sampling them to break the correlation between consecutive states, while the target network helps reduce divergence in the learning process by periodically updating its weights.

In the case of the CarRacing environment, the input state is a visual representation, so a ***Convolutional Neural Network (CNN)*** is used instead of a standard neural network. CNNs are particularly effective at extracting features from images, which is crucial in this scenario for understanding the road layout, speed, and other visual cues.

3.1 State Preparation

In the DQN implementation for CarRacing, the state is represented by the image of the track rather than a simplified tuple of values, as was done in the Q-learning approach. This approach is better suited for DQN and leads to improved results, as the image contains richer and more detailed information about the environment, enabling the model to learn more complex patterns.
However, the image is not used in its raw form but is zoomed into a region of interest to focus on the track immediately ahead of the car, as done previously. Specifically, a section of the image,

```
zoom = state[0:64, 16:80]
```

is extracted, providing a slightly larger view than before.

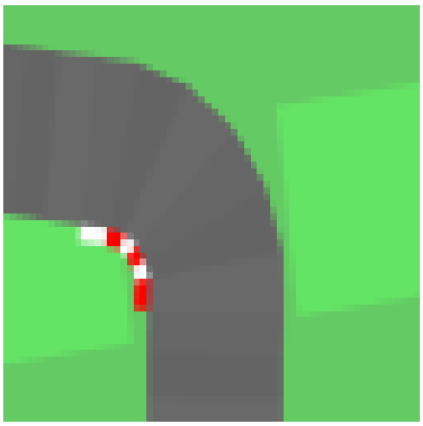

Figure 22: New zoomed track

This zoomed region is then resized using linear interpolation to a resolution of 32x32 pixels, maintaining a wide view of the track while reducing the level of detail. (Figure 23)

Next, the image is processed as before to highlight only the edges of the track while removing all other elements. This is achieved by eliminating all colors except the gray of

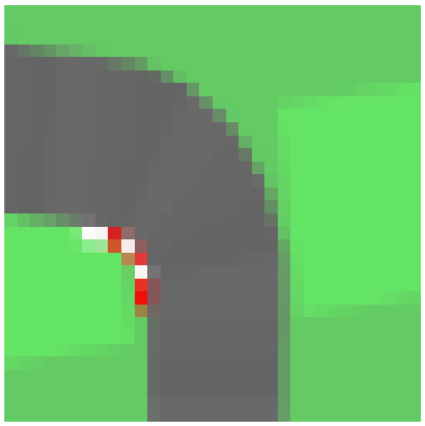

Figure 23: Zoomed after interpolation

the track, followed by the application of the Canny edge detector.

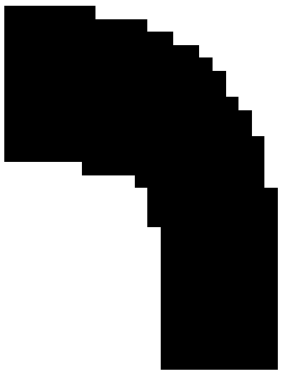

Figure 24: Color removal

The edges, which are initially represented by pixels with a value of 255, are converted to a binary format where edge pixels are set to 1. This results in a final binary image of size 32x32 (a total of 1024 pixels) that captures the essential structure of the track immediately ahead of the car. (Figure 25)

Additionally, as the lower part of the image containing the dashboard values (speed and steering angle) is cropped out, these numerical values are directly fed into the network through a parallel, ***non-CNN branch*** that merges later in the network. The ***speed*** and ***steering*** values are calculated as before, but the car's position relative to the track is no longer included, as the binary image already provides sufficient information about the car's

Figure 25: Final image used as input of CNN

location on the track.

3.2 Neural Network Architecture

The neural network used in the DQN for the CarRacing environment is designed to ***combine two different types of input***: the track image and numerical values representing the car's speed and steering angle. This architecture allows the model to leverage the CNN's ability to interpret visual features while incorporating dynamic information crucial for decision-making.

The ***main path of the network is a CNN***, responsible for processing the track image. This image is passed through three convolutional layers. The first layer applies 16 filters, reducing the image resolution through strided convolutions, thus focusing on the essential features of the track. The subsequent two layers apply additional convolutions with 32 filters, extracting progressively more complex features from the image. At the end of this process, the output is flattened to be concatenated with the numerical data.

The second path handles the numerical values: speed and steering angle. These values are passed through a fully connected layer, converting them into a feature representation.

Finally, the outputs of the two paths are combined via concatenation and passed through another fully connected layer to integrate both visual and numerical information, ultimately producing the action outputs. This combined approach allows the network to uti-

lize both image-based and numerical inputs efficiently to make informed decisions in the driving environment.

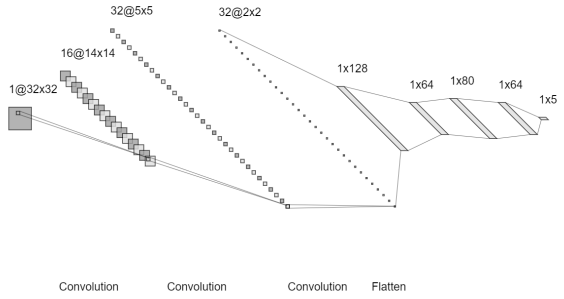

Figure 26: Net

3.3 DQN Training

In the DQN training loop for CarRacing, the agent interacts with the environment by running multiple episodes. At each step of an episode, the agent selects an action using an epsilon-greedy approach:

- with probability ***epsilon***, it explores by choosing a random action
- with probability ***1 - epsilon***, it exploits the current knowledge by selecting the action that maximizes the predicted Q-value from the policy network.

After each action, the environment returns a tuple containing the current state, action taken, reward, next state, and whether the episode was truncated or terminated. This tuple is stored in a replay memory.

At the end of each episode, a ***replay function*** is applied, which randomly samples a batch of stored experiences from memory. These experiences are used to ***update the policy network***. By replaying stored experiences rather than updating the model immediately after each step, the agent can learn more effectively, as the replay helps break the temporal correlation between consecutive actions and stabilizes learning.

Additionally, after every episode, the epsilon value is reduced, making the agent less exploratory as it gains more knowledge of the environment, encouraging it to rely more on the policy learned.

3.3.1 Policy Network vs Target Network

The ***policy network*** is the main model that the agent uses to select actions and optimize during training. However, directly updating this network can lead to instability due to sudden fluctuations in predicted Q-values.

To address this, a ***target network*** is introduced. The target network is a slower-updating copy of the policy network and is used to compute the target Q-values during training.

In my implementation, the target network is updated at the end of each episode by copying the weights from the policy network. Both networks are updated with the same frequency, but they serve different purposes. During the replay function, the policy network is updated based on the samples from the replay buffer.
However, the Q-values used for updating the policy network come from the target network, which remains unchanged throughout the replay function call. The target network is only updated at the end of each episode, ensuring that the target Q-values remain stable during the replay.

This separation between the networks provides more stability during training by smoothing the updates and preventing rapid oscillations in Q-value predictions.

3.3.2 Handling the Initial Zoom Phase

In the DQN implementation, I applied the same initial zoom handling as used in Q-learning for similar reasons.
During the first 45 steps of each episode, the environment provides a zooming image, transitioning from an overhead view to the appropriate scale for gameplay. The images during this zoom phase are significantly different from the typical images the model encounters later on, and might confuse the network by leading to excessive generalization rather than focusing on more relevant track images.
To address this, the Q-learning loop is adapted: for the first 45 steps, the outcomes of steps are not stored in the memory and action 0 (do nothing) is executed. Storing samples

commence from step 45 onward, allowing the agent to interact with a stable, fully zoomed-in environment.

3.3.3 Early-stop mechanism

In the DQN implementation, a similar early-stop mechanism was applied, but with slightly more tolerance regarding the distance from the track.
If the car goes off the track, as detected by my custom function, the reward is set to -100, and the episode ends. This adjustment accelerates learning by avoiding unnecessary steps and storage for episodes where the car moves significantly off course, thereby compelling the agent to remain on the track and maximize rewards more efficiently.

I also experimented with an alternative approach where the episode would end not when the car went off the track but if it was so far from the track that it could no longer see it. This method, however, led to worse results, as detailed in the results section later in the report.

3.3.4 Parameters

I experimented with various configurations adjusting the following parameters:

- ***Batch-Size***
- ***Learning Rate***
- ***Gamma*** (Discount Factor)

All the trials are run using a starting ***Epsilon*** of 1 and at each episode it is reduced by a decay of 0.9992 with a limit min. of 0.15.
Due to the complexity of the 'Car-Racing-v2' environment, each approach required ***3000 episodes*** to produce noticeable learning.

3.3.5 Evaluation without Early Termination

Notice that in the evaluation phase, the custom penalty of -100 for going off-track is not applied, and the episode does not terminate prematurely if the car leaves the track.
Instead, the default termination and reward mechanism of the original environment is preserved. This decision ensures that the evaluation results remain more aligned with the original benchmark, providing more meaningful and comparable outcomes.

3.4 Results

Here are the results from training with various configurations. The best performance was achieved with configuration v1, which resulted in an average reward of ***738.21*** over 15 episodes. Notably, the difference between v1 and v2 is that v2 employed an alternative early-stop mechanism, terminating episodes only when the car moved so far from the track that it was no longer visible. As mentioned, this approach led to worse results. The rewards shown are averages from 15 episodes executed using the trained network with each configuration. For reference, a reward obtained by taking random actions across 15 episodes averages around ***-60***.

-	Batch	lr	Gamma	Reward
v1	***256***	***1e-4***	***0.99***	***738.21***
v2	256	1e-4	0.99	433.39
v3	256	1e-4	0.95	187.23
v4	128	1e-4	0.99	-95.49
v5	512	1e-4	0.99	460.12
v6	512	5e-5	0.99	523.45

Table 2: Results

3.4.1 Best run

The following images illustrate a run achieved with the best-performing neural network. As can be seen, the car has effectively learned to navigate the curves with notable speed, allowing it to earn a high reward. (Figure 27)

4 DQN, Q-learning comparison

Let's now compare the two algorithms and the results:

- The ***Q-Learning algorithm*** is more simple and it's core concept is easy to understand, focusing on learning the value of actions directly through a Q-table, but it struggles with high-dimensional state

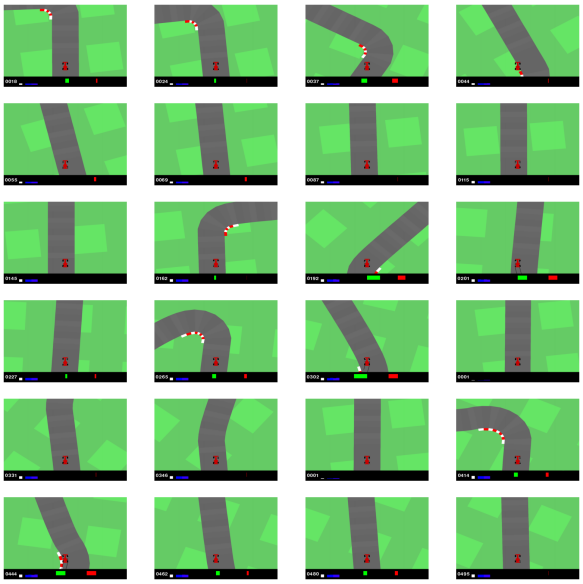

Figure 27: Images from a run

spaces.
In environments like CarRacing, where the state space is complex and involves visual inputs, maintaining a Q-table becomes infeasible.
To make Q-Learning work with complex environments, state spaces are often simplified. For CarRacing, this involved reducing the state to a tuple of simplified features. While this approach made learning feasible, it led to significant generalization. The agent learned to perform well under the simplified conditions but struggled with finer details, resulting in less optimal performance compared to DQN.
In conclusion, due to the simplification and the larger number of episodes required to achieve satisfactory performance, Q-Learning took longer to train effectively.

- The ***DQN algorithm*** isntead, utilizes Neural Networks (and in particular in this case Convolutional Neural Networks: CNNs) to process high-dimensional input states, such as images. This allows the algorithm to handle complex state spaces like those in CarRacing more effectively.
  The CNN in DQN automatically extracts relevant features from raw pixel data, which helps in capturing more nuanced aspects of the environment compared to a simplified state representation.
  DQN proved to be significantly more effective. The network's ability to learn from detailed visual inputs and make more informed decisions led to superior performance and higher rewards.
  Of course, implementing and tuning DQN involves more complexity compared to Q-Learning. It requires careful design of the network architecture, tuning of hyperparameters, and managing the replay buffer and it requires more computational resources due to the use of deep neural networks and the need for extensive experience replay.

In summary, DQN emerged as the more suitable algorithm for the CarRacing environment due to its ability to handle complex state representations through CNNs. Although Q-Learning provided a feasible solution with considerable effort, the results were not as robust, emphasizing the benefits of deep reinforcement learning approaches in high-dimensional and intricate environments.