Advanced Machine Learning Project Report

Kyra Lee

October 2025

Task 1: Train a CNN to predict a clear road ahead

Design Choices

For the Level-2 Minirace CNN, the architecture and hyperparameters were selected through a combination of research, prior experience, and empirical experimentation. The input resolution is small (16×16), so overly deep or complex networks were unnecessary and risked overfitting to the limited visual information. Initial attempts with three convolutional layers or larger feature maps often resulted in slower training and no significant improvement in accuracy, while smaller networks with fewer filters struggled to capture the curvature of the track and local obstacles. The final two-layer convolutional design strikes a balance between extracting sufficient spatial features and maintaining computational efficiency, allowing for real-time evaluation during training and testing. Max pooling and dropout layers were included to reduce overfitting and improve generalization, while ReLU activations were chosen to introduce non-linearities that enable the network to model complex relationships between visual cues and safe driving decisions.

Experimentation with different kernel sizes and pooling strategies informed the final choices: 3×3 kernels provided enough local context without excessive parameter growth, and 2×2 max pooling preserved critical spatial information while reducing dimensionality. Similarly, dropout rates were tuned to prevent memorization of specific track features without hindering convergence. Overall, the design reflects a series of trade-offs aimed at maximizing generalization, training stability, and responsiveness to the environment.

CNN Architecture

The convolutional neural network consists of two convolutional blocks, each followed by a ReLU activation and max pooling operation. The first convolutional layer extracts low-level visual features from the 16×16 grayscale input, such as track edges, lane markings, and the car's position. After max pooling, the spatial resolution is reduced to 8×8, which allows the network to focus on salient patterns while reducing sensitivity to small shifts in the car's position. The second convolutional layer expands the feature space to 32 channels, combining local features into higher-level representations such as road curvature or obstacles. A second max pooling layer further reduces the spatial dimensions to 4×4, which is then flattened into a 512-dimensional vector (see Table 1).

ReLU activations after each convolution introduce non-linearities, enabling the network to learn complex mappings from the input image to the desired output. Max pooling provides translational invariance and reduces the total number of parameters, improving both efficiency and generalization. A dropout layer is applied before the fully connected layers, which randomly deactivates a subset of neurons during training. This encourages the network to learn redundant, robust feature representations and mitigates overfitting, especially given the small input size.

The fully connected layers integrate spatial features into a compact global representation, culminating in a single output logit. This output predicts whether the car can safely continue straight in the next frame ($y = 1$) or not ($y = 0$), framing the problem as a binary classification task. The network is trained using the *BCEWithLogitsLoss* function in PyTorch, which combines a sigmoid activation with binary cross-entropy calculation for numerical stability. Minimizing this loss aligns the predicted probability with the true driving condition, directly supporting the model's objective of safe navigation.

In practice, this architecture provides a balance between representational power and computational simplicity. It efficiently captures both fine-grained and global visual cues necessary for real-time decision-making in the racing environment, while remaining small enough to allow rapid training and inference. This design was informed by iterative experimentation, where overly large or deep networks did not provide meaningful gains, and smaller architectures failed to generalize across varying track segments.

Training and Validation

The CNN was trained using the Adam optimizer with a learning rate of $1x10^{-3}$ and a batch size of 32. Training was conducted for 50 epochs, which provided a good balance between convergence and overfitting. Each epoch took approximately 0.2-0.3 seconds resulting in a total training time of about 10-15 seconds.

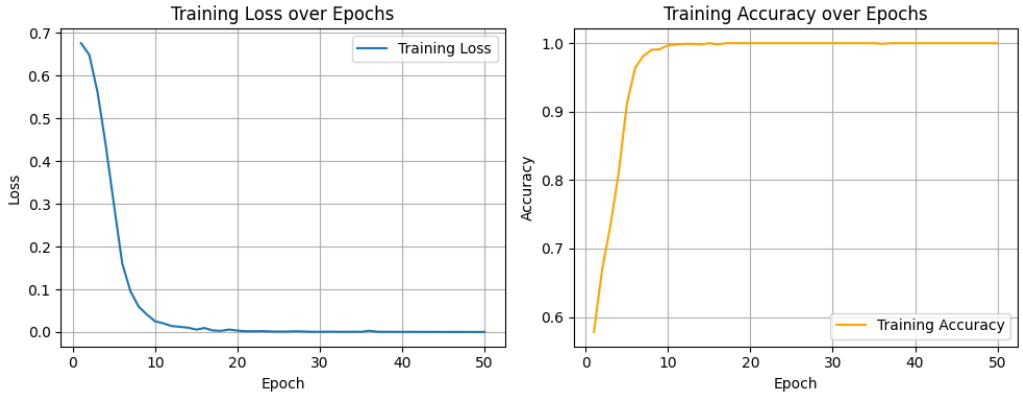


Figure 1: *Loss and Accuracy Across Training*

Both training and test accuracy reached 100% after 50 epochs, indicating that the CNN successfully learned to distinguish between safe and unsafe trajectories in this dataset. The high performance demonstrates that the network's architecture is sufficient for the task, and that the visual features required to make the binary decision are straightforward for the model to capture. While these results confirm excellent generalization on the current test set, caution is warranted for larger or more complex datasets, where overfitting might become a concern.

Parameter / Component	**Value / Description**
Input size	16×16 grayscale image
Number of convolutional layers	2
Conv1 filters	16, kernel size 3×3, padding 1
Conv2 filters	32, kernel size 3×3, padding 1
Activation function	ReLU after each convolution and first FC layer
Pooling layers	MaxPool 2×2 after each convolutional layer
Dropout	$p = 0.2$ before fully connected layers
Fully connected layers	FC1: $512 \to 64$, FC2: $64 \to 1$ (logit)
Loss function	Binary Cross-Entropy with Logits (BCEWithLogitsLoss)
Optimizer	Adam, learning rate 1×10^{-3}
Batch size	32
Number of epochs	50
Training time	$\sim$10–15 seconds total (0.2–0.3 s/epoch)

Table 1: Hyperparameters and Model Architecture of the MiniRaceCNN

Task 2: Train a convolutional autoencoder

The goal of this task is to design a convolutional autoencoder (CAE) that compresses 16×16 grayscale racing game frames into a compact latent representation and then reconstructs them with minimal loss of information. Below, the steps, design decisions, and results are described, including the size of the hidden representation, model architecture, training process, and evaluation of prediction quality.

Undercomplete vs Denoising Autoencoder

Autoencoders are unsupervised neural networks that learn compressed representations of their input data. Their goal is to reduce the dimensionality of the input while retaining enough information to accurately reconstruct it. By exploiting existing patterns in the data, autoencoders learn meaningful features without supervision.

For example, an undercomplete autoencoder is specifically designed to learn a compressed representation by

constraining the latent dimension (the size of the bottleneck layer) to be smaller than the input dimension. This forces the network to focus on the most salient features needed for reconstruction, effectively performing dimensionality reduction and feature extraction.

In contrast, a denoising autoencoder is trained to reconstruct the original, clean input from a corrupted or noisy version. Unlike an undercomplete autoencoder, the latent dimension in a denoising autoencoder can be smaller, equal to, or larger than the input dimension. Its primary distinction lies in the use of corrupted inputs during training, which encourages the network to learn robust representations that can recover the underlying structure of the data even in the presence of noise.

Overall, both types of autoencoders aim to learn meaningful feature representations, but they achieve this through different constraints and training strategies.

Convolutional, Undercomplete Autoencoder Architecture

Overview

The convolutional autoencoder (CAE) designed for this task is a symmetric neural network architecture composed of two main parts: an encoder and a decoder. Its goal is to learn a compact, low-dimensional representation (or latent code) of the 16×16 grayscale racing game screenshots and then reconstruct the original images from that compressed form. This allows the network to automatically learn meaningful visual features such as the track boundaries and the car's position without supervision.

Model Design Process

To develop the convolutional autoencoder, I started by preprocessing the MiniRace 16×16 frames, normalizing pixel values to keep training stable. Initially, I experimented with a very shallow network with only one convolutional layer in the encoder and decoder. However, the reconstructions were extremely blurry, indicating that the model was not able to capture enough detail from the input frames. To address this, I increased the number of convolutional layers in both the encoder and decoder, which improved the overall structure of the reconstructions but still lacked sharpness in some features.

Next, I tried adjusting the latent space size. A very small latent dimension caused the model to lose important details, while a very large latent space led to over-parameterization without noticeable improvements. I settled on a middle-ground latent dimension that provided a good balance between compression and reconstruction quality. I also experimented with different kernel sizes and numbers of filters; increasing these allowed the model to better capture fine spatial details in the frames.

For training, I used the mean squared error (MSE) loss and the Adam optimizer. Early on, a higher learning rate caused unstable training, with the loss fluctuating dramatically between epochs. Reducing the learning rate and using a batch size of 32 allowed the model to converge smoothly. Over 50 epochs, the loss steadily decreased, indicating that the network was learning to reconstruct the frames more accurately. Below, the final design for the encoder and decoder is described in further detail, and all model design decisions are represented in Table 2.

Encoder

The encoder progressively reduces the spatial dimensions of the input while increasing the number of feature maps, allowing the network to capture increasingly abstract visual patterns.

- **Input:** Each image is a 16×16 grayscale frame, reshaped into a tensor of size (1, 16, 16).
- **First Convolutional Layer:** The first layer uses 16 convolutional filters with a kernel size of 3×3 and padding of 1, preserving the spatial resolution. This layer captures local spatial features such as edges and contours.
- **Max Pooling Layer:** A 2×2 max-pooling operation reduces the feature map size from 16×16 to 8×8, effectively downsampling the image while retaining the most prominent features.
- **Second Convolutional Layer:** A second convolution layer with 32 filters (3×3, padding 1) is applied, further enriching the feature representation while keeping the 8×8 resolution.
- **Second Pooling Layer:** Another 2×2 pooling operation reduces the feature map to 4×4, resulting in a compact 32×4×4 feature tensor.

Parameter / Component	Value / Description
Input size	16×16 grayscale image
Number of convolutional layers	2 (encoder) + 2 (decoder, transposed conv)
Conv1 (encoder)	16 filters, kernel size 3×3, padding 1
Conv2 (encoder)	32 filters, kernel size 3×3, padding 1
Pooling layers	MaxPool 2×2 after each encoder conv layer
Flattening	$32 \times 4 \times 4 \rightarrow 512$-dimensional vector
Latent vector	Fully connected: $512 \rightarrow 64$ dimensions (bottleneck)
FC layer (decoder)	Fully connected: $64 \rightarrow 512$, reshaped to $32 \times 4 \times 4$
Transposed Conv1 (decoder)	Upsample $4 \times 4 \rightarrow 8 \times 8$, $32 \rightarrow 16$ channels
Transposed Conv2 (decoder)	Upsample $8 \times 8 \rightarrow 16 \times 16$, $16 \rightarrow 1$ channel
Activation functions	ReLU after each conv/transposed conv layer; final sigmoid for output
Loss function	Mean Squared Error (MSE)
Optimizer	Adam, learning rate 1×10^{-3}
Batch size	32
Number of epochs	50
Training time	$\sim$15–20 seconds total

Table 2: Hyperparameters and Model Architecture of the MiniRace Convolutional Autoencoder

- **Flattening and Latent Encoding:** The resulting tensor is flattened into a 512-dimensional vector, which is then passed through a fully connected layer to produce a latent vector of smaller dimension (e.g., 32). This latent code serves as a compressed representation of the input image, containing the most essential information needed for reconstruction.

Decoder

The decoder performs the inverse process of the encoder, expanding the latent vector back to the original image dimensions through a series of learned upsampling steps.

- **Fully Connected Layer:** The latent vector is first projected back to a 512-dimensional space and reshaped into a tensor of size (32, 4, 4), effectively restoring the spatial structure of the last encoder feature map.
- **First Transposed Convolution Layer:** This layer upsamples the feature maps from 4×4 to 8×8 while reducing the channel count from 32 to 16.
- **Second Transposed Convolution Layer:** This layer further upsamples the feature maps from 8×8 to 16×16, outputting a single-channel grayscale image.
- **Activations:** ReLU activations are applied after each convolutional and transposed convolutional layer to introduce non-linearity. A final sigmoid activation ensures that the reconstructed pixel intensities are constrained between 0 and 1, matching the normalized input data.

Latent Representation Size

The latent vector is positioned at the bottleneck of the network, between the encoder and decoder, where it captures the most essential features required for reconstruction. Each input frame to the autoencoder consists of 16×16 pixels, which equals 256 values per image. The autoencoder compresses this input into a smaller hidden representation in the latent space, denoted as $h = f(x)$, where f represents the encoder function. In this model, the latent space was designed to have 64 dimensions. This means that each 16×16 input frame, originally represented by 256 pixels, is encoded into a vector of length 64.
The compression ratio can be calculated as:

$$\text{Compression ratio} = \frac{\text{size of input}}{\text{size of latent representation}} = \frac{256}{64} = 4.$$

Thus, the autoencoder reduces the original input to one-quarter of its size in the latent space. Despite this reduction, the latent representation retains the most salient features of the frame. This is reflected in the reconstructions shown in Figure 3: even at early epochs, the model captures the overall structure of the frame, and as training progresses, finer details emerge. The latent space effectively balances compression and information retention, allowing the decoder to reconstruct the input with increasing accuracy over successive epochs.

Training and Results

During training, the encoder compresses each input image into a lower-dimensional latent representation, while the decoder attempts to reconstruct the original image from this compressed form. This iterative process minimizes reconstruction error, encouraging the latent space to encode the most informative and salient features of the environment. Training was performed over 50 epochs with a batch size of 32, and the loss curve indicates steady convergence with diminishing returns as the number of epochs increases (Figure 2).

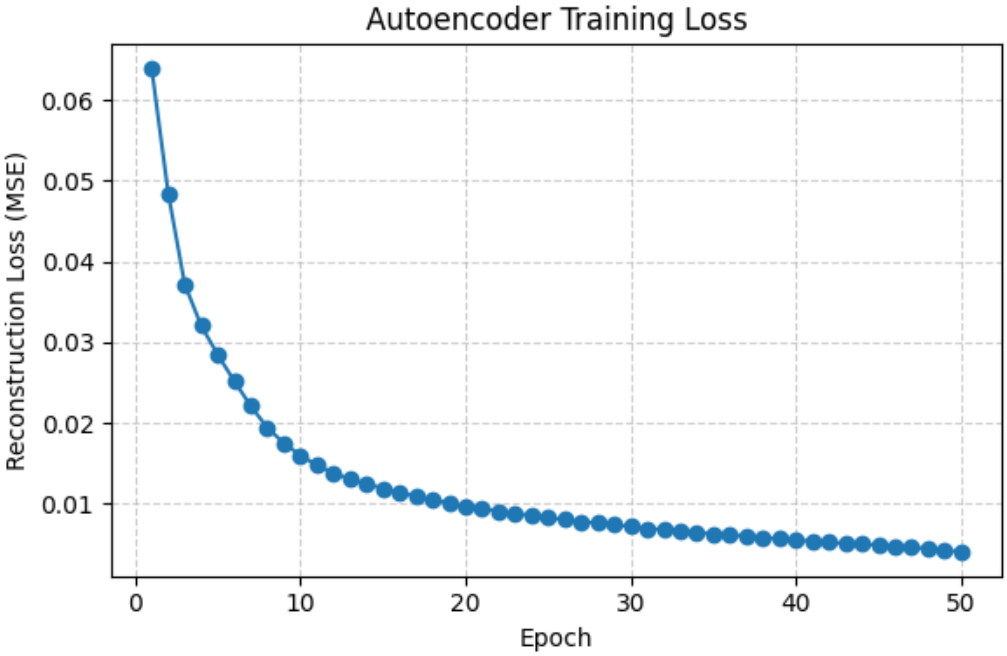


Figure 2: *Autoencoder Loss Over Training. The loss decreases consistently over epochs, demonstrating the model's ability to learn meaningful representations of the input frames.*

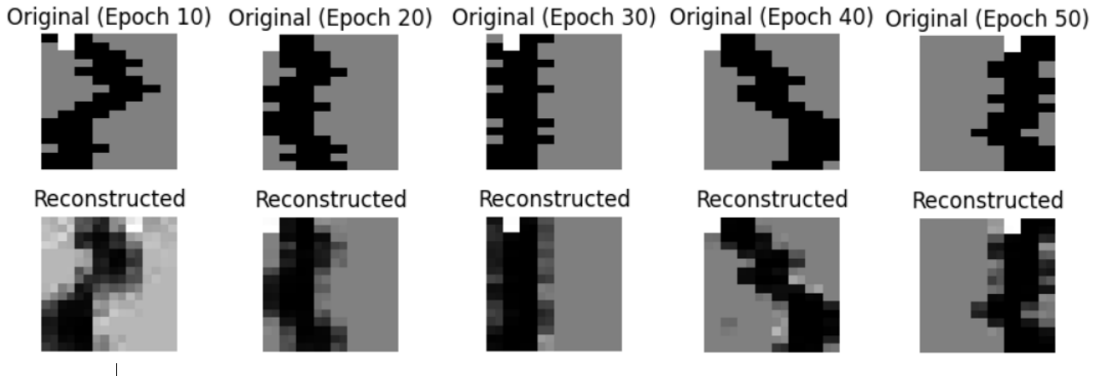


Figure 3: *Reconstruction of a single MiniRace 16×16 frame over training epochs. The top row shows the original input frames, while the bottom row displays the corresponding reconstructions produced by the convolutional autoencoder after 10, 20, 30, 40, and 50 epochs of training.*

After training, the model is able to reconstruct input frames with varying degrees of fidelity depending on the number of epochs. Figure 3 illustrates the reconstruction of a single MiniRace 16×16 frame over 10, 20, 30, 40, and 50 epochs. Initially, reconstructions capture general shapes and positions but lack fine details. As training progresses, the reconstructions improve, showing sharper edges and more accurate feature placement, though subtle details remain imperfect.

Task 3: Create a RL agent for Minirace (level 1)

Design Choices

Q-learning was selected as the control algorithm for the Level 1 MiniRace environment due to its simplicity, interpretability, and proven effectiveness in low-dimensional, fully observable tasks. In this setup, the environment provides a single state variable, dx, representing the car's lateral deviation from the track center. Because the state and action spaces are both discrete and small, a Q-learning approach is well-suited for this problem—allowing the agent to maintain and update a Q-table without requiring function approximation. Moreover, Q-learning's off-policy nature enables the agent to learn an optimal policy through exploration while following an ϵ-greedy behavior policy, balancing exploration of new actions with exploitation of the learned value estimates. The deterministic nature of the Level 1 environment, combined with its limited state space, ensures that the Q-table converges rapidly to near-optimal values, making Q-learning a natural first step.

Reinforcement Learning Architecture

The level-1 driving agent was implemented using a tabular Q-learning approach. In this setup, the environment provides a single observation variable, dx, which represents the car's lateral offset from the track center. Ignoring crash terminations, dx can take on five discrete values, $dx \in \{-2, -1, 0, 1, 2\}$, yielding five possible states in total. At each timestep, the agent selects one of three discrete steering actions, $a \in \{-2, 0, 2\}$, corresponding to steering left, going straight, or steering right, respectively. Combining these state and action spaces results in a Q-table of size

$$\text{states} \times \text{actions} = 5 \times 3,$$

where each entry $Q(s, a)$ represents the estimated cumulative reward for taking action a in state s. This compact formulation allows for efficient computation while still capturing meaningful driving behavior.

To balance the need for exploration and exploitation, the agent employed an ϵ-greedy policy for action selection. With probability ϵ, a random action was selected to encourage exploration, while with probability $1-\epsilon$, the agent exploited its current knowledge by choosing the action with the highest Q-value. This mechanism ensured that the agent could discover effective steering strategies early in training before gradually converging to a more deterministic policy as ϵ decayed.

After each state transition, the Q-table was updated using the temporal-difference (TD) learning rule:

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

where α is the learning rate and γ is the discount factor. This update rule allows the agent to iteratively propagate reward information backward through previously visited states, improving its policy through repeated interaction with the environment.

The reward structure in the `Minirace` environment is intentionally simple, providing a clear signal for survival rather than precision. The immediate reward is defined as:

$$r = \begin{cases} 1.0 & \text{if the car remains on track (non-terminal state)} \\ 0.0 & \text{if the car crashes (terminal state).} \end{cases}$$

As a result, the agent is incentivized to stay on the track for as many time steps as possible. There is no explicit penalty for lateral deviation or corrective steering, so the cumulative reward directly corresponds to the agent's survival time. This sparse reward structure emphasizes long-term stability and robustness, providing a clean testbed for evaluating temporal-difference learning without the confounding effects of shaped feedback.

Overall, this architecture offers a minimal yet effective reinforcement learning setup for autonomous driving in the level-1 `Minirace` environment. Its simplicity enables clear interpretation of learning behavior while highlighting the balance between exploration, exploitation, and survival-based reward optimization.

Hyperparameter Selection and Tuning

The following hyperparameters were chosen after empirical tuning:

Parameter	Value	Description
Learning rate	0.1	Controls the update magnitude for new Q-values
Discount factor	0.99	Encourages long-term reward optimization
Initial epsilon	1.0	Forces full exploration at the start of training
Minimum epsilon	0.01	Ensures small but persistent exploration
Epsilon decay	0.995	Gradually shifts from exploration to exploitation
Episodes	2000	Total number of training iterations

The learning rate $\alpha = 0.1$ provided a good trade-off between stability and adaptation speed. A higher value caused excessive oscillation in Q-values, while a smaller one slowed learning progress. The discount factor $\gamma = 0.995$ encouraged the agent to value long-term rewards, promoting smoother and more stable driving. For exploration, ϵ started at 1.0 and decayed exponentially via $\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \times 0.995)$. This ensured broad exploration in early training and gradual convergence to consistent policies later on. A slower decay (e.g., 0.9995) extended exploration but delayed convergence, while faster decay led to premature suboptimal behavior. Thus, 0.995 was selected as a balanced compromise.

Training Performance and Observations

The Q-learning agent was initially trained for 2000 episodes using the specified hyperparameters, with a runtime averaging between 1–2 seconds. The average of rewards steadily increased from approximately 7 in the early stages of training to above 20 near the end, suggesting that the agent gradually improved its driving behavior. However, when evaluated over 50 test episodes with a fully greedy policy ($\epsilon = 0.0$), the performance metrics were as follows:

$$\text{Test-Average Reward} = 24.16, \quad \text{Test-StandardDeviation} = 20.39.$$

This result indicates that the learned policy was somewhat effective but largely inconsistent across different random tracks. To investigate whether extended training would improve stability, the model was subsequently trained for 5000 episodes (average runtime 5–10 seconds). The evaluation produced:

$$\text{Test-Average} = 21.66, \quad \text{Test-StandardDeviation} = 20.24$$

Contrary to expectations, the mean performance slightly decreased and the variability remained nearly unchanged, implying that additional training did not enhance the agent's generalization or robustness. Thus, an even longer training run of 10,000 episodes (which had an average runtime of 10-15 seconds) produced the following results:

$$\text{Test-Average} = 20.28, \quad \text{Test-StandardDeviation} = 15.02.$$

Although the mean reward decreased further, the reduced standard deviation suggests a modest improvement in stability—potentially reflecting more consistent, but not substantially better, driving behavior. Overall, these findings indicate diminishing returns with increased training duration, as the agent appeared to converge toward a suboptimal yet repeatable policy. Figure 4 illustrates the evolution of training rewards over time, highlighting the early learning phase and subsequent fluctuations after convergence. The relatively high variance in test

Figure 4: *Training Reward Over All Episodes. Early episodes (100–300) have low average rewards (8–9) that increase nonlinearly and then fluctuate after convergence (16-23).*

performance reflects the stochastic nature of the Minirace environment and the coarse discretization of the state

space. Since the tabular method cannot generalize to unseen states, the driving policy remains inconsistent across different random tracks. Nevertheless, these results demonstrate some learning of a basic driving strategy purely through experience, validating the effectiveness of Q-learning in a dynamic and partially unpredictable setting. After analyzing the agent's performance across multiple episodes, including the mean and standard

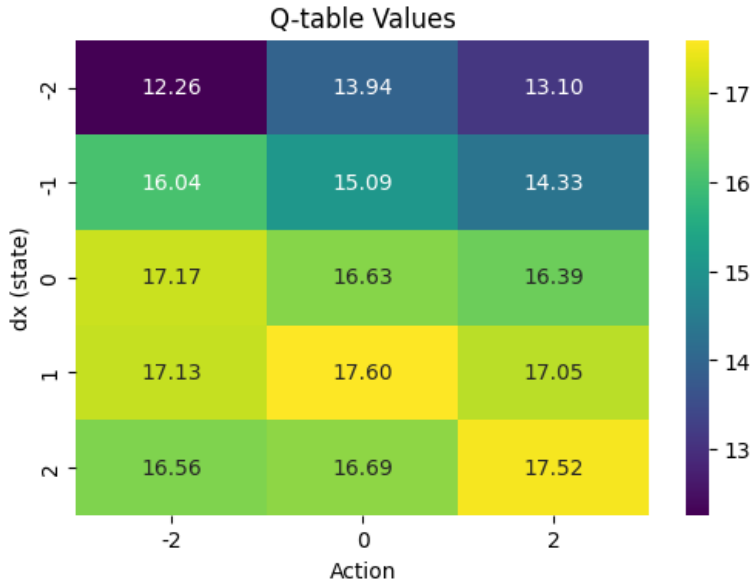


Figure 5: *Heatmap of the learned Q-table values. Rows correspond to dx states, columns correspond to actions.*

deviation of cumulative rewards, it is informative to examine the internal policy encoded in the Q-table. This provides insight into how the agent translates experience into action selection. The Q-table learned by the agent provides a compact representation of the expected cumulative rewards for each state-action pair. Figure 5 shows a heatmap of the Q-values after training. Each row corresponds to a discretized dx state (-2 to 2), and each column corresponds to one of the three steering actions (left, straight, right). Higher Q-values indicate actions that the agent has found to yield higher long-term rewards from that state. The heatmap reveals clear patterns in the agent's preferences: for example, states with negative dx values tend to favor corrective actions to steer right, whereas positive dx states favor steering left. The Q-table thus encodes the learned strategy for keeping the car centered on the track.

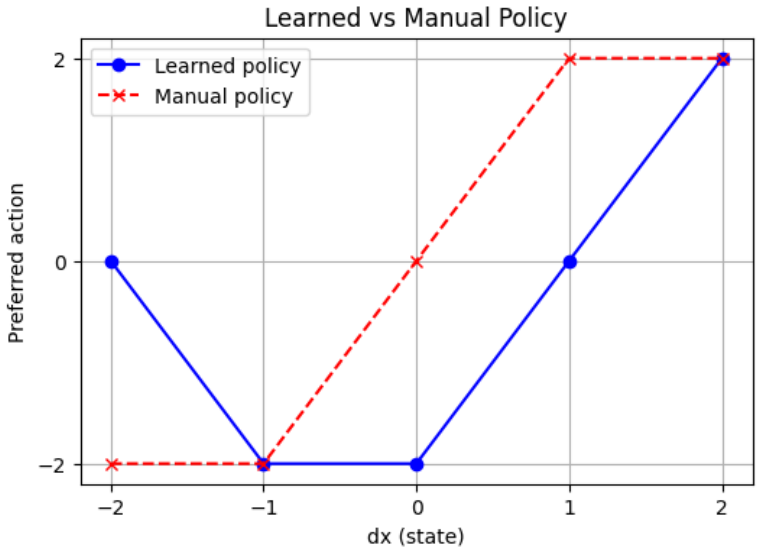


Figure 6: *Comparison of the learned policy (greedy from Q-table) vs. a manually designed heuristic.*

To better understand the practical effect of the learned Q-values, the corresponding greedy policy was extracted by selecting the action with the highest Q-value for each dx state. Figure 6 compares this learned policy to a manually designed heuristic policy. On the horizontal axis, dx indicates the car's lateral offset,

and on the vertical axis, the chosen steering action is shown. The manual policy prescribes more aggressive corrective actions, for example steering left when $dx = -1$, whereas the learned policy is more conservative, often preferring to go straight unless the offset is extreme. This difference arises because the learned policy adapts to the stochasticity of the environment: since track segments are randomly generated each episode, the agent favors stability and long-term survival over aggressive corrections that might occasionally result in crashes.

Both the heatmap and the policy comparison emphasize that even with a small state-action space, the agent can discover nuanced strategies that balance exploration and exploitation over time, reflecting the trade-off captured in the mean and standard deviation results discussed previously.

Task 4: Create a RL agent for Minirace (level 2)

Design Choices

For the level 2 Minirace agent, I expanded upon the Q-learning approach from level 1 to utilize a Deep Q-Network (DQN) with a single hidden layer to approximate the action-value function $Q(s, a)$. In Level 1, the environment provided only a single variable, dx, representing the car's lateral offset from the center, which made a simple tabular Q-learning approach both feasible and effective. However, Level 2 introduces a much richer state space based on visual input frames, which makes it impossible to maintain a Q-table — the number of possible pixel configurations grows exponentially. Instead, the DQN learns to approximate $Q(s, a)$ directly from the images, allowing the agent to generalize between visually similar states and recognize driving-relevant patterns in the input. This shift from a tabular to a neural-network-based approach was necessary to handle the added complexity and visual nature of the task.

In practice, training such a model proved much trickier than in Level 1. The main challenge was stability — early in training, the Q-values tended to fluctuate heavily and sometimes diverged. To address this, I researched and used two core techniques that are standard in DQN: experience replay and a target network. Experience replay helped break the strong temporal correlations between sequential frames by sampling random mini-batches from a replay buffer. This made learning more stable and sample-efficient, since the agent could reuse past experiences. The target network, updated periodically from the policy network, further helped stabilize training by preventing the network from "chasing" its own moving predictions.

The network was trained with the Adam optimizer and a mean-squared TD-loss objective. I initially tried a few other settings — including different learning rates and optimizers like SGD — but found that Adam's adaptive learning rate worked best for the highly non-stationary nature of reinforcement learning. Gradient clipping also helped prevent large updates that could destabilize training, although even with these measures, the loss curve remained somewhat noisy throughout.

An ϵ-greedy exploration policy was used to balance exploration and exploitation, starting with a high ϵ value to encourage random behavior before gradually decaying it over time. I experimented with different decay schedules — linear and exponential — but the linear decay led to smoother performance overall. The architecture itself was kept simple, with one hidden layer, to ensure training remained fast and interpretable while still allowing the agent to approximate a useful policy. Deeper networks were attempted but tended to overfit or plateau early, likely due to limited diversity in the replay buffer and relatively short training times.

Overall, this design process reflected a balance between complexity and stability. While the DQN agent ultimately learned to drive reasonably well, some runs still showed instability or performance drops after long training periods. This suggests that while the approach is sound, more advanced techniques — such as prioritized experience replay or Double DQN — might help further stabilize learning in future experiments.

Reinforcement Learning Architecture

The Deep Q-Learning (DQN) agent was implemented using a feedforward neural network to approximate the action-value function $Q(s, a)$. The network received as input the two-dimensional state vector $s = [dx_1, dx_2]$, corresponding to the car's lateral offset in the current and upcoming track segments. This representation provided the agent with short-range spatial context sufficient for local trajectory planning. The network consisted of three layers:

- **Input layer:** Two neurons corresponding to dx_1 and dx_2.
- **Hidden layer:** 32 neurons with ReLU activation, selected to provide sufficient non-linear capacity to map the continuous state space to discrete Q-values without excessive computational overhead.

- **Output layer:** Three neurons representing the discrete action set $a \in \{-2, 0, 2\}$, with each output corresponding to the predicted Q-value for that action.

The model parameters were optimized using the Adam optimizer with a learning rate of 1×10^{-3}, which provided a good trade-off between fast convergence and numerical stability. The discount factor was set to $\gamma = 0.95$, encouraging the agent to prioritize long-term survival on the track. A batch size of 64 was used for mini-batch updates, providing a balance between gradient stability and computational efficiency.

The training process relied on several key functions:

- `linear_epsilon(start, end, decay_episodes)`: Linearly decayed the exploration rate ϵ from 1.0 to 0.05 over 500 episodes, allowing the agent to transition smoothly from exploration to exploitation.
- `choose_action(Q, s, epsilon)`: Implemented an ϵ-greedy policy, selecting a random action with probability ϵ, or the greedy action $\arg\max_a Q(s, a)$ otherwise.
- `compute_td_loss(batch)`: Calculated the temporal-difference (TD) loss for a batch of sampled transitions (s, a, r, s') from the replay buffer:

$$L = E_{(s,a,r,s')}\left[(r + \gamma \max_{a'} Q_{\text{target}}(s', a') - Q_{\text{policy}}(s, a))^2\right]$$

  This loss was minimized using gradient descent to align the policy network's Q-value predictions with the TD targets computed by a separate target network.

To improve training stability, a target network Q_{target} was maintained and updated every 50 training steps. This reduced oscillations caused by rapidly changing Q-value estimates. Experience replay was also employed using a fixed-size buffer of 10,000 transitions, from which random mini-batches were drawn. This process broke the correlations between sequential experiences and improved sample efficiency. Training was conducted for 1000 episodes, with performance evaluated periodically to monitor convergence. The combination of a compact network architecture, stable TD updates, and controlled exploration allowed the agent to learn a robust steering policy capable of maintaining track alignment across diverse track geometries.

Component	**Value / Setting**	**Rationale**
Input size	2-dimensional (dx_1, dx_2)	Current and next track positions
Hidden layers	1 fully connected, 32 neurons	Sufficient for low-dimensional state, fast training
Activation	ReLU	Nonlinearity, efficient gradient flow
Output layer	3 neurons (Q-values: left, stay, right)	Maps actions to expected return
Output activation	Linear	Q-values unrestricted
Discount factor γ	0.95	Balance immediate vs. long-term reward
Learning rate	5×10^{-3}, Adam	Fast convergence with stability
Epsilon-greedy	ϵ: 1.0 $\rightarrow$ 0.05 over 4000 eps	Explore early, exploit later
Replay buffer	5000 capacity, random sampling	Breaks temporal correlation, stabilizes updates
Batch size	64	Trade-off: stability vs. computation
Min replay before train	500	Enough data for meaningful update
Target network	Update every 100 episodes	Stabilizes TD targets
Loss function	MSE (predicted vs. TD target)	Aligns network with expected reward
Training episodes	10,000	Enough to converge on level 2 track
Running reward	Exponential moving average	Smooth performance tracking
Rendering	Optional	Visual debugging without slowing training
Training time	$\sim$15–20 min (CPU)	Depends on replay usage and rendering

Table 3: Level 2 MiniRace DQN Agent: Architecture and Hyperparameters

Hyperparameter Selection

The hyperparameters were experimented with at length to balance efficient learning with stable convergence in the level-2 Minirace environment. After extensive experimentation, a total of 8,000 episodes seemed to be optimal to ensure that the agent accumulates sufficient experience across a wide variety of track segments, while avoiding unnecessarily long training times. Shorter runs (e.g., 5,000 episodes) occasionally produced unstable policies that overfit to early track segments, whereas longer runs beyond 15,000 episodes offered minimal performance gains relative to the increased computation time.

The discount factor was set to $\gamma = 0.95$, which strikes a balance between prioritizing immediate survival rewards and considering the long-term consequences of actions. Lower values of γ led to myopic policies that prioritized avoiding immediate collisions but failed to plan for upcoming curves, while values closer to 1.0 caused slow learning due to the increased variance in TD targets.

A learning rate of 5×10^{-3} was chosen for the Adam optimizer. Initially, smaller learning rates (1×10^{-3}) were tested but resulted in very slow convergence, while larger rates (1×10^{-2}) caused unstable updates and erratic Q-value estimates. This intermediate rate provided a good compromise, enabling the network to adapt quickly while maintaining numerical stability.

The epsilon-greedy exploration schedule starts fully random at $\epsilon = 1.0$ and decays linearly to $\epsilon = 0.05$ over 4,000 episodes. Alternative schedules, such as exponential decay or faster linear decay, either caused premature exploitation—leading the agent to repeatedly crash in underexplored track regions—or prolonged excessive randomness, delaying convergence.

Experience replay is implemented with a buffer size of 5,000 transitions and a batch size of 64. Smaller buffers led to overfitting on recent transitions, while excessively large buffers (e.g., 20,000) slowed down training due to sampling overhead without significant performance gains. Similarly, smaller batch sizes produced noisy gradient updates, whereas larger batches reduced the frequency of updates and slowed learning. The target network is updated every 100 episodes to stabilize TD targets; more frequent updates caused oscillations, and less frequent updates slowed the propagation of new knowledge.

Finally, evaluation is conducted over 50 episodes, providing a robust estimate of policy performance, while optional rendering allows for qualitative assessment without significantly impacting training time. Together, these hyperparameters reflect a careful trade-off between training efficiency, stability, and performance, informed by both theoretical considerations and empirical experimentation.

Hyperparameter	**Value**	**Rationale / Notes**
Seed	1	Ensures reproducibility
Number of episodes	8,000	Optimal for convergence on Level 2
Discount factor γ	0.95	Balance immediate vs. future rewards
Learning rate	5×10^{-3}	Adam optimizer for stability
Epsilon start	1.0	Full exploration initially
Epsilon end	0.05	Minimal exploration at convergence
Epsilon decay episodes	4,000	Linear decay over early training
Replay buffer size	5,000	Store past transitions for stable learning
Batch size	64	Mini-batch for gradient update
Min replay before training	500	Ensure meaningful gradient updates
Target update interval	100 episodes	Stabilize Q-targets with separate network
Rendering	False / Optional	Visual debugging without slowing training
Test episodes	50	Evaluate learned policy performance

Table 4: Hyperparameters for Level 2 MiniRace DQN Agent

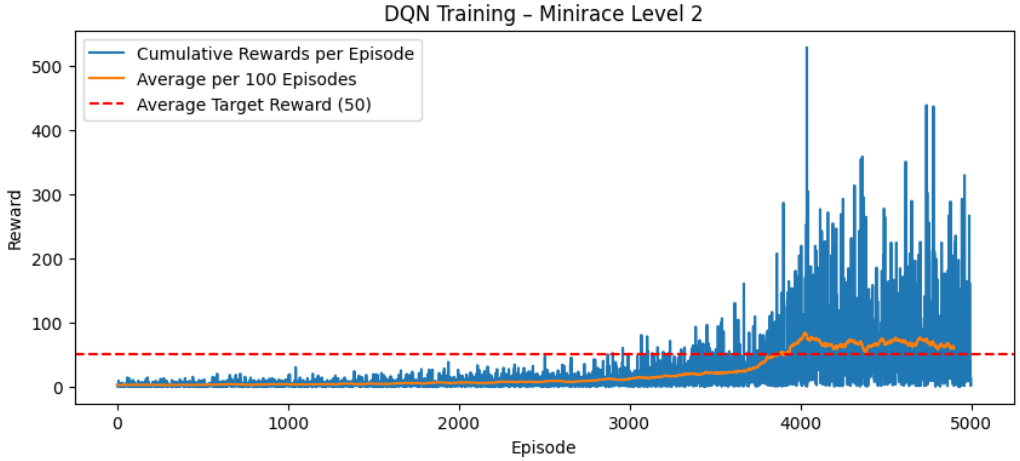
Results

The trained DQN agent was evaluated using a greedy policy over multiple episodes for three different training durations: 5000, 8000, and 10000 episodes. The results for cumulative reward per episode are summarized in Table 5.

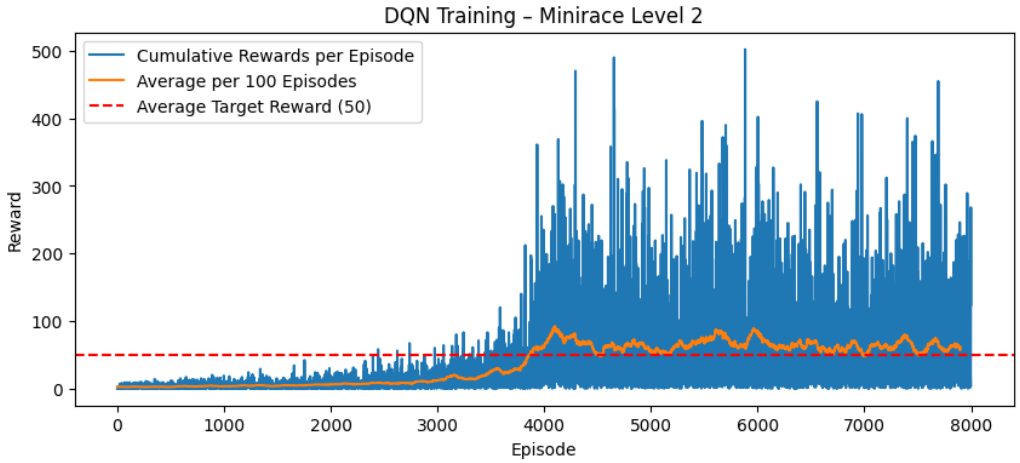
Training Episodes	**Mean Reward**	**Std. Dev.**	**Runtime**
5000	264.22	185.04	2 min
8000	500.00	0.00	10 min
10000	180.10	160.62	15 min

Table 5: Test results (greedy policy) for different numbers of training episodes.

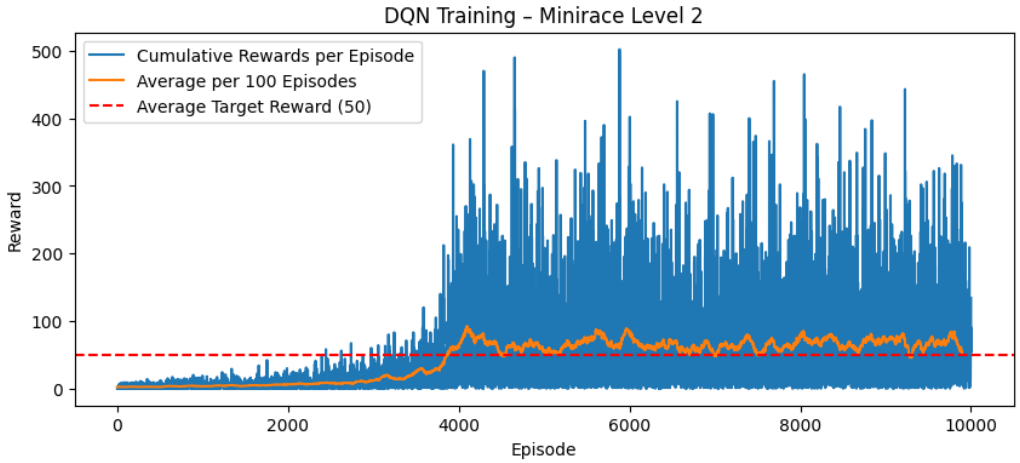
The results indicate that training for 8000 episodes achieves the best performance, with the agent reaching the maximum mean reward of 500 and zero variance. This demonstrates that the policy learned is fully consistent, allowing the agent to reliably complete episodes without failure. Training for 5000 episodes produces a



(a) 5000 Episodes



(b) 8000 Episodes



(c) 10000 Episodes

Figure 7: Training results for different numbers of episodes. Each plot shows cumulative rewards per episode.

moderate mean reward with high variance, suggesting the agent has partially learned effective strategies but is still prone to errors. In contrast, training for 10000 episodes results in lower mean rewards and high variance, likely due to overfitting or oscillations in Q-value estimates when training continues beyond the optimal duration.

Figure 7 presents the cumulative reward per episode for each training duration, stacked vertically for clarity. Across all durations, episode lengths generally increase in tandem with rewards, reflecting the agent's ability

to remain on track for longer periods as learning progresses. For the 8000-episode training, episode lengths reach the maximum allowed, consistent with the perfect mean reward observed. The 5000- and 10000-episode trainings display more variability in episode lengths, mirroring the fluctuations in rewards and highlighting under-training and over-training effects.

The observed high variance in both 5000- and 10000-episode runs can perhaps be partially explained by the chosen epsilon decay schedule and replay buffer configuration. With a 4000-episode epsilon decay, early training experiences involve extensive exploration, which may cause inconsistent behavior in shorter training durations (5000 episodes). Conversely, when training extends to 10000 episodes, the agent may start overfitting to frequently replayed transitions in the fixed-size replay buffer, reinforcing suboptimal patterns and increasing performance variability. This underscores the importance of carefully balancing exploration, replay buffer size, and training duration to achieve a stable and generalizable policy.

Overall, these results demonstrate that Deep Q-Learning with a single hidden layer, experience replay, and a target network is effective for MiniRace Level 2. The experiments highlight the existence of an optimal training duration (here, around 8000 episodes) that balances exploration, policy convergence, and generalization. Episode length trends provide an additional metric for understanding agent learning progress and robustness, complementing cumulative reward measurements.

Acknowledgment of AI Assistance

Parts of the code were edited with the assistance of an AI language model (ChatGPT by OpenAI), but all design choices were my own. The AI provided suggestions for syntax errors and clarity improvements, but all technical content, analyses, and interpretations were done by me.