Imitation Learning for Path Following Using NVIDIA JetBot

A Hands-On Project Report

Authors:

Adel Saidani – U6104239
Enis Hedri –
Mahra Alhosani – U1100303

Supervisors:

Dr. Palomeras Rovira, Narcis
Real Vial, Marta

Master in Intelligent Robotic Systems (MIRS)

January 28, 2026

Abstract

This report presents the development of an autonomous path-following system for the NVIDIA JetBot using imitation learning. The goal was to enable the robot to navigate a track using only RGB camera input, learning driving behavior directly from human demonstrations. We employed behavioral cloning with a ResNet18 convolutional neural network, training the model to predict steering commands from single-frame images. The initial model achieved reliable path following, and cornering performance was further improved using Dataset Aggregation (DAgger). Building on this success, we extended the system to predict both steering and speed, allowing the robot to slow down in corners and accelerate on straight segments. The final model was trained on approximately 11,000 images and achieved over 5 minutes of continuous autonomous driving without errors. This project demonstrates the effectiveness of behavioral cloning for real-world robotic control tasks and highlights the importance of iterative refinement techniques such as DAgger for improving model robustness. All development was conducted using PyTorch, with training performed on a PC and deployment on the JetBot's onboard Jetson Nano.

Contents

Supplementary Materials

The source code, trained models, and demo videos are available:

- **GitHub:** `https://github.com/AdelSaidani/Imitation-Learning-for-Path-Following-Usi tree/main`
- **Demo Video:** `https://drive.google.com/drive/folders/1SANZlyvb_CsEvBK4kqLl-YiQpiO9 usp=drive_link`

1 Introduction

Autonomous navigation is a fundamental capability for mobile robots, with applications ranging from warehouse logistics to self-driving vehicles. A core challenge in autonomous navigation is enabling a robot to perceive its environment and make appropriate control decisions in real-time. Traditional approaches often rely on hand-crafted rules or complex sensor fusion, which can be difficult to design and tune for varying conditions.

Imitation learning offers an alternative approach where a robot learns to replicate the behavior of a human demonstrator. Rather than explicitly programming control logic, the robot observes expert demonstrations and learns a mapping from sensory input to control commands. This approach is particularly attractive for resource-constrained platforms, where the computational requirements of reinforcement learning or classical planning algorithms may exceed the available hardware capabilities.

In this project, we developed an autonomous path-following system for the NVIDIA JetBot, a small differential-drive robot equipped with a camera and powered by the Jetson Nano. Behavioral cloning, a form of imitation learning that treats the problem as supervised learning, is well-suited for this platform as it requires only offline training and efficient inference.

1.1 Task Definition

Goal: Enable the JetBot to autonomously follow a track using only RGB camera images as input, predicting both steering angle and forward speed.

Robot: NVIDIA JetBot—a differential-drive platform powered by the Jetson Nano (4GB), equipped with an IMX219 RGB camera.

Observation Vector: Single RGB image ($224\times224\times3$), normalized using ImageNet statistics: $\mu = [0.485, 0.456, 0.406]$, $\sigma = [0.229, 0.224, 0.225]$.

Action Space: Two continuous outputs:

- Steering: $[-1, +1]$ where negative values indicate left turns and positive values indicate right turns
- Speed factor: $[0, 1]$ scaled to actual motor speed during deployment

Algorithm: Behavioral Cloning (BC)—a supervised learning approach that minimizes the mean squared error between predicted and demonstrated actions.

1.2 Contributions

The main contributions of this project are:

- A complete behavioral cloning pipeline for the JetBot, including data collection, image preprocessing, model training, and real-time deployment
- Implementation of DAgger for iterative policy improvement to address distribution shift
- Extension to dual-output prediction (steering and speed) for adaptive velocity control
- Experimental validation demonstrating over 5 minutes of continuous autonomous driving

2 State of the Art

2.1 Imitation Learning and Behavioral Cloning

Imitation learning is a paradigm in which an agent learns to perform a task by observing demonstrations from an expert. Unlike reinforcement learning, which requires the agent to explore the environment and learn from reward signals, imitation learning leverages existing expertise to bootstrap the learning process. This makes it particularly suitable for tasks where defining a reward function is difficult or where exploration could be dangerous or expensive.

The simplest form of imitation learning is *behavioral cloning*, which frames the problem as supervised learning. Given a dataset of expert demonstrations consisting of state-action pairs $\{(s_i, a_i)\}_{i=1}^{N}$, the goal is to learn a policy $\pi_\theta(s)$ that minimizes the difference between predicted actions and expert actions:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|\pi_\theta(s_i) - a_i\|^2 \tag{1}$$

In our context, the state s is an RGB image from the robot's camera, and the action a consists of steering and speed commands. The policy π_θ is implemented as a convolutional neural network.

2.2 Distribution Shift and DAgger

A fundamental challenge in behavioral cloning is *distribution shift*. During training, the model learns from states visited by the expert. However, during deployment, small prediction errors can cause the robot to visit states that differ from the training distribution. Since the model has not seen these states, it may make poor predictions, leading to compounding errors.

Dataset Aggregation (DAgger) [3] addresses this problem by iteratively collecting data in states visited by the learned policy. The algorithm proceeds as follows:

1. Train an initial policy π_1 on expert demonstrations $\mathcal{D}$
2. For iterations $i = 1, 2, \ldots, N$:

(a) Execute policy π_i to collect new states

(b) Query the expert for correct actions at these states

(c) Aggregate new data with $\mathcal{D}$

(d) Train policy π_{i+1} on aggregated dataset

By training on states actually visited by the learned policy, DAgger reduces the distribution mismatch and improves robustness.

2.3 End-to-End Learning for Autonomous Driving

The idea of learning driving behavior directly from images has a long history. ALVINN (Autonomous Land Vehicle In a Neural Network) [1] demonstrated in 1989 that a neural network could learn to steer a vehicle from camera images. More recently, NVIDIA's PilotNet [2] showed that a convolutional neural network trained end-to-end on human driving data could learn to steer a real car, processing raw pixels directly into steering commands.

These approaches share a common philosophy: rather than decomposing the driving task into perception, planning, and control modules, they learn a direct mapping from sensory input to control output. This end-to-end approach simplifies the system architecture and allows the model to learn features that are directly relevant to the control task.

2.4 Convolutional Neural Networks for Visual Control

Convolutional neural networks (CNNs) are well-suited for extracting spatial features from images. For visual control tasks, CNNs can learn hierarchical representations—from low-level edges and textures to high-level semantic features like lane boundaries and road geometry.

ResNet [4] introduces residual connections that enable training of deeper networks by mitigating the vanishing gradient problem. ResNet18, a relatively compact variant with 18 layers, provides a good balance between representational capacity and computational efficiency. Pre-trained on ImageNet [7], ResNet18 offers learned low-level features (edges, textures, colors) that transfer well to new visual tasks, reducing the amount of domain-specific training data required.

3 Methodology

3.1 Hardware Platform

The JetBot is an open-source robot platform designed by NVIDIA for education and research in AI and robotics. Our setup consists of:

- **Compute Module**: NVIDIA Jetson Nano (4GB), featuring a quad-core ARM Cortex-A57 CPU and 128-core Maxwell GPU capable of running neural network inference in real-time
- **Camera**: IMX219 RGB camera module (8 megapixels), mounted at the front facing forward, capturing at 640×480 resolution

- **Motors**: Two DC motors in a differential drive configuration
- **Controller**: Logitech F710 analog gamepad for teleoperation and data collection

The track consists of a black surface with white boundary lines and red corner markings to provide visual cues for distinguishing turns from straight segments.

3.2 Data Collection

Data collection was performed by manually driving the JetBot around the track using the analog joystick controller. During teleoperation, we recorded synchronized pairs of camera images and control commands at approximately 20 Hz.

The control scheme used the following mapping:

- **Left Stick (X-axis)**: Steering command, ranging from -1 (full left) to $+1$ (full right)
- **Right Trigger (RT)**: Speed factor, ranging from 0 (minimum speed) to 1 (maximum speed)
- **Right Bumper (RB)**: Hold to enable driving and recording

Each recorded image was saved with its corresponding labels encoded in the filename:

`{timestamp}_{steering}_{speed_factor}.jpg`

3.3 Image Preprocessing

Raw camera images (640×480 pixels) contain regions that are not relevant for navigation. We applied a cropping operation **during data collection** to focus on the road region immediately in front of the robot:

- **Top**: 20% removed (eliminates distant/irrelevant areas)
- **Bottom**: 0% removed (keeps the near road surface)
- **Left**: 8% removed (removes non-essential periphery)
- **Right**: 12% removed (removes non-essential periphery)

After cropping, images were resized to 224×224 pixels to match the input size expected by ResNet18. Finally, pixel values were normalized using ImageNet statistics.

3.4 Data Augmentation

Before training, we applied data augmentation to improve generalization and robustness to varying lighting conditions. The augmentation pipeline consisted of random color jittering:

- **Brightness**: ±30%
- **Contrast**: ±30%

- **Saturation**: ±30%
- **Hue**: ±5%

These augmentations simulate different lighting conditions without altering the geometric content of the image, ensuring that the steering and speed labels remain valid.

3.5 Neural Network Architecture

We used ResNet18 as our base architecture, leveraging pre-trained ImageNet weights for transfer learning. The original ResNet18 classification head (1000 classes) was replaced with a regression head for our control outputs.

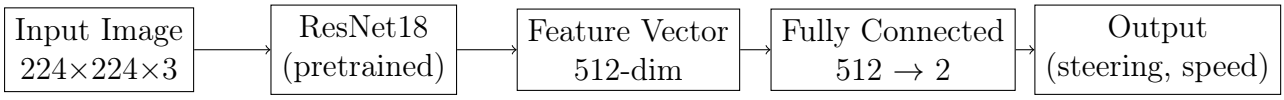


Figure 1: Model architecture: ResNet18 backbone with custom regression head

Architecture Details:

- Backbone: ResNet18 (pretrained on ImageNet)
- Final layer replaced: $512 \rightarrow 2$ neurons (steering, speed factor)
- Total parameters: 11.18 million
- All parameters trainable

The speed factor output is scaled to actual motor speed during deployment:

$$v_{\text{actual}} = v_{\min} + \text{speed\_factor} \times (v_{\max} - v_{\min}) \tag{2}$$

where $v_{\min} = 0.12$ and $v_{\max} = 0.25$.

3.6 Training Setup

Parameter	**Value**
Optimizer	Adam
Learning Rate	1×10^{-4}
Batch Size	8
Weight Decay	1×10^{-5}
Train/Validation Split	80% / 20%
Early Stopping Patience	7 epochs
LR Scheduler	Reduce by 0.5× after 3 plateau epochs

Table 1: Training hyperparameters

The loss function combines mean squared error (MSE) for both outputs with equal weights:

$$\mathcal{L} = \text{MSE}(\hat{y}_{\text{steering}}, y_{\text{steering}}) + \text{MSE}(\hat{y}_{\text{speed}}, y_{\text{speed}}) \tag{3}$$

Training was performed on a PC with GPU (35 minutes for the final model), and the trained model was transferred to the JetBot for deployment.

3.7 DAgger Implementation

To address distribution shift, we implemented DAgger with the following process:

1. Deploy the trained model on the JetBot for autonomous driving
2. When the model makes errors (particularly in corners), interrupt by taking manual control
3. While teleoperating, continue recording camera frames with corrective steering/speed commands
4. Merge the new correction data with the original training dataset
5. Retrain the model on the combined dataset

This approach specifically targets the states where the learned policy struggles, providing expert guidance for those challenging situations.

3.8 Deployment

The trained model was deployed on the JetBot for real-time inference at approximately 20 Hz. Motor commands are computed via differential drive:

$$\begin{aligned} v_{\text{left}} &= v_{\text{actual}} + \text{steering} \times k_{\text{gain}} \\ v_{\text{right}} &= v_{\text{actual}} - \text{steering} \times k_{\text{gain}} \end{aligned} \tag{4}$$

where $k_{\text{gain}} = 0.08$ is a steering gain parameter.

4 Results

4.1 Dataset Statistics

Three datasets were collected during development:

Dataset	Total Images	Training Set	Validation Set
Initial (steering only)	8,314	6,651	1,663
+ DAgger corrections	8,497	6,797	1,700
+ Speed control (final)	11,039	8,831	2,208

Table 2: Dataset sizes across development iterations (80/20 train/validation split)

Figure 2 shows the distribution of steering and speed values in the final dataset. The speed factor distribution is bimodal with peaks at 0 (slow/corners) and 1 (fast/straights), reflecting the track layout.

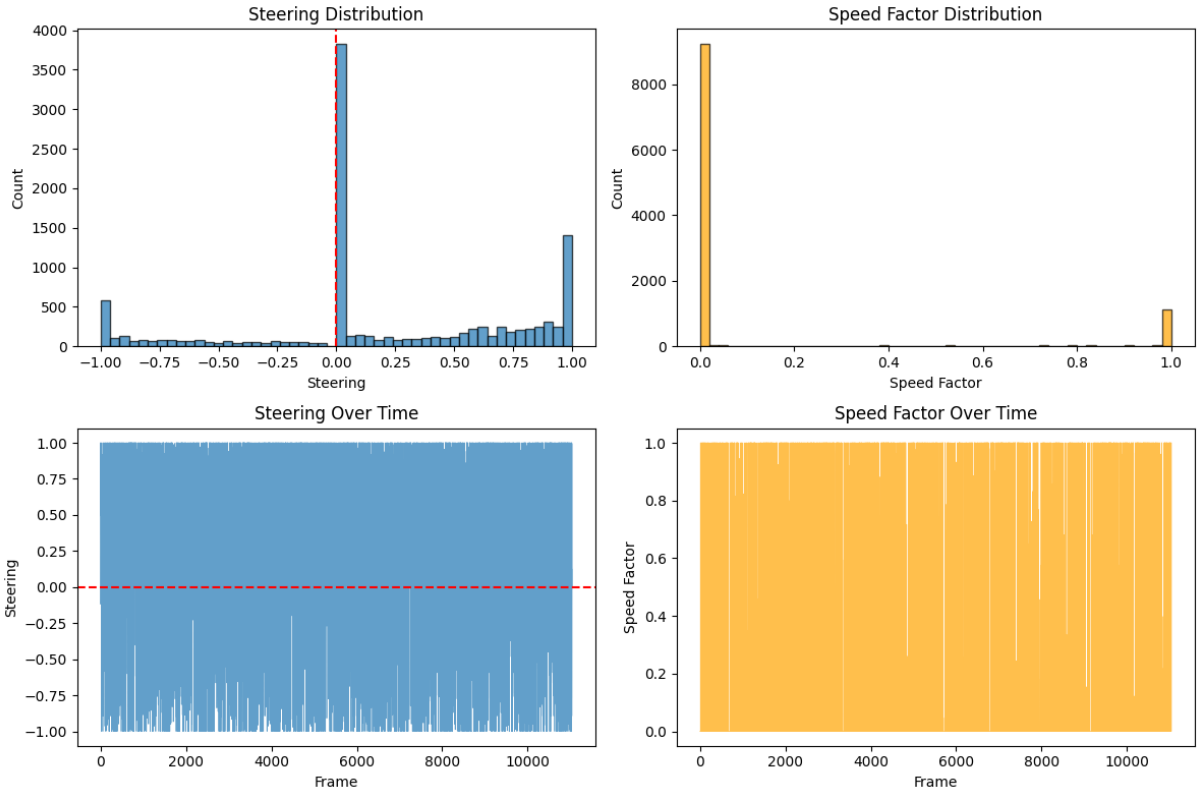


Figure 2: Distribution of steering and speed factor values in the final dataset. Top: histograms. Bottom: values over time showing correlation between steering activity and reduced speed.

4.2 Training Results

The final dual-output model was trained for 34 epochs before early stopping, achieving a best validation loss of 0.0398. Figure 3 shows the training dynamics.

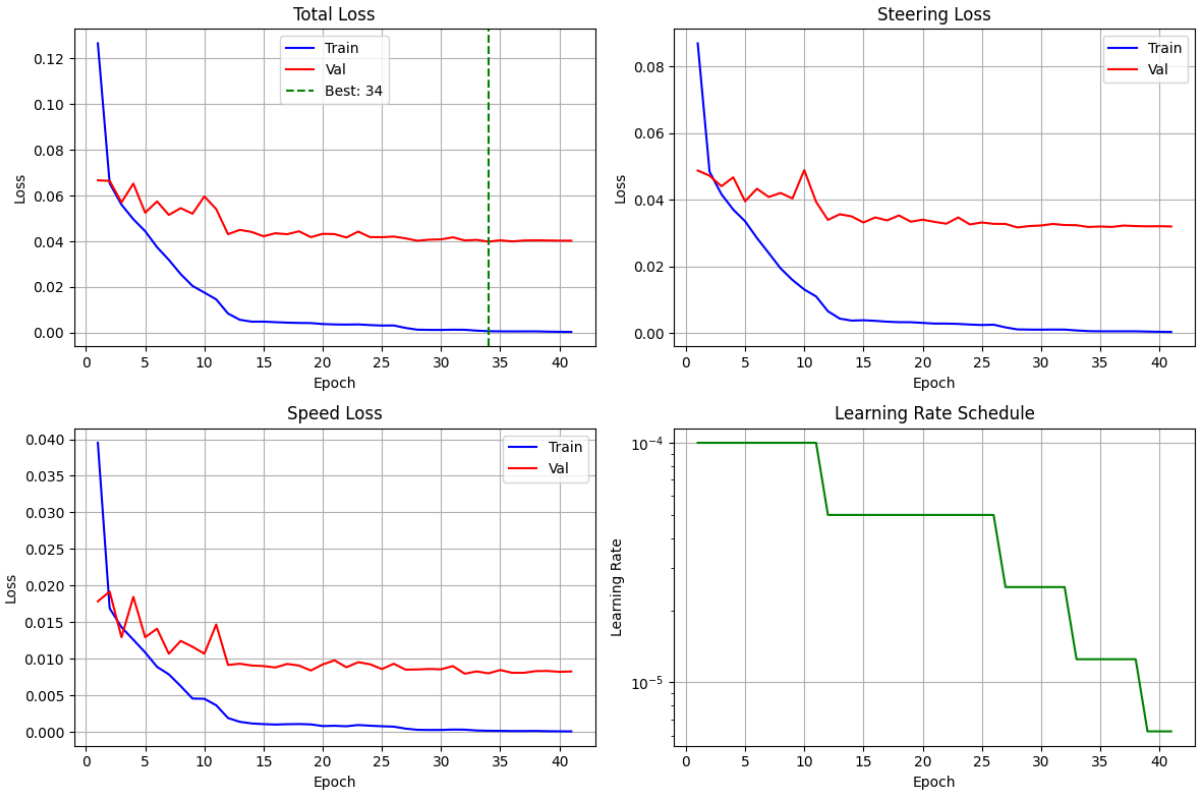


Figure 3: Training dynamics for the dual-output model. Top-left: total combined loss. Top-right: steering loss. Bottom-left: speed loss. Bottom-right: learning rate schedule.

Metric	**Steering Output**	**Speed Output**
Test MSE	0.0318	0.0080
Test MAE	0.1106	0.0293

Table 3: Final model performance metrics

Speed prediction achieves notably lower error (MAE 0.0293) than steering (MAE 0.1106). This is expected because speed primarily correlates with track geometry (corners vs straights), while steering requires precise lateral positioning.

4.3 Prediction Quality

Figure 4 shows the prediction quality on the validation set. Both outputs show near-zero bias (steering mean = 0.007, speed mean = 0.001), confirming unbiased predictions.

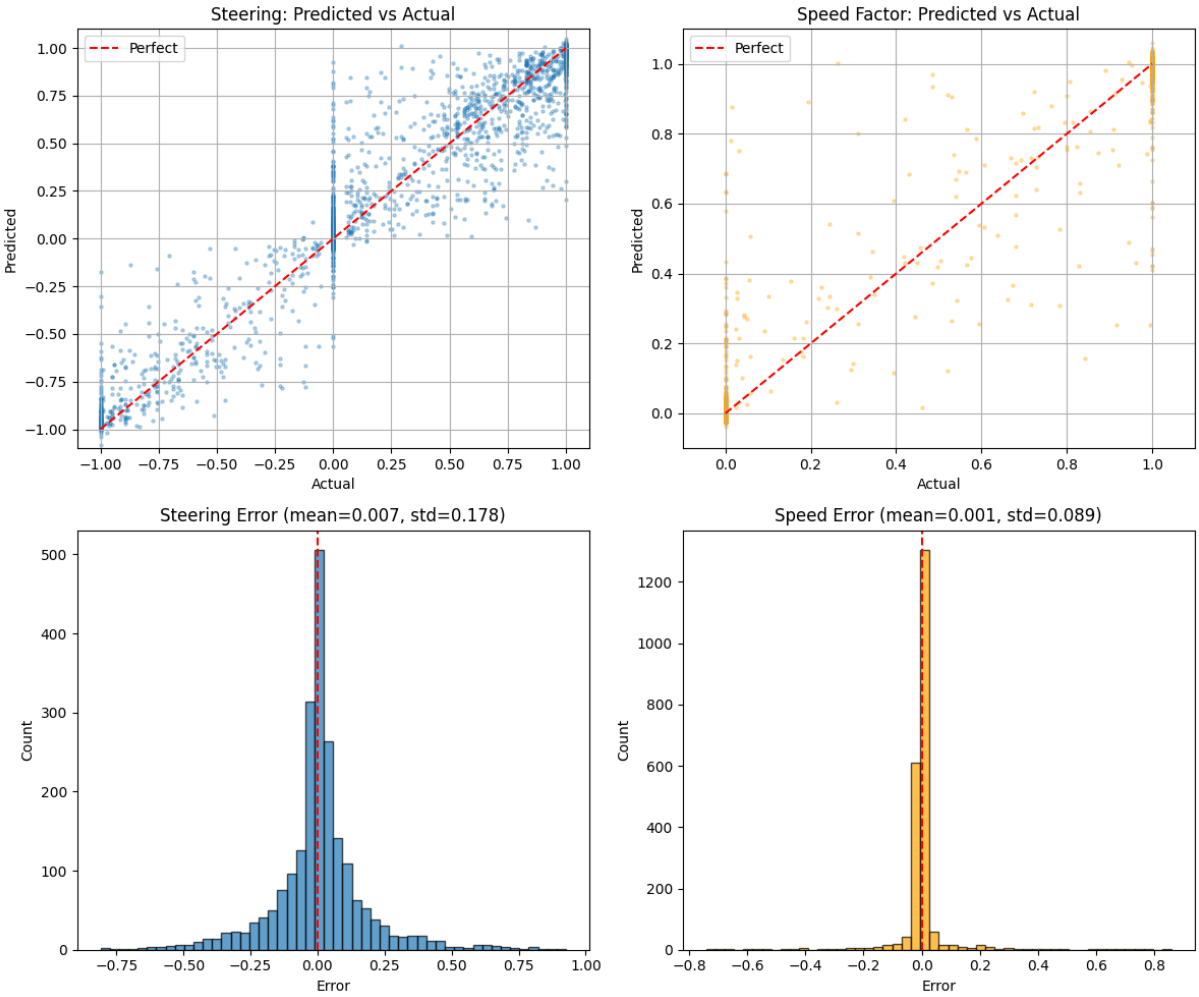


Figure 4: Predicted vs actual plots and error distributions for steering (left) and speed (right).

4.4 Telemetry Analysis

Figure 5 shows telemetry traces comparing ground truth with model predictions. The model successfully tracks both steering and speed signals.

Figure 5: Telemetry comparison (frames 0-150). Top: steering. Middle: speed factor. Bottom: per-frame errors.

4.5 DAgger Impact

Metric	Before DAgger	After DAgger
Validation MSE	0.0265	0.0265
Test MAE	0.0917	0.0892
Corner success rate	Frequent failures	Reliable

Table 4: DAgger comparison: minimal metric change but significant real-world improvement

Key Finding: While aggregate metrics improved by only 2.7%, real-world corner handling improved dramatically. This demonstrates that standard validation metrics may not capture improvements in safety-critical edge cases.

4.6 Model Interpretability

Saliency maps reveal what visual features the model uses for predictions. Figure 6 shows the model focuses on track boundaries rather than background objects, confirming that preprocessing was effective.

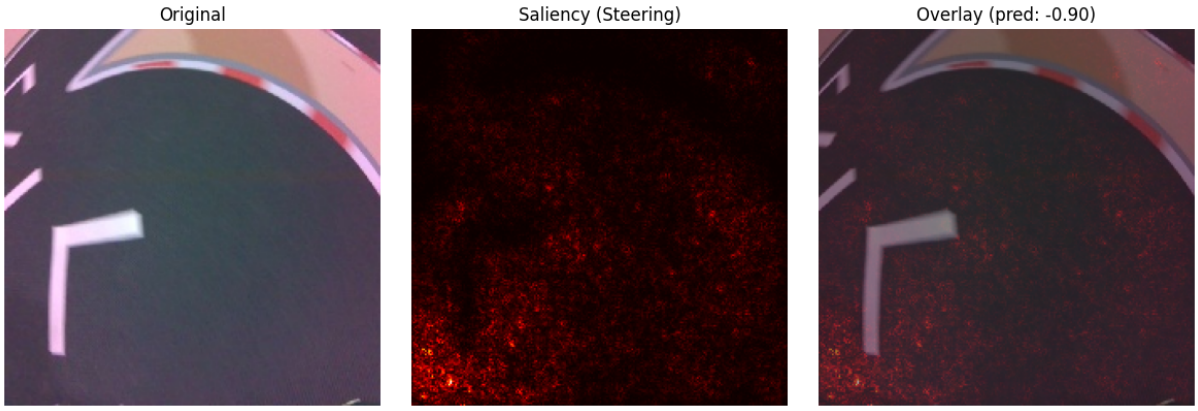


Figure 6: Saliency map for steering prediction. The model attends to track boundaries to determine steering correction.

4.7 Real-World Performance

The final dual-output model achieved:

- **Continuous autonomous driving**: Over 5 minutes without intervention
- **Successful laps**: Multiple complete laps of the test track
- **Speed adaptation**: Slows in corners ($v_{min} \approx 0.12$), accelerates on straights ($v_{max} \approx 0.25$)
- **Inference rate**: Consistent 20 Hz on Jetson Nano
- **First-attempt success**: The dual-output model worked correctly on its first deployment

5 Discussion

5.1 Problems Encountered

During development, we encountered several challenges that informed our final design:

1. **Grayscale images without cropping**: Our initial approach failed to generalize because the model learned background features (walls, furniture) rather than track features. Cropping was essential.
2. **Digital joystick**: Discrete steering values (-1, 0, $+1$) caused jerky motion and poor demonstrations. Switching to analog control enabled smooth, continuous steering.
3. **Frame buffer (temporal context)**: We hypothesized that stacking multiple frames would help, but this approach failed due to latency and memory constraints on the Jetson Nano. Single-frame input proved superior.
4. **Distribution shift**: The initial model failed on corners not seen during training. DAgger effectively addressed this by collecting targeted corrections.

The complete iterative development process is documented in Appendix A.

5.2 Key Insights

Single-frame sufficiency: Contrary to intuition, single-frame input outperformed multi-frame temporal approaches. For reactive control on embedded hardware, latency matters more than temporal context. The continuous visual feedback from the track makes explicit memory unnecessary.

Preprocessing is critical: Image cropping alone dramatically improved generalization by forcing the model to focus on relevant track features rather than spurious background correlations.

Metrics vs reality: DAgger improved MAE by only 2.7% but transformed corner handling from unreliable to robust. Aggregate metrics weight all samples equally, but real-world performance is dominated by worst-case scenarios. This has important implications for evaluating safety-critical robotic systems.

Transfer learning efficiency: ImageNet-pretrained ResNet18 enabled effective learning from only 11,000 images in 35 minutes of training. The pretrained features (edges, textures, colors) transfer well to track-following despite the domain difference.

5.3 Limitations

- **Track specificity**: The model was trained and tested on a single track configuration. Generalization to different tracks was not evaluated.
- **Static environment**: No obstacle detection capability—the model assumes a clear track.
- **Manual speed tuning**: The speed range (v_{min}, v_{max}) was determined empirically rather than learned.
- **Lighting dependency**: While augmentation improved robustness, extreme lighting conditions were not tested.

6 Conclusions

This project successfully developed an autonomous path-following system for the NVIDIA JetBot using behavioral cloning. The key findings are:

1. **Behavioral cloning is effective**: With approximately 11,000 images and 35 minutes of training, we achieved robust autonomous navigation on a resource-constrained platform.
2. **Preprocessing is essential**: Image cropping to remove irrelevant regions and data augmentation for lighting invariance proved critical for generalization.
3. **DAgger addresses distribution shift**: Dataset Aggregation successfully improved corner handling with minimal additional data collection, despite showing only marginal improvement in validation metrics.
4. **Single-frame input is sufficient**: Temporal approaches (frame buffers) failed on embedded hardware due to latency. Reactive control with single-frame input proved both simpler and more effective.

5. **Dual-output enables adaptive control**: Predicting both steering and speed allowed the robot to slow for corners and accelerate on straights, resulting in smoother and more capable driving.

The final system achieved over 5 minutes of continuous autonomous driving without human intervention, working correctly on its first deployment attempt.

6.1 Future Work

- **Domain adaptation**: Techniques like domain randomization could improve generalization to new tracks and environments
- **Obstacle detection**: Adding object detection would enable safe navigation in dynamic environments
- **Reinforcement learning fine-tuning**: The BC policy could serve as initialization for RL to further optimize performance
- **End-to-end speed learning**: The speed bounds could be learned rather than manually specified

References

[1] D. A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, pages 305–313, 1989.

[2] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[3] S. Ross, G. J. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 627–635, 2011.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[5] NVIDIA. JetBot: An open-source robot based on NVIDIA Jetson Nano. `https://github.com/NVIDIA-AI-IOT/jetbot`, 2019.

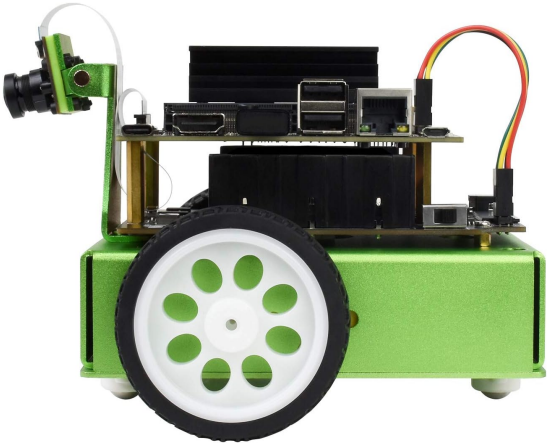
[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

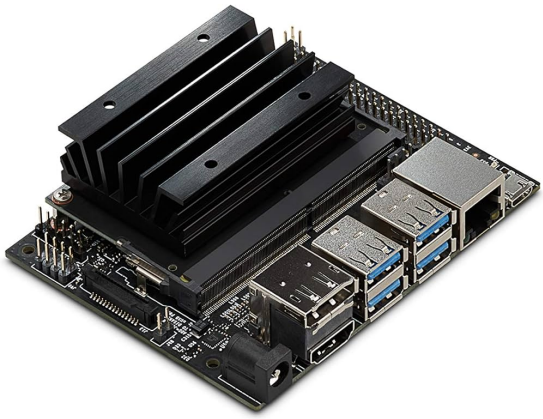
A Iterative Development Process

This appendix documents our complete iterative development process, including failed approaches that informed our final design.

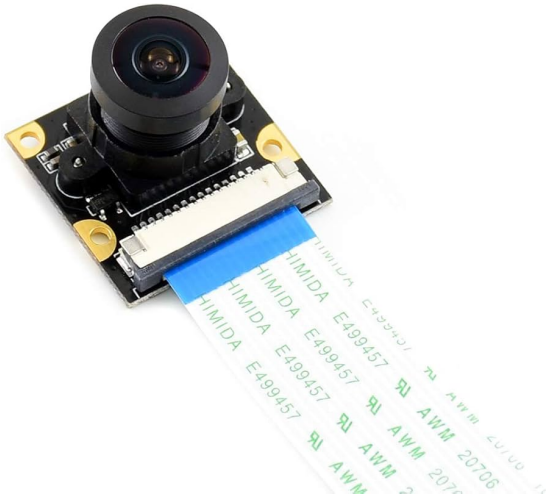
A.1 Hardware Setup



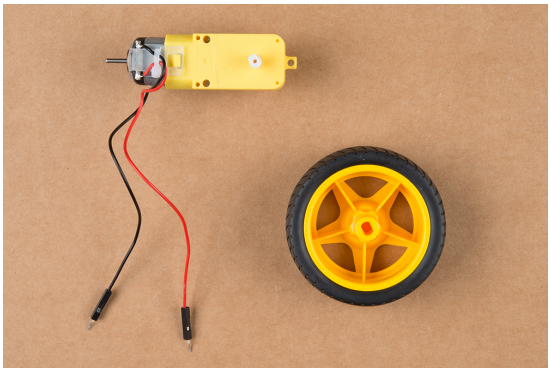
(a) Complete JetBot assembly



(b) NVIDIA Jetson Nano



(c) IMX219 camera module



(d) DC motors and wheels

Figure 7: JetBot platform components

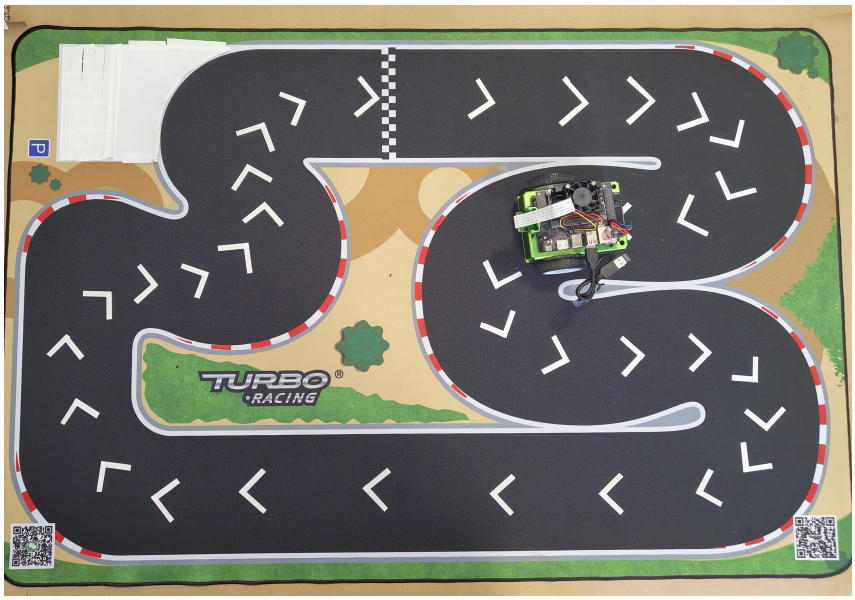


Figure 8: Track layout with black surface, white boundaries, and red corner markings

A.2 Iteration 1: Grayscale with Digital Control

Our first implementation used grayscale images with a digital joystick where steering values could only be -1, 0, or $+1$. The model used constant speed with steering-only prediction.

Problems identified:

- No image cropping: raw images included background objects that the model learned to associate with steering commands
- Discrete steering: caused aggressive, jerky movements
- Grayscale: lost color information (red corner markings)

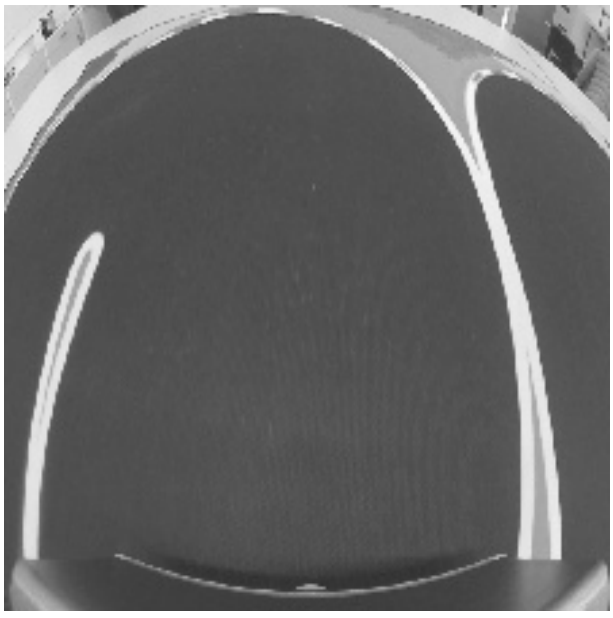


Figure 9: Uncropped grayscale image showing background objects that confused the model

Result: Worked in training environment but failed to generalize.

A.3 Iteration 2: RGB with Analog Control

We switched to RGB images and analog joystick for continuous steering values.

Result: Improved smoothness, but corners remained problematic.

A.4 Iteration 3: Frame Buffer (Failed Experiment)

We hypothesized that temporal context would help. We implemented a frame buffer that stacked 3 consecutive RGB frames (9 channels total).

Problems:

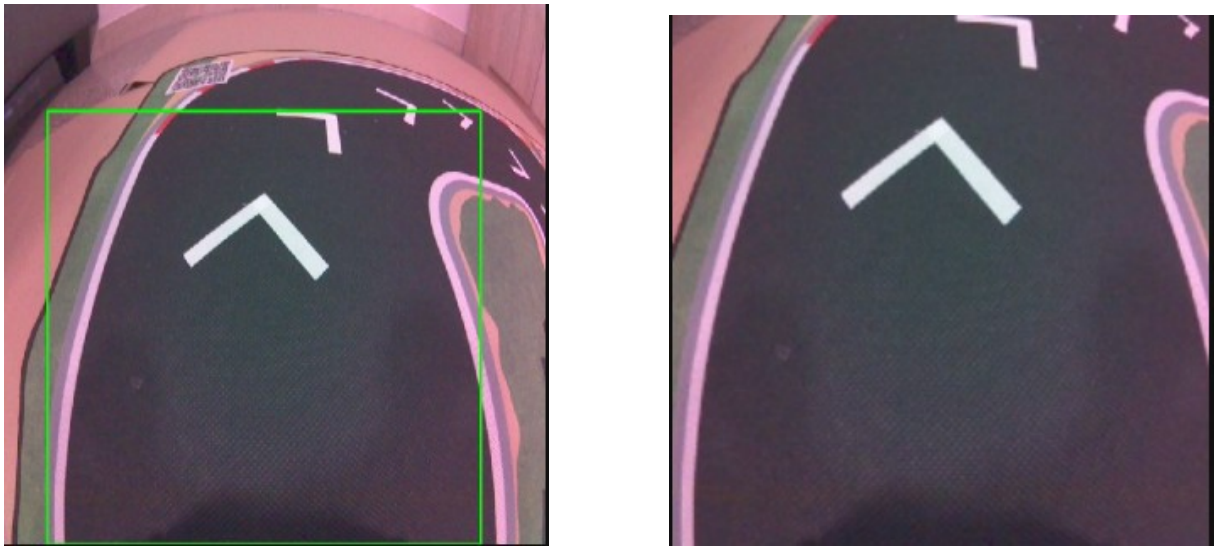
- Latency: responses felt delayed, critical in corners
- Memory: Jetson Nano RAM maxed out
- No benefit: temporal information didn't help

Result: Performance degraded. We concluded single-frame input is superior for real-time control on embedded hardware.

A.5 Iteration 4: Preprocessing and Augmentation

We implemented proper image preprocessing:

- Cropping during data collection (top 20%, left 8%, right 12%)
- Color jitter augmentation before training



(a) Raw image (640×480) (b) Cropped image (224×224)

Figure 10: Preprocessing pipeline: cropping removes irrelevant regions

Result: Dramatic improvement in generalization.

A.6 Iteration 5: DAgger for Corner Improvement

With preprocessing in place, we applied DAgger to fix remaining corner failures:

1. Run model autonomously
2. Intervene when it fails (corners)
3. Record corrective actions
4. Merge with dataset and retrain

Result: Cornering became reliable. Added 183 targeted correction samples.

A.7 Iteration 6: Speed Control

Extended the model to predict both steering and speed factor.

Before collecting data, we determined safe operating speeds using image blur (Laplacian variance):

- $v_{min} = 0.12$ (safe for corners)
- $v_{max} = 0.25$ (before significant motion blur)

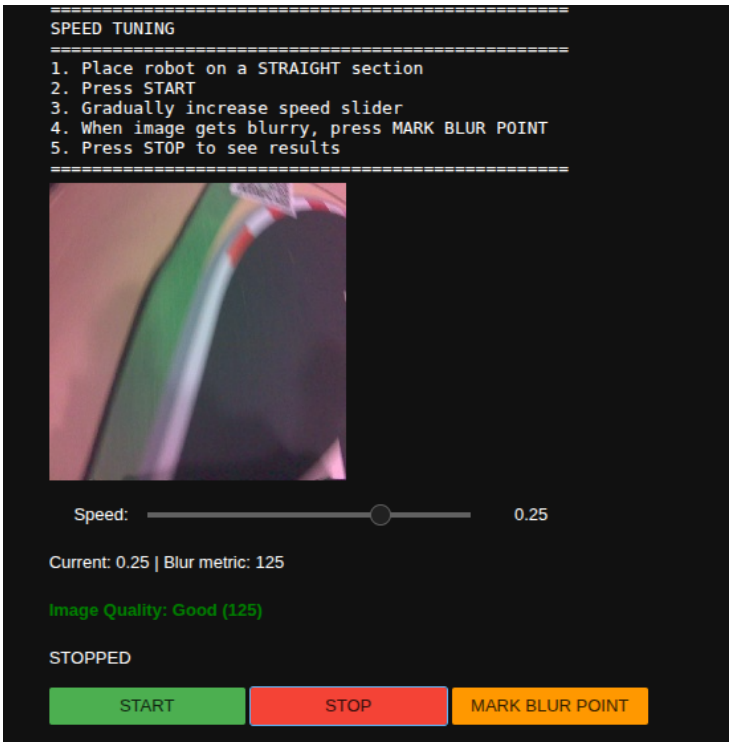


Figure 11: Speed tuning: monitoring image blur at different speeds

Result: Worked on first deployment attempt!

A.8 Summary Table

Iteration	Changes	Outcome
1. Grayscale + Digital	Grayscale, discrete steering, no crop	Failed to generalize
2. RGB + Analog	Color images, continuous steering	Better but corners failed
3. Frame Buffer	3 frames stacked (9 channels)	Worse: latency + memory
4. Preprocessing	Cropping + augmentation	Major improvement
5. DAgger	Collected corner corrections	Corners fixed
6. Speed Control	Dual output (steering + speed)	First-try success

Table 5: Complete iteration summary

A.9 Additional Training Plots

A.9.1 Steering-Only Model

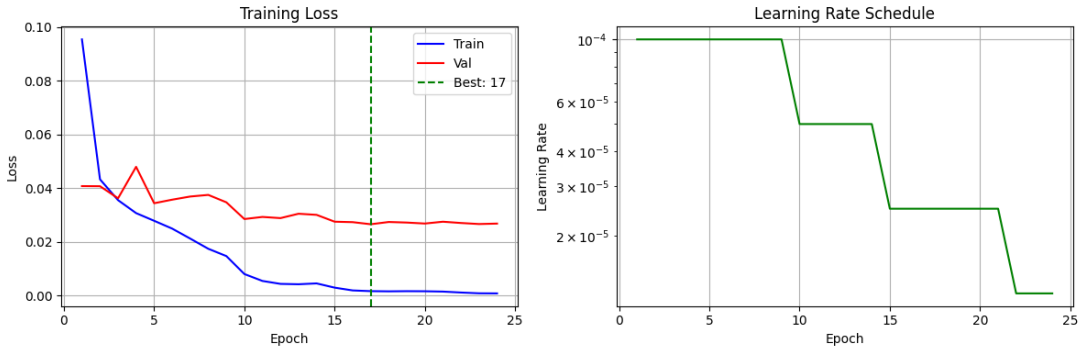


Figure 12: Training curves for steering-only model (best at epoch 17)

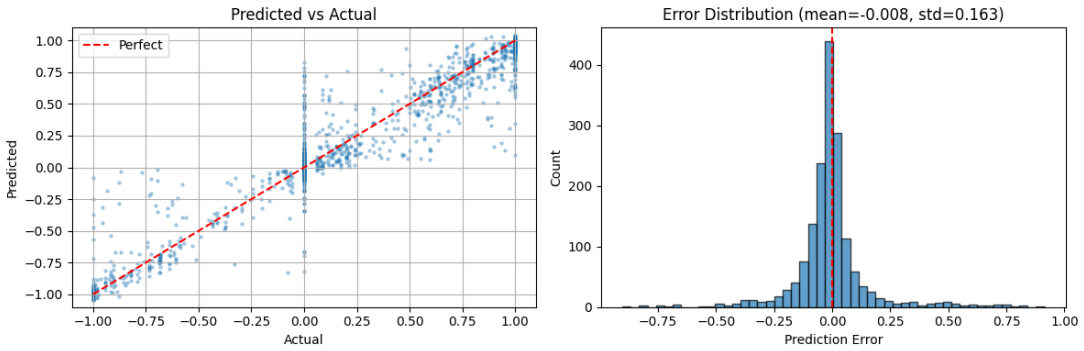


Figure 13: Prediction quality for steering-only model

A.9.2 DAgger Model

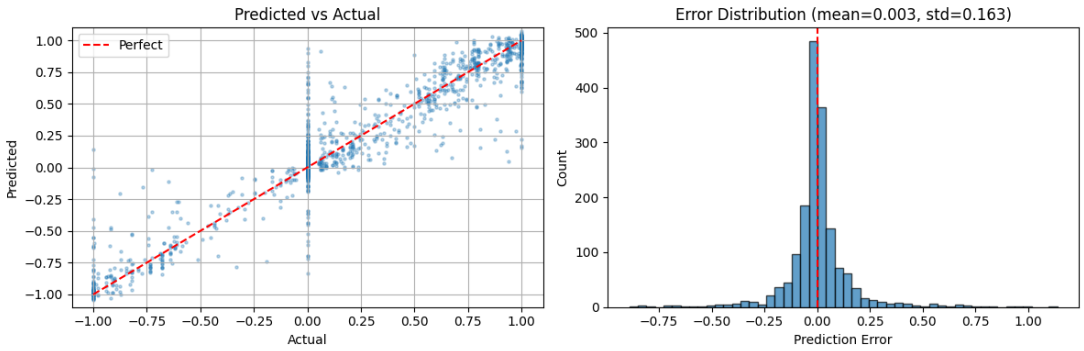


Figure 14: Prediction quality for DAgger-enhanced model

A.9.3 Additional Telemetry

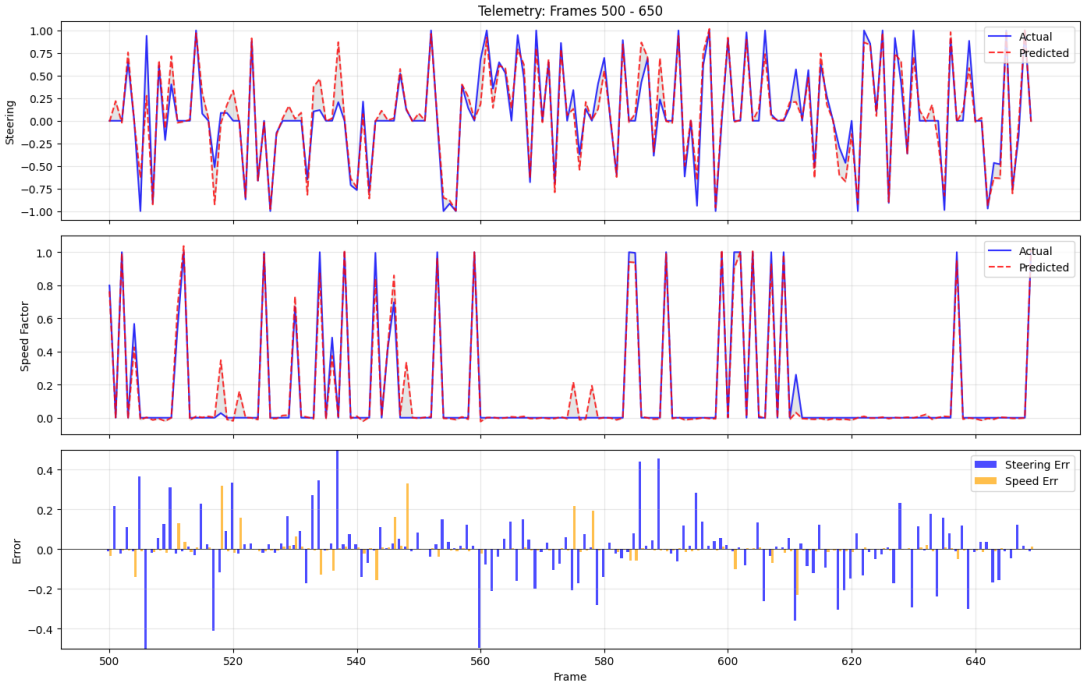


Figure 15: Telemetry for dual-output model (frames 500-650)

A.9.4 Saliency Analysis for Speed

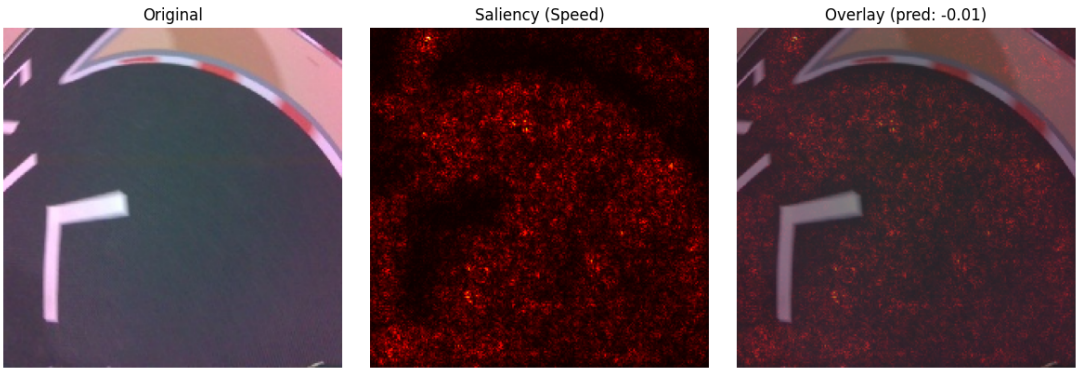


Figure 16: Saliency map for speed prediction—model attends to track curvature