University of British Columbia

ENPH 353

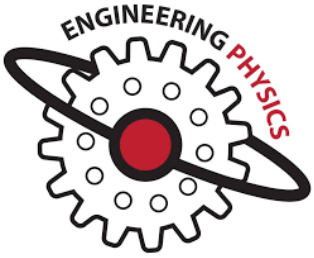
Fizz Detective Machine Learning Project

ENPH 353 Final Report

Abigail Adam Danny Elizur

Team 18

December 14, 2025



Contents

1 Introduction

1.1 Competition Description

This report details the design, creation, and performance of an autonomous robot capable of driving along a track while identifying clues and obeying various traffic laws. The robot is programmed using the ROS framework and the environment is simulated through Gazebo. An image of the environment can be seen in 1.

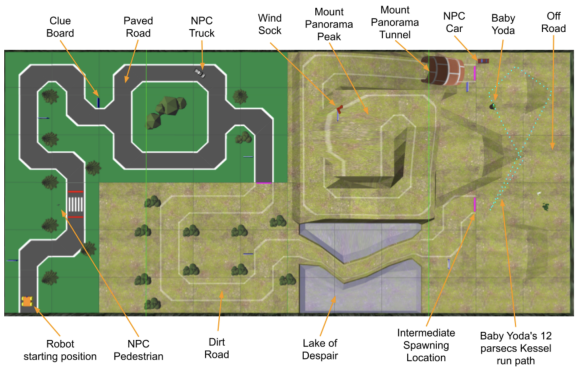


Figure 1: The Gazebo competition environment with the important components labeled.

The objective of the competition is for the robot to identify and read information on eight clue boards placed throughout the course and publish the messages to ROS topic that awards the robot points for correct reading. This is done while the robot autonomously drives through the course, avoiding collisions with several NPCs (non-playable characters). Points are deducted for driving off the road, teleporting the robot, and colliding with NPCs.

1.2 Contribution Split

Abi was responsible for the software allowing the robot to detect and read the clueboards, and Danny was responsible for the robot's driving and avoiding collisions with the NPCs.

1.3 Software Architecture

Two GitHub repositories were used for this project: enph353_controller_pkg, which contains our ROS nodes and robot interfaces, and enph353_cnn_trainer, which contains the Google Colab notebooks we used to create and train our models. It also contains the code used to generate test data for clueboard detection.

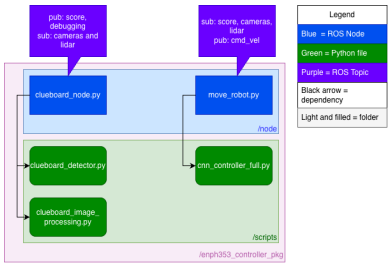


Figure 2: Diagram of our software architecture.

Figure 2 shows how our code worked together. Note this is a diagram of the parts of controller_pkg directly relevant to the control of the robot and reading boards. The files left out are those containing support functions for these nodes and the TFLite files containing the neural network models. We also used a Finite State Machine (FSM) to navigate the course, which is described in Figure 3.

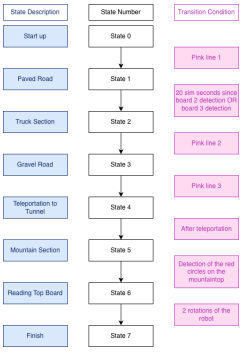


Figure 3: FSM Diagram

As shown, our FSM was used to transition between the Convolutional Neural Networks (CNNs) designed to drive the different sections of the course, trained via imitation learning. The condition for changing the states is to detect the pink lines that delineate the boundaries of these sections. A visualization of the pink line detection can be seen in 4. The only exception to this is the transition to the truck model, which occurs when the third clueboard is detected (with contingencies for the cases in which that board is not sensed using a timer after the second board is detected). The others are based on the teleportation and the robot finding the windsock at the top of the mountain.

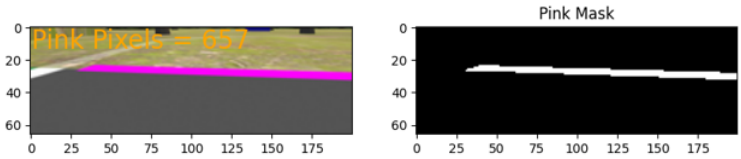


Figure 4: Pink Pixel Mask

2 Discussion

2.1 Driving Module

2.1.1 Neural Network Information

Four neural networks were used for driving, with a model for each section of the course (paved, truck, gravel, and mountain). All of these models had the same structure, only differing in the section of the course on which they were trained. As such, their information is presented together.

2.1.1.1 Data Engine and Data Acquisition

The models were built and trained in Google Colab.

They were trained on (image, twist command) pairs obtained using a data collection script. A custom keyboard teleop controller was designed that makes the robot move forward at a constant speed, and turns by moving the mouse left and right on screen. While the user is driving, the collector script obtains frames from the camera at a rate of 10Hz as well as the current angular z value from the latest twist command. Then, images are saved with their corresponding angular z command appended to the end of their file name, creating a mapping of driving frames to twist commands.

The actual data collection consists of recording multiple runs of the region corresponding to the current model being trained. The data is then enriched with examples of tricky sections, sharp turns, and regaining the road after going off course.

2.1.1.2 Data Pre-Processing

The input data size is reduced by saving only the bottom half of the camera frame and then downsizing to a final resolution of 200 by 66. An example of the downsized images can be seen in figure 5.

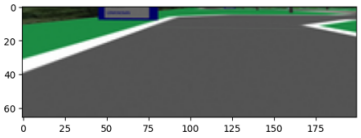


Figure 5: Sample CNN Input Image

The data is uploaded to a Google Colab script that assigns data labels, splits the data, and oversamples where necessary. The data is also normalized by dividing the pixel values by 255, such that each pixel is between 0 and 1. Both the pixel values and the angular velocities are cast to float 32, as it is less computationally intensive than float 64.

2.1.1.3 CNN Architecture

The driving network architecture was based on NVIDIA's PilotNet architecture, designed as an end-to-end steering model for self-driving cars. The final output is linear to allow for positive and negative steering angles on a range of outputs greater than -1 to 1. The model summary can be seen in figure 6 As seen in the model summary, the models had 1,595,511 trainable parameters and 0 non-trainable parameters.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 31, 98, 24)	1,824
conv2d_1 (Conv2D)	(None, 14, 47, 36)	21,636
conv2d_2 (Conv2D)	(None, 5, 22, 48)	43,248
conv2d_3 (Conv2D)	(None, 3, 20, 64)	27,712
conv2d_4 (Conv2D)	(None, 1, 18, 64)	36,928
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 1164)	1,342,092
dense_1 (Dense)	(None, 100)	116,500
dense_2 (Dense)	(None, 50)	5,050
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11

Total params: 1,595,511 (6.09 MB)
Trainable params: 1,595,511 (6.09 MB)
Non-trainable params: 0 (0.00 B)

Figure 6: Driving Model Structure

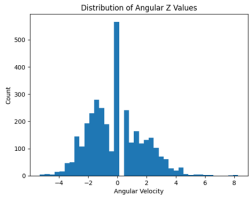
2.1.1.4 Training Parameters

The data was trained for between 25-50 epochs with the learning rate decided by the Adam optimizer.

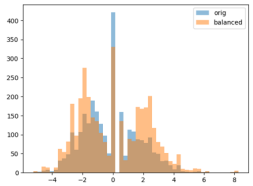
The data was split into a validation and training set, with 80% training and 20% validation. A test set was neglected, since the validation examples could just be examined to

determine model behaviour. Any test data would be too similar to the validation data to be useful.

The training set was then split into three subsets based on the angular velocity: large positive values, large negative values, and values around zero. The large turning value subsets were oversampled to have the same set size as the middle subset. This was done to prevent from producing trivial models that only cause the robot to drive straight. The oversampling is visualized in figures 7a and 7b. Figure 7b also serves as a histogram of the training data.



(a) Original Angular Z Distribution Histogram



(b) Angular Z Balanced Distribution Histogram

Figure 7: Paved Model Training Data Histogram

Training was done based on the mean squared error (MSE), and early stopping of training was based on the mean average error (MAE) of the validation set. MSE was used for training because it discourages overturning. This is because big spikes are penalized heavily due to error squaring. MAE was used for the stopping condition since it does not lead to big error spikes due to under-predicting sharp turns. Since there are few sharp turns on the track, it is better to have the model drive with smaller, less aggressive turns. Early stopping was implemented with a patience of 10 iterations (with a minimum validation MAE improvement of 0.001) for early runs, and fine tuning was done with a patience of 5.

2.1.1.5 Training and Validation Record

The MSE and MAE of each of the models during training can be found in Appendix B (Note that the paved model graphs display fine-tuning after initial training).

Numerous tools were used for error analysis and were applied to the validation set. The following explain the type of plot, and the data analysis insights they provide. Note that the figures shown are all for the paved model as a good baseline example.

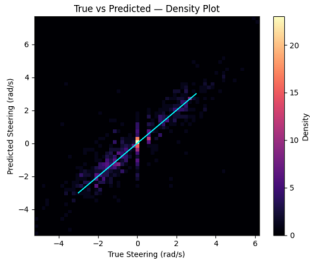
- Model Predictions Density Plot, figure 8a: Used to compare the true values vs. predicted values of the model. The teal line represents the "ideal" performance.
- Residual Plot, figure 8b: Used to determine whether the model is oversteering or understeering for different turning values. The residual is the true value subtracted

from the predicted value. A positive residual indicates oversteering, and negative residual corresponds to understeering. Ideal behaviour is an oval cloud centered around zero residual and spread along negative and positive angular predictions.

- Probability Distribution Overlap, figure 8c: This visualizes the probability density of the predicted vs real values. Here, it displays an overlap of the steering value distribution, which makes it easy to pick out regions the model struggles with. Ideal behavior is a near complete overlap of the distributions.

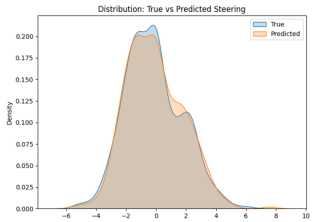
- Error v. Steering Magnitude, figure 8d: This plot compares the magnitude of error vs the steering magnitude. This displays whether the model is making more mistakes for bigger turns or smaller turns. A good model will have an even distribution across the state space.

- True vs. Predicted Scatter Plot, figure 8e: Similar to the true vs. predicted density plot, but shows a raw scatter plot instead, which makes it easier to pick out specific problem regions. A closely packed random distribution around the y = x line is the ideal behaviour.

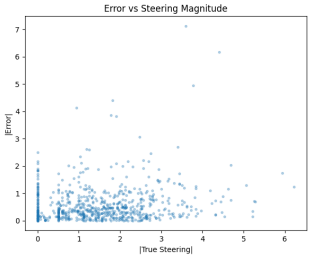


(a) True vs. Prediction Density Plot

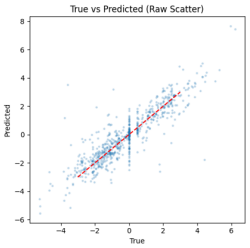
(b) Residual Plot



(c) True vs. Predicted Distribution Overlap



(d) Error vs. Steering Magnitude Plot



(e) True vs. Predicted Scatter Plot

Figure 8: Model prediction performance visualizations

These graphs were mainly used to validate whether training went well or poorly. Once the plots showed reliable predictions, the 10 worst model predictions were displayed. Going through these provided insight into where the model was struggling. In the best performance cases, the incorrect predictions would actually be better than the ground truth/label (indicating poor driving during data collection). In problem regions where the model was under-predicting or incorrectly predicting turns, the data was enriched with more examples from that region, and the model got retrained.

2.1.2 Driving Controller

To implement the driving controller, the models trained in Colab were saved as TFLite files. A separate CNN controller was made for each section, then they were combined into one for the final competition controller. The controller scripts initialize a local model and use it in tandem with an image subscriber to take in images and publish twist commands based on the model predictions. Switching between the models was done using the FSM discussed above.

2.1.3 Pedestrian, Truck, and Baby Yoda

2.1.3.1 Pedestrian

The pedestrian avoidance relies on sensing the red line indicating the start of the sidewalk and then waiting until the pedestrian has passed in front of the robot. The LiDAR used is separate from the one for clueboard detection, and is set up so it is low enough to detect the NPC hitboxes. The pedestrian is identified by the number of consecutive rays reading below 0.5m in the central 80°.

2.1.3.2 Truck

The truck detection is similar to the pedestrian detection. Once the robot is in state 2 (truck section), the LiDAR scan is continuously checked for $3 - 15$ consecutive rays in the central 80° reading below 0.3m. A common issue in this section was the LiDAR detecting clueboards as trucks, which required fine tuning of the ranges and engagement distance. A small turn to the left is added while the robot waits for the LiDAR, to encourage it to take the left turns through the truck section (preventing the robot and truck driving in opposing directions).

2.1.3.3 Baby Yoda

There was sadly not time to implement the Baby Yoda section. Instead, the robot teleports through the off road section. For more information, see Section 3.2.2.

2.1.4 Edge Case Performance Analysis

Overall, the robot performed fairly well throughout the various sections. It behaved very consistently outside of the edge cases that follow:

1. Truck: the biggest issue the robot would face was crashing into the third clueboard. Despite both the paved and truck models both being trained around this clueboard, neither handled it well

2. Mountain: Sometimes in the last uphill section, the robot would choose to leave the line and scale the side of the mountain. However, it still reaches the peak successfully.

2.2 Clueboard Recognition Module

2.2.1 General Approach

A neural network was used to read the plates by identifying the characters individually. To identify when to process and image to find a plate, a LiDAR was used to locate clueboards by looking for sections in which the distance linearly decreases. These candidates were also filtered by the number of dark blue pixels (the colour being tuned to only count boards). Then, the code finds the homography of the clueboard by isolating the contour representing the board's border, and from there isolates and identifies the characters by feeding the images to the CNN.

2.2.2 Neural Network Information

2.2.2.1 Data Engine and Data Acquisition

A python script (which can be found at clueboard_generation/generate_perfect_clueboards.py) was used to generate 500 clueboard images based on the code provided in the 2025_competition repository. All the further processing and CNN training was done in Google Colab.

2.2.2.2 Data Pre-Processing

Once the perfect clueboards were imported into Google Colab, ImageDataGenerator was used to generate versions of those images with different degrees of distortion, lighting, and zoom. Other functions were also used to add gaussian and motion blur, in addition to grain and translation. Initially, only one flawed clueboard would be generated for each perfect one, but as time went on the training data was enriched with more flawed versions of boards containing characters the CNN had issues identifying.

2.2.2.3 CNN Architecture

Figure 9 is the output of the model definition from Colab. This structure is based on the CNN from Lab 6, used for license plate recognition. That is because they are both designed for recognizing characters and as such it made sense to keep the structure similar.

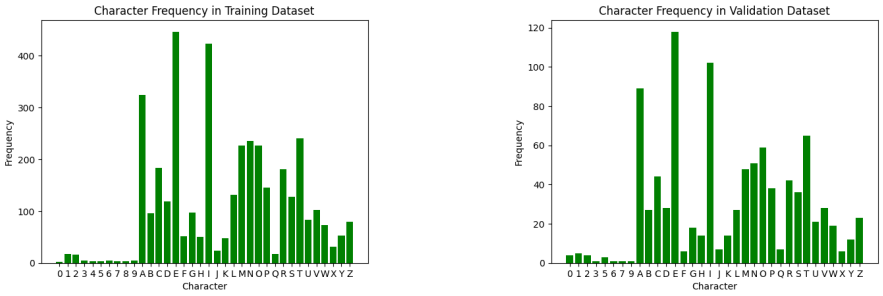
In total, the CNN has 8,686,020 trainable parameters and 896 non-trainable parameters.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 128, 128, 32)	896
batch_normalization (BatchNormalization)	(None, 128, 128, 32)	128
activation (Activation)	(None, 128, 128, 32)	0
conv2d_1 (Conv2D)	(None, 128, 128, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 128, 128, 32)	128
activation_1 (Activation)	(None, 128, 128, 32)	0
max_pooling2d (MaxPooling2D)	(None, 64, 64, 32)	0
dropout (Dropout)	(None, 64, 64, 32)	0
conv2d_2 (Conv2D)	(None, 64, 64, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 64, 64, 64)	256
activation_2 (Activation)	(None, 64, 64, 64)	0
conv2d_3 (Conv2D)	(None, 64, 64, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 64, 64, 64)	256
activation_3 (Activation)	(None, 64, 64, 64)	0
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 64)	0
dropout_1 (Dropout)	(None, 32, 32, 64)	0
conv2d_4 (Conv2D)	(None, 32, 32, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 32, 32, 128)	512
activation_4 (Activation)	(None, 32, 32, 128)	0
conv2d_5 (Conv2D)	(None, 32, 32, 128)	147,584
batch_normalization_5 (BatchNormalization)	(None, 32, 32, 128)	512
activation_5 (Activation)	(None, 32, 32, 128)	0
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 128)	0
dropout_2 (Dropout)	(None, 16, 16, 128)	0
flatten (Flatten)	(None, 32768)	0
dense (Dense)	(None, 256)	8,388,864
dropout_3 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 36)	9,252

Figure 9: CNN Structure (Keras Summary)

2.2.2.4 Training Parameters

I used 30 epochs and a learning rate of 1.20e-4. My training set had 3878 characters, and the validation and test sets had 969 and 1211 characters respectively. Figures 10a and 10b show the amount of training examples for each letter. Note the test set is left out as the relative frequencies with which the letters appear can be read from the confusion matrix.

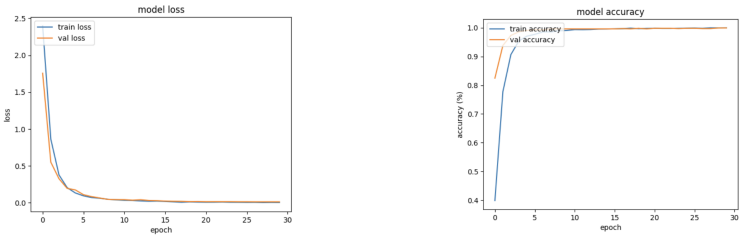


(a) Training Data (b) Validation Data

Figure 10: Histogram of Character Frequency in Training and Validation Data

2.2.2.5 Training and Validation Record

Figures 11a and 11b are the graphs detailing the model accuracy and loss against epoch, and figure 12 is the confusion matrix.



(a) Model Loss (b) Model Accuracy

Figure 11: Model loss and accuracy

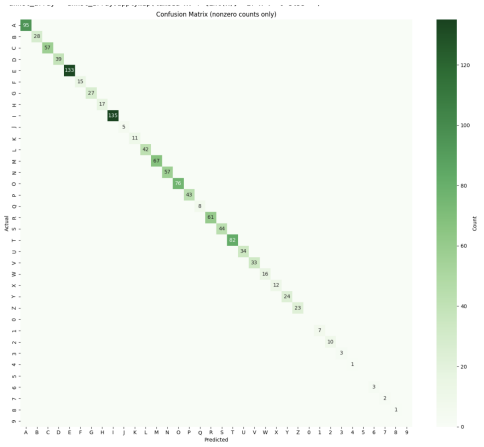


Figure 12: Confusion Matrix

3 Conclusion

3.1 Competition Performance

Our performance in the competition was decent, but sadly was not as good as our runs while practicing. We scored 18 points in 240 seconds. This is because the robot got stuck at the third clueboard. However, in the best runs, the robot manages to make it all the way to the end and collect 36 points (accounting for the deduction due to teleporting) in 124 seconds.

3.2 Methods Attempted and Discarded

3.2.1 SIFT for Clueboard Detection

The main method attempted which did not pan out was using SIFT to detect the clueboards by searching for the image of the fizz detective. This was fully implemented using multi-threading so no potential boards would be missed while one was being processed. However, this proved futile as SIFT is very computationally expensive, and running it would take several seconds to process a single clueboard candidate image. As such, it had to be abandoned and the clueboards were found and isolated using pure image processing, which was much more efficient and easier to integrate with the driving.

3.2.2 Baby Yoda and Mountain Following

An attempt was made to train a CNN that follows the contour of the mountain for the offroad section. It quickly became evident that this strategy would not work for the following reasons: First, it is easy to lose the mountain. Second, the offroad section has many bumps which are confused for the mountainside. Third, when performing properly, the model would leave the robot too far from the tunnel. The best alternative solutions considered were:

1. Counting the number of green pixels near the center of the image, and PID on keeping the green pixels centered.

2. Using LiDAR to detect Yoda, and use PID to keep yoda centered on screen.

3. Using edge detection to find the direction Yoda is moving, and use PID to minimize his relative motion to the robot.

All of these solutions would also require some additional strategy to recognize and move toward the tunnel. We planned on doing this by counting red pixels on screen (to determine proximity to the red car by the tunnel).

3.3 Reflection

The switch from SIFT happened quite late in the competition, so there was not as much time to optimize the software after integrating as would have been ideal. Given more time, we could have invested in increasing the robot's robustness as a whole and implementing one of the ideas discussed in section 3.2.2.

4 References

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016, April 1). *End to End Learning for Self-Driving Cars*. NASA ADS. https://doi.org/10.48550/arXiv.1604.07316

LearnOpenCV. (2023, March 27). Mastering deep learning: Implementing a convolutional neural network from scratch with Keras [Video]. YouTube. https://www.youtube.com/watch?v=EJQaIWT3mRE

5 Appendix A: Important LLM Conversations

For brevity, this section contains the most important CNN conversation for each of the modules. All others can be found in our respective logbooks.

5.1 Driving Model

The most useful conversation with ChatGPT regarding the driving model, was asking to chat to help find a suitable driving model architecture. Originally, I had attempted to repurpose a classifier model, by simply modifying the input and output layers. This worked quite poorly. Upon consultation with ChatGPT, I was recommended numerous driving models, and eventually settled on the NVIDIA End-to-End Steering architecture. The full transcript from my conversation with ChatGPT can be found on pg. 280 - 284 of ENPH 353 Logbook - Danny.

The model chosen was based on one developed in a paper published by NVIDIA in 2016. Their proposed neural network architecture is shown in 13:

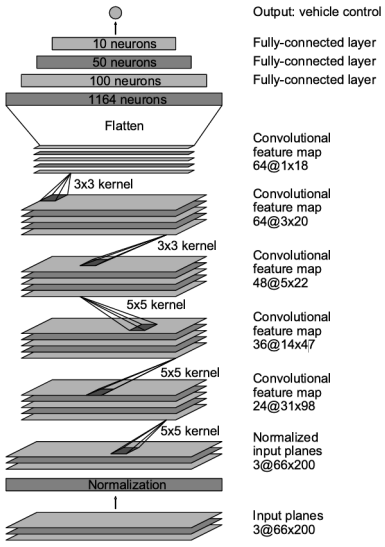


Figure 13: NVIDIA End to End Steering Model Architecture (Bojarski et al., 2016)

5.2 Data Augmentation for Clueboard Reading

The most useful conversation I had with an LLM was when I was having issues getting the CNN to recognise characters in the actual envrionment after moving away from SIFT. The full conversation can be found here: ChatGPT Conversation on Data Augmentation.

The salient point was helping me move away from extreme distortions to ones that more accurately reflected my data, and consisted of implementing functions that allowed me to even more precisely control the following augmentations:

1. Gaussian blur
2. Motion blur
3. Graininess
4. Downsampling
5. Translation of the clueboard

To see the difference, figure 14 is an example of a typical character used in training before and after the change in augmentation.

(a) Bad Augmentation

(b) Good Augmentation

Figure 14: Good and Bad Data Augmentation Examples

Training on the new data made the CNN much more accurate (as can be seen above) and restored the performance it had before switching the detection method.

6 Appendix B: MSE and MAE Diagrams for Driving Models

(a) Paved MSE (b) Paved MAE (c) Truck MSE (d) Truck MAE

(e) Gravel MSE (f) Gravel MAE (g) Mountain MSE (h) Mountain MAE

Figure 15: Training MSE and MAE Graphs for Paved, Truck, Gravel, and Mountain Models