ENPH 353 Final Report

Benedikt Howard & Shuyu van Kerkwijk

University of British Columbia

April 12, 2025

Contents

1. Background

The ENPH 353 competition simulates an environment in Gazebo where a robot must navigate through a course, detect and read signs containing clue messages, and avoid obstacles like a pedestrian, truck, and Baby Yoda. Our project fulfills these goals through a modular software stack and robustly trained convolutional neural networks (CNNs).

Figure 1: Competition World in Gazebo.

2. Team Contributions

Benedikt Howard

- Designed the software architecture.
- Built the data engine for driving.
- Trained four CNNs for driving.
- Developed pedestrian detection.
- Collected driving training data.

Shuyu van Kerkwijk

- Built the pipeline for sign reading.
- Built the data engine for sign reading.
- Trained CNN for character recognition.
- Integrated sign reading into the project.
- Collected driving training data.

3. Software Architecture

We built our system on the Robot Operating System (ROS1) and followed a modular node-based architecture designed for readability, fault isolation, and real-time feedback. Each core function of the robot was encapsulated in a dedicated ROS node. Figure 2 shows an overview of the full software stack.

3.1 Core System Nodes

- **Control Node**: Acts as the primary state coordinator. It manages transitions between manual and autonomous driving modes, handles keyboard inputs, and tracks section timeouts. It ensures the robot defaults to safe behavior in uncertain situations.
- **Controller Node**: Reads controller input (joystick and button) from a serial connection and publishes corresponding velocity commands. It also toggles the record mode used during training data collection.

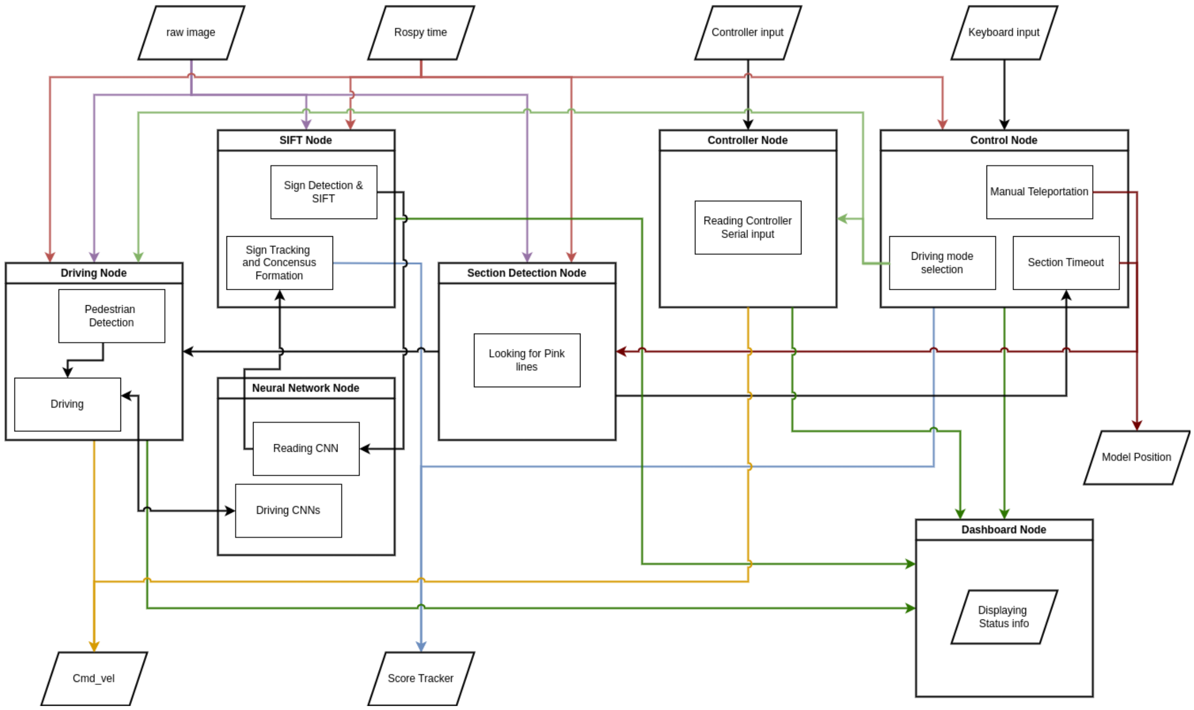


Figure 2: Overview of ROS nodes and data flow in our competition software architecture.

- **Driving Node**: Executes driving logic when in autonomous driving mode. It subscribes to a camera feed, implements pedestrian detection using motion thresholding, and calls Neural Network Node for driving services based on what course section the robot is in.
- **Section Detection Node**: Analyzes the incoming frames to detect pink lines at the base of the image. These pink markers separate different sections of the track (Road, Gravel, Ramp, Off-Road). It applies a timeout mechanism to provide robust section transitions.
- **Neural Network Node**: Loads and runs section-specific CNN models and exposes services for the driving node to request velocity command predictions from input images. It also communicates with the Scale-Invariant-Feature-Transform (SIFT) node to identify characters with a CNN. This node exists because loading TensorFlow runtimes in multiple nodes led to memory issues.
- **SIFT Node**: Executes logic for detecting and reading signs. It uses the SIFT algorithm and communicates with the Neural Network Node to identify characters on signs. It uses a consensus mechanism to identify and confirm clues and publishes this output to the Score Tracker.
- **Dashboard Node**: Provides live telemetry and state feedback to the operator. It receives inputs from several processes and renders them in a real-time GUI.

3.2 Data Flow and Design Rationale

The system was designed with a clear separation of concerns to allow independent development and debugging. All messages passing between nodes were managed using ROS topics, with the exception of images for the driving prediction model, which were handled by services.

Table 1: Summary of ROS topics and associated nodes

Topic	Type	Description	Publisher(s)	Subscriber(s)
/cmd_vel	Twist	Velocity command	Control Node, Driving Node, Controller Node	Gazebo

Topic	Type	Description	Publisher(s)	Subscriber(s)
/track_section	String	Current track section	Section detection Node	Driving Node, Control Node
/*/status	String	Publishes the state of the node	*	Dashboard Node
/image_raw	Image	The camera feed from the robot	Gazebo	Driving Node, SIFT Node
/auto	Bool	Automatic driving activation	Control Node	Driving Node
/teleop	Bool	Teleoperation activation	Control Node	Controller Node
/reset	String	Communicates Control node forced state changes	Control Node	Driving Node, Section Detection Node, SIFT Node
/read_input_images	ImageWithID[1]	Images of letters to be read	SIFT Node	Neural Network Node
/read_image_result	String	Read letters returned with ID and prediction	Neural Network Node	SIFT Node
/B1/{track section}_service	Image → String (Service)	Evaluates the driving output for images	Neural Network Node	SIFT Node
/score_tracker	String	Outputs score-able events and information	SIFT Node, Control Node	Score Tracker

3.3 Version Control and Collaboration

The full system was versioned in a private GitHub repository[2]. Due to hardware limitations on Shuyu's computer, which could not run the full application stack, we adopted a slightly unconventional workflow. Shuyu developed and unit-tested her components locally, then shared updates with Benedikt via email. She then integrated the code into the full system, performed system-level testing and validation, and pushed the changes to the repository. This process ensured reliable integration while accommodating our hardware constraints.

4. Discussion

4.1 Driving Controller

Overview

We used an imitation learning approach for our driving controller, inspired by NVIDIA's end-to-end self-driving research [[1]]. This choice was made early in the project after considering alternatives like proportional-integral-derivative (PID) controllers and reinforcement learning. While PID offered proven reliability, we prioritized gaining hands-on machine learning experience. Reinforcement learning was considered too impractical for the competition context.

A model was trained for each section (Road, Gravel, Off-Road, Ramp). All models used a CNN with a 341×316 input resolution and produced two outputs − linear and angular velocity. We used a standard `tanh` activation function to match the output range of the two velocities.

For deployment, the camera feed was used as input to the model for the section the robot was driving in, and the model's predictions were published as command velocities. In some cases, the model's predictions were multiplied by speed-up factors to increase driving speed. Additional logic was layered on top to handle obstacles safely (Figure 5).

Section Transition

The robot's ability to handle transitions between different track segments was essential, as each segment demanded unique driving behaviour.

To detect section changes, we implemented a lightweight `section_detector` node. This node performed color thresholding in HSV space to isolate the pink boundary lines demarcating section transitions (Figure 3). By masking the lower portion of the image and computing the contour area of the detected pink regions, we were able to reliably identify the presence of a boundary line. A timeout condition was applied to prevent multiple transitions from being triggered if the boundary line was in the field of view for a sustained period.

[1] `ImageWithID` is a custom datatype that wraps an Image and a String.
[2] `https://github.com/shuyuvankerkwijk/enph353_competition`

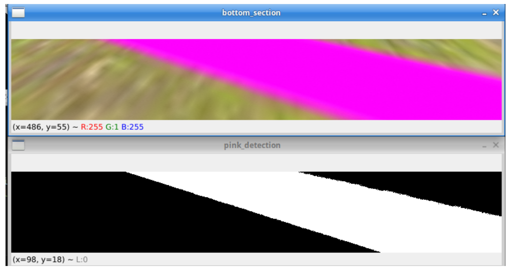


Figure 3: Example of section detector thresholding

Pedestrian Avoidance

The pedestrian exhibited the following predictable behaviour: it would remain stationary, slowly rotate, then suddenly cross the road.

For pedestrian detection and avoidance, we used a two-part strategy. First, the robot identified the start of the crosswalk by thresholding for its red boundary line (Figure 1). Upon detection, it stopped and activated a custom motion detector to monitor for pedestrian movement, proceeding only once the pedestrian had crossed. This approach was chosen over integrating detection into the driving model due to the complexity of the pedestrian's motion.

To detect motion, we performed frame differencing between consecutive camera frames. Weighted color filtering was applied to suppress noise from green-spectrum foliage present in the background. We also cropped to a narrow horizontal slice of the image corresponding to the pedestrian's expected location – which excluded unrelated motion from objects like the truck (Figure 4).

The robot waited at the red line until it detected the pedestrian moving and then coming to a complete stop, at which point it drove quickly through the crosswalk. For robustness, we implemented a rolling buffer: the robot only resumed once all frames in the buffer confirmed the absence of motion. This mechanism proved effective, introduced minimal latency, and allowed the robot to consistently navigate the crosswalk safely.

Figure 4: Frame Differencing Output for Detecting Pedestrian Crossing

Truck and Yoda Avoidance

We aimed for our imitation learning models to handle as much of the course as possible autonomously, including truck and Baby Yoda avoidance – viewing it as an exciting challenge. However, both the truck and Baby Yoda could be encountered at different points in their travel paths, meaning the driving models would have to learn very complex patterns to navigate these obstacles.

To overcome this problem, our training approach was guided by strict behavioral rules designed to shape the CNN's learned responses. For the truck, we would wait if it was in the quarter-loop before the intersection, but proceed normally otherwise. For Baby Yoda, we would wait until it fully entered the field of view and turned away before following. These behaviors were demonstrated consistently during data collection.

During testing, we monitored the CNN outputs and observed encouraging behavior: the robot would slow or stop when encountering the truck or Baby Yoda. We reinforced this behavior by recording more training data until the model reliably handled all edge cases. We also added a small safety layer: if the predicted velocity fell below a threshold, we clamped it to zero. This prevented the robot from creeping forward and misjudging a scene as safe to drive.

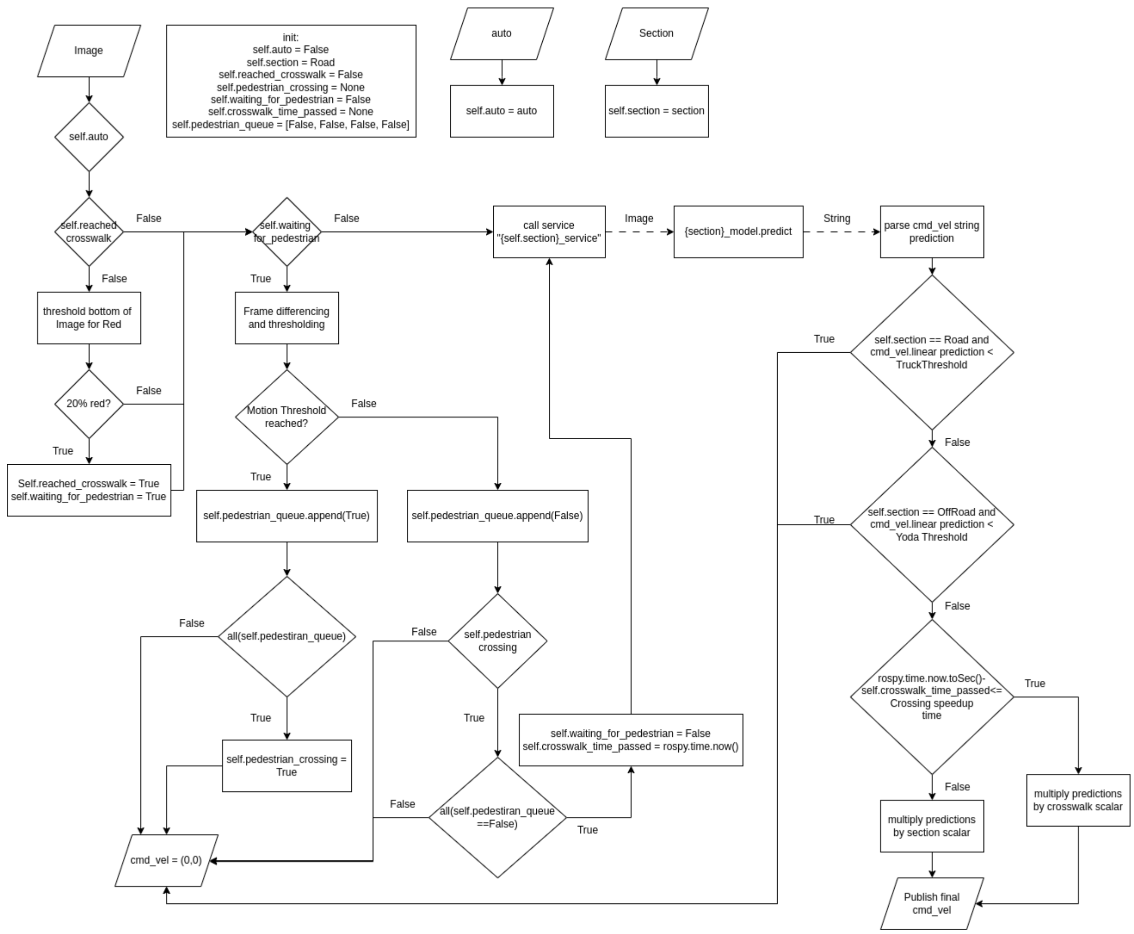


Figure 5: Overview of Driving Controller Logic

Data Engine

To implement imitation learning, we built a custom teleoperation system using a joystick for driving and a physical pushbutton to toggle recording. The joystick's analog signals were processed by an ESP32, which applied a log-to-linear transformation to compensate for the nonlinearity of the joystick. These values were transmitted over Serial and published as command velocities at a frequency of 20Hz.

When recording was active, images of the camera feed were resized, saved to disk, and labeled via timestamped filenames that encoded the linear and angular velocities as well as the current section of the course. A real-time debug overlay on the camera feed displaying telemetry such as recording status and section helped us maintain high-quality training data and discard any mislabeled or unusable frames.

To improve data quality, we prevented the recording of undesirable driving commands – such as moving backward (i.e., negative linear velocity). We also manually reviewed portions of the training data to ensure that the captured data reflected our intended driving behavior.

Data Preprocessing

The training data set – over 100,000 labeled images – were organized by section, enabling us to train separate models. Images were parsed using regex-based labeling. The distribution and relation between velocity commands in our training data for the Road section is shown in Figure 6. See 14 for distributions of the other sections.

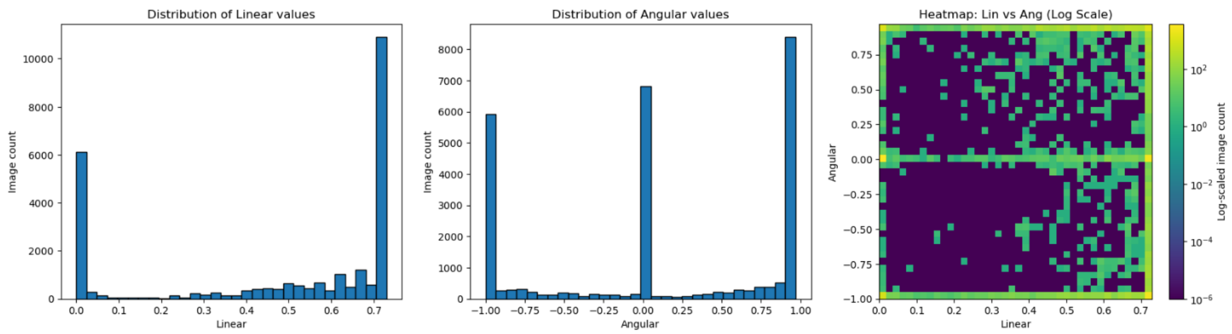


Figure 6: Training Data Command Velocity Distributions for Road Section

Neural Network Architecture

The CNN architecture and parameters used for all sections are shown below. Training accuracy for both the initial training (Figure 7a) and fine-tuning (Figure 7b) are shown.

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 rescaling (Rescaling)        (None, 316, 384, 3)       0
 conv2d (Conv2D)              (None, 316, 384, 64)      4864
 batch_normalization          (None, 316, 384, 64)      256
 max_pooling2d                (None, 158, 192, 64)      0
 dropout (Dropout)            (None, 158, 192, 64)      0
 conv2d_1 (Conv2D)            (None, 158, 192, 128)     73856
 batch_normalization_1        (None, 158, 192, 128)     512
 max_pooling2d_1              (None, 79, 96, 128)       0
 dropout_1 (Dropout)          (None, 79, 96, 128)       0
 conv2d_2 (Conv2D)            (None, 79, 96, 256)       295168
 batch_normalization_2        (None, 79, 96, 256)       1024
 global_average_pooling2d     (None, 256)               0
 dropout_2 (Dropout)          (None, 256)               0
 dense (Dense)                (None, 512)               131584
 dense_1 (Dense)              (None, 256)               131328
 dense_2 (Dense)              (None, 2)                 514
=================================================================
Total params: 639,106
Trainable params: 638,210
Non-trainable params: 896
```

Training Details

Initial Training:

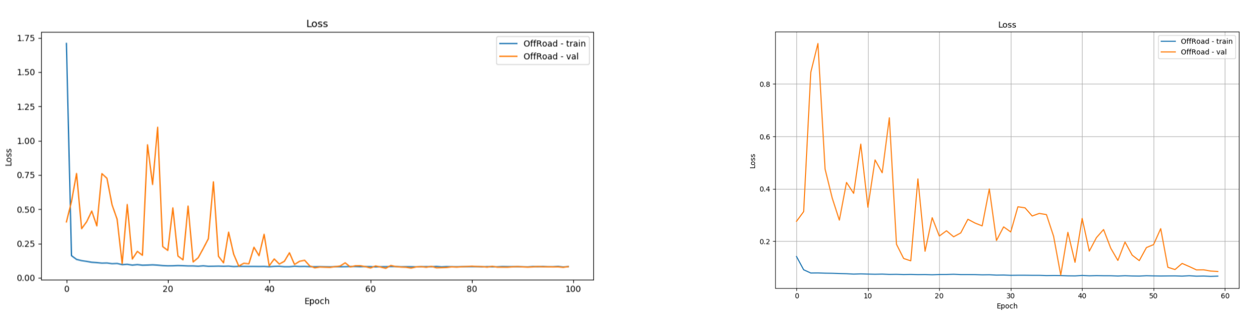
- Optimizer: Adam
- Learning Rate: 0.0005
- Epochs: 100
- Training Set Size: 10,000 images
- Mini Batch Size: 20
- Validation Split: 20%

Fine-tuning:

- Learning Rate: 0.0001
- Epochs: 50
- Training Set Size: 25,000+ images

Callbacks:

```
tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=50, min_delta=1e-5, mode='min',
    restore_best_weights=True),
tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=7, verbose=1),
tf.keras.callbacks.ModelCheckpoint('/home/fizzer/ros_ws/training_for_driving/'+Section+'/
    best_model.h5', save_best_only=True, save_weights_only=False, monitor='val_loss')
```



(a) Initial training loss

(b) Fine-tuning training loss

Figure 7: Training loss during initial training and fine-tuning phases.

Driving Controller Performance

Overall, the driving controller delivered stable and reliable performance across all course segments. The CNNs consistently produced smooth, human-like trajectories with minimal jitter, even in regions with limited features.

Edge cases (e.g. transitions between sections or near-misses with the truck) were handled safely and predictably. The models demonstrated strong generalization: we observed that the robot could recover from previously unseen orientations or perturbations in position, a result of the model having been trained on diverse and intentionally imperfect demonstrations. The system was particularly resilient in challenging conditions such as the narrow ramp entry and sharp turns (where off-track behavior was common in early tests). In these situations, the robot corrected itself smoothly, typically by reducing speed or subtly adjusting steering. Additionally, the layered safety logic ensured that incorrect velocity predictions that hovered near decision boundaries did not lead to unsafe actions. Across more than 20 consecutive runs leading up to the competition, we observed no critical failures and consistent completion of the full course.

4.2 Sign Detection and Reading

Overview

We used a SIFT-based pipeline to locate and warp signs (Figure 9a). Individual letters were extracted and cropped using classical computer vision techniques. Finally, a CNN was used to classify letters and decipher the clue on the signboard. A detailed overview of the sign reading process is shown in Figure 8.

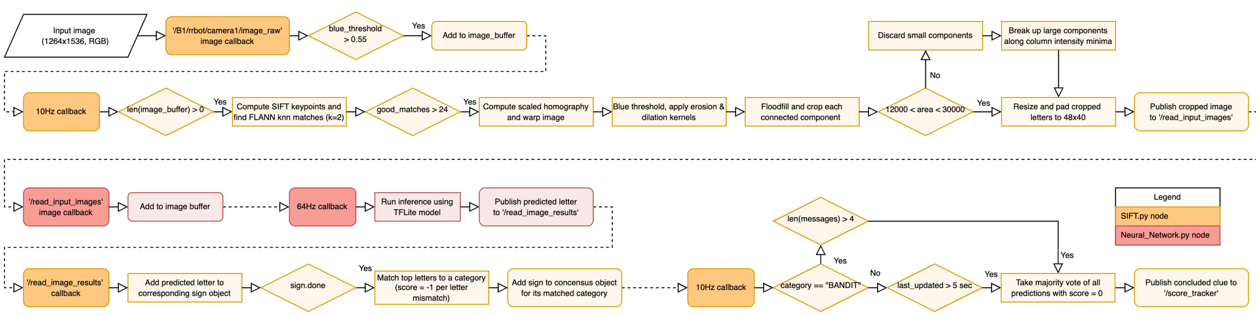


Figure 8: Overview of Sign Reading Logic

Identifying and warping signs

A lightweight blue filter mask was applied through HSV thresholding to all camera frames to detect whether a sign was present and nearby. If enough pixels fell within our defined "blue" range, the image was added to a processing queue for further analysis. To robustly localize the sign, we used SIFT and Fast Library for Approximate Nearest Neighbors (FLANN) to find and match keypoints and descriptors between a camera image and a template image of a blank sign. If a sufficient number of good matches were found, a homography matrix was computed, and the camera image was warped and cropped to extract only the sign. This alignment corrected for perspective and scale variations, allowing for consistent downstream processing.

Extracting and predicting letters

A blue threshold was applied to the warped image to isolate the letters. We used morphological operations (dilation, erosion) followed by floodfill and connected components analysis to extract character regions [[2]]. Boxes that were abnormally large or small were discarded. Oversized boxes (2–3× expected area) were assumed to be merged characters. These were split using the minima of column-wise intensity projections. The end product was two sets of cropped character images — one for the top word and one for the bottom.

Each cropped character region was resized and padded to a standardized dimension of 40×48. We then published each image, along with an identifying ID string, as a custom `ImageWithID` message to the Neural Network Node, which returned an integer index (i.e. letter `A` through `Z` or digit `0` through `9`). By offloading letter recognition to the Neural Network Node via ROS messages, we avoided GPU memory conflicts and maintained a clear separation of function in our codebase.

Sign classification and consensus

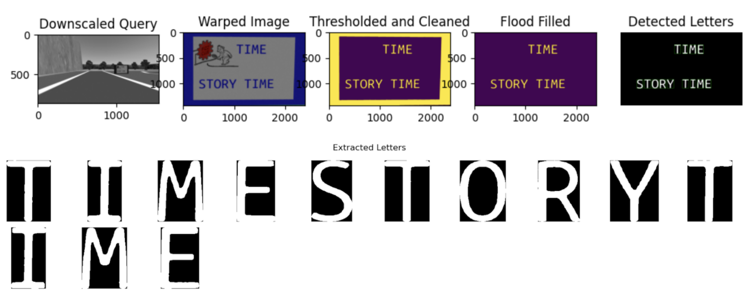
We tracked each sign with an internal `Sign` object, storing the individual character predictions until the top and bottom words were fully determined. Once a `Sign` was complete, the top word was mapped to the 8 known categories (i.e. `SIZE`, `VICTIM`, `CRIME`, etc.) and assigned a score based on its agreement (-1 per letter mismatch). All signs of the same category were aggregated in a `Concensus` object, which, after enough signs had been added, implemented logic to conclude on the sign clue using a majority vote system. The deciphered clue was published to the scoring logic.

Data Engine

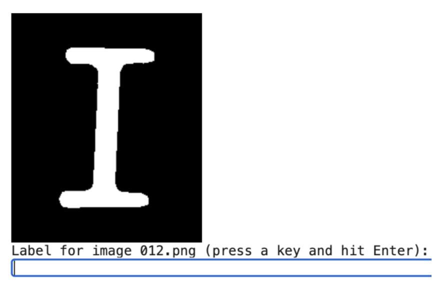
We captured video frames from the course using a front-facing camera in the same method as 4.1. Multiple drive-bys of each sign were recorded. Frames were saved and uploaded to Google Drive.

Data Preprocessing

For consistency, all images were preprocessed using the same pipeline as in the competition. Each recorded frame containing a sign was warped using SIFT and then processed and cropped into character boxes (Figure 9a). A custom Python script generated by ChatGPT 7.1 was used to rename each image file based on the character it contained (Figure 9b). Twenty images of each alphanumeric character were collected and labeled. To improve generalization, a Keras `ImageDataGenerator` was used to augment and supplement the training dataset. Four samples were created per labeled image, yielding 80 examples per character (Figure 11). Finally, all labels were one-hot encoded across 36 classes (26 letters, 10 numbers).



(a) Example of processed sign



(b) Custom labeling script

Figure 9: Sign preprocessing pipeline

ID	Clue	Value	Predicted	Score
1	SIZE	C5FA231W	C5FA231W	6
2	VICTIM	VUUG	VUUG	6
3	CRIME	YRF3FFDN2GF	YRF3FFDN2GF	6
4	TIME	4QO57FPT	4QO57FPT	6
5	PLACE	87DDX8W	87DDX8W	6
6	MOTIVE	U4B	U4B	6
7	WEAPON	F7T	F7T	8
8	BANDIT	SPAAJ	SPAAJ	8

Figure 10: Randomized clue set

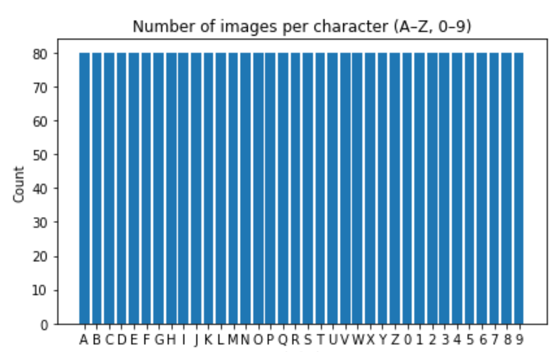


Figure 11: Histogram of training data

Neural Network Architecture

The CNN architecture and parameters are shown below. The model learned quickly (Figure 12) and performed very well on the test data set, achieving an accuracy of 1.00 (Figure 13). In order to speed up processing, the model was quantized and saved in the TFLite format. This reduced average inference time per character from 0.318 seconds to 0.0018 seconds and did not decrease accuracy.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 46, 38, 32)        320
 batch_normalization_3       (None, 46, 38, 32)        128
 max_pooling2d_3             (None, 23, 19, 32)        0
 conv2d_4 (Conv2D)           (None, 21, 17, 64)        18,496
 batch_normalization_4       (None, 21, 17, 64)        256
 max_pooling2d_4             (None, 10, 8, 64)         0
 conv2d_5 (Conv2D)           (None, 8, 6, 128)         73,856
 batch_normalization_5       (None, 8, 6, 128)         512
 max_pooling2d_5             (None, 4, 3, 128)         0
 flatten_1 (Flatten)         (None, 1536)              0
 dense_2 (Dense)             (None, 256)               393,472
 dropout_1 (Dropout)         (None, 256)               0
 dense_3 (Dense)             (None, 36)                9,252
=================================================================
Total params: 496,292
Trainable params: 495,844
Non-trainable params: 448
```

Training Details

- Optimizer: Adam
- Learning Rate: 0.0001
- Epochs: 80
- Training Set Size: 2015 images
- Validation Set Size: 432 images (15%)
- Test Set Size: 433 images (15%)
- Mini Batch Size: 16

Callbacks

```
tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=50, min_delta=0.00001, mode='min',
    restore_best_weights=True),
tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=7, verbose=1)
```

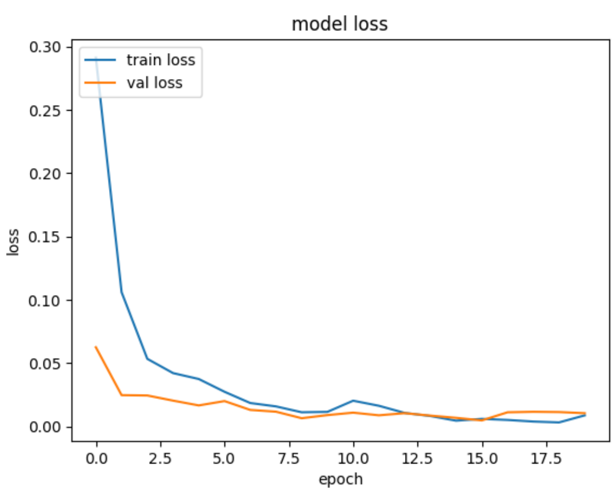


Figure 12: Accuracy vs. Epoch

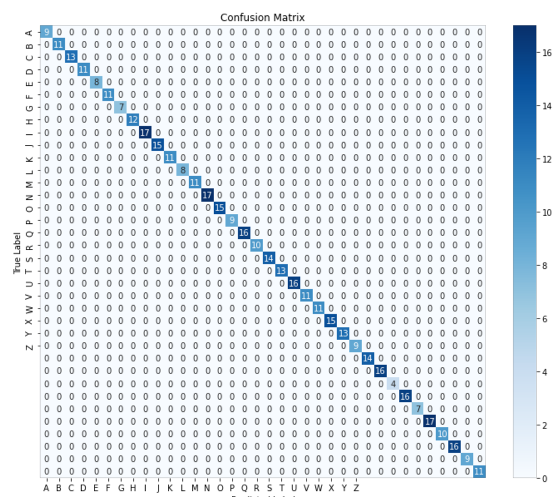


Figure 13: Confusion matrix on test dataset

Sign Reading Performance

The sign reading pipeline was remarkably stable during testing – when tested on randomly generated combinations of alphanumeric characters (see Figure 10) it did not mischaracterize a single sign in the more than 50 runs we tested.

5. Conclusion

5.1 Competition Performance

Our robot successfully completed all tasks during the competition. It accurately read every sign, avoided collisions with any obstacles, and finished the course in 87 seconds while running at a 0.25 real-time factor. Compared to other teams, our strategy of capturing multiple images per sign and forming a consensus from these readings proved highly effective.

5.2 Abandoned Methods

We initially used OpenCV Contour Detection for cropping the letters on clue plates. This was more computationally heavy and performed less reliably than flood-filling.

The original design of the CNN for driving had a split head design, one for the angular velocity and one for the linear. This led to training instabilities 7.1.

5.3 Future Improvements

While we performed well in the competition and achieved first place, there remain several areas we would have liked to improve had we had more time:

- **Refactor overall program structure:** Our software architecture could have been cleaner and more streamlined. For instance, instead of having separate ROS topics for teleoperation and autonomous driving modes, we could have implemented a unified state topic.
- **Optimize driving strategies:** Our driving was very stable, as many of our driving decisions were intentionally conservative to ensure reliability. However, more aggressive strategies would have significantly reduced our run time.
  - Driving through the crosswalk anytime the pedestrian is stationary, rather than waiting for it to complete a full crossing.
  - Driving in the opposite direction to the truck after it had passed to avoid limiting ourselves to the truck speed.
  - Taking a shortcut through the Off-Road section by turning left and driving over the hill straight to the tunnel. This would have dramatically reduced our course time, as we spent a significant amount of time waiting for Baby Yoda.
- **Driving model size:** Quantizing and pruning the driving models could improve inference speed and move us closer to real-time performance.
- **Sign reading optimization:** SIFT is very computationally intensive. We currently apply it to images scaled to 0.4×4K resolution. Further lowering the resolution of the input image could potentially make sign reading more efficient without reducing reliability.
- **Advanced frame filtering:** We currently use a simple blue pixel threshold to select frames for the sign reading pipeline, which results in many false positives, i.e frames processed by SIFT where no sign is visible. A more sophisticated method (e.g. applying a horizontal convolution filter to identify structured blue regions) could reduce false positives, limiting unnecessary computation.

6. References

[1] M. Bojarski et al., *End to End Learning for Self-Driving Cars*, arXiv preprint arXiv:1604.07316, 2016. https://arxiv.org/abs/1604.07316

[2] user16176940, *Cropping out multiple regions around letters with Python OpenCV*, Stack Overflow, 2020. https://stackoverflow.com/questions/65471444/cropping-out-multiple-regions-around-letters-with-pyth

[3] OpenAI, *ChatGPT*, 2024. Accessed via https://chat.openai.com/

7. Appendices

7.1 Appendix A: ChatGPT Conversations

Labeling character data

```
You:

i have a folder full of 200x240 png images of letters, but they are not labelled. can you please
    write me a script that will pull up one image at a time, wait for one keystroke (the letter to
     label the image as), and then rename the image with the corresponding name. ideally this
    would be in collab so it can directly access the same google drive but if not possible it can
    be in vs code

ChatGPT:

Sure! Here's a script that does exactly that: it opens each image in a folder one by one, waits
    for a keypress (the label), and then renames the image file to match the label you typed. You
    can run this in **VS Code**, or  I ll  help you adapt it for **Google Colab** with Google
    Drive integration afterward.

---

 **Python Script for VS Code (Local Use)**

'''python
import os
import cv2

# === CONFIGURATION ===
# Set the folder where your PNG images are located
folder_path = "path/to/your/images"

# Get all PNG files in the folder
image_files = [f for f in os.listdir(folder_path) if f.endswith('.png')]
image_files.sort()  # optional: process in order

# Labeling loop
for filename in image_files:
    file_path = os.path.join(folder_path, filename)

    # Load and show the image
    img = cv2.imread(file_path)
    cv2.imshow('Label this image', img)

    # Wait for a single key press
    print(f"Label for: {filename} -> Press a key")
    key = cv2.waitKey(0)  # waits indefinitely for a key press
    if key == 27:  # ESC key to quit
        print("Exiting.")
        break

    label = chr(key).upper()  # Convert to uppercase
    new_filename = f"{label}_{filename}"
    new_path = os.path.join(folder_path, new_filename)

    os.rename(file_path, new_path)
    print(f"Renamed to: {new_filename}")
```

```
cv2.destroyAllWindows()
```

To Run It in VS Code:

2. Install OpenCV:

```bash
pip install opencv-python
```

4. Replace `folder_path` with the path to your image folder.

6. Run the script, and press one key per image. It renames files like `A_001.png`.

8. Press `ESC` to stop labeling.

Want it in Google Colab?

No problem here's a version that works with **Google Drive + IPython GUI** :

Would you like that version **next**?

You:

yes please I would like it **for** google colab

ChatGPT:

Great! Here's the **Google Colab** version of the labeling script. It:

2. Mounts your Google Drive

4. Lets you select the folder with the unlabeled PNGs

6. Displays each image one at a time in a notebook cell

8. Waits for you to type a label and press Enter

10. Renames the image with that label

Step-by-Step Google Colab Script

```python
# STEP 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

Run that cell, click the link, and authorize access to your Google Drive.

```python
# STEP 2: Set path to the folder containing your images
# Example: '/content/drive/MyDrive/letter_images'
import os

folder_path = '/content/drive/MyDrive/letter_images'
assert os.path.exists(folder_path), "Check your folder path!"
image_files = [f for f in os.listdir(folder_path) if f.endswith('.png')]
image_files.sort()
print(f"Found {len(image_files)} images.")
```

```python
# STEP 3: Helper function to display an image and get a label
from IPython.display import display
from PIL import Image
import ipywidgets as widgets

def label_image(filename):
    full_path = os.path.join(folder_path, filename)
    img = Image.open(full_path)
    display(img)

    text_input = widgets.Text(
        value='',
        placeholder='Type label here and press Enter',
        description='Label:',
        disabled=False
    )
    display(text_input)

    # Wait for user input
    output = widgets.Output()

    def handle_submit(sender):
        label = text_input.value.strip().upper()
        if label:
            new_name = f"{label}_{filename}"
            new_path = os.path.join(folder_path, new_name)
            os.rename(full_path, new_path)
            with output:
                print(f"Renamed to: {new_name}")
        else:
            with output:
                print("Skipped (empty label).")

    text_input.on_submit(handle_submit)
    display(output)
```

```python
# STEP 4: Label each image one by one
# You can re-run this cell to continue labeling remaining files

for filename in image_files:
    if '_' not in filename:  # skip already-labeled files
        print(f"\n--- Labeling: {filename} ---")
        label_image(filename)
        break  # Show one at a time
```

```
## To label the next image:

Just re—run the **last cell** (Step 4) to show the next unlabeled image.

This keeps your notebook responsive and avoids clutter.

---

Let me know if you'd like to:

- Label with more than one character

- Store labels in a CSV instead of renaming

- Undo a label

- Display labels in a GUI dropdown

Happy labeling!
```

CNN comparison and debugging

You:

why would it be that this network works very well for the training of a small self driving robot:

```
def create_model():
    model = models.Sequential([
        layers.Rescaling(1./255, input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),

        # Enhanced convolutional base
        layers.Conv2D(64, (5, 5), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.2),

        layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.3),

        layers.Conv2D(256, (3, 3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.GlobalAveragePooling2D(),
        layers.Dropout(0.4),

        # Dense layers with L2 regularization
        layers.Dense(512, activation='relu', kernel_regularizer='l2'),
        layers.Dense(256, activation='relu', kernel_regularizer='l2'),
        layers.Dense(2, activation='tanh')
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005),  # Lower initial LR
        loss='mse',
        metrics=['mae']
    )
    return model
```

but this one basically does not work at all?:

```
def create_model():
    inputs = tf.keras.Input(shape=(IMG_HEIGHT, IMG_WIDTH, 3))

    x = layers.Rescaling(1./255)(inputs)

    # Convolutional base
    x = layers.SeparableConv2D(64, (7, 7), activation='relu', padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((2, 2))(x)
    x = layers.Dropout(0.2)(x)
```

```
    x = layers.SeparableConv2D(128, (5, 5), activation='relu', padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((2, 2))(x)
    x = layers.Dropout(0.3)(x)

    x = layers.SeparableConv2D(256, (3, 3), activation='relu', padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.4)(x)

    # Split here
    # Steering head
    s = layers.Dense(128, activation='relu')(x)
    s = layers.Dense(64, activation='relu')(s)
    steering = layers.Dense(1, activation='tanh', name='steering')(s)

    # Velocity head
    v = layers.Dense(128, activation='relu')(x)
    v = layers.Dense(64, activation='relu')(v)
    velocity = layers.Dense(1, activation='sigmoid', name='velocity')(v)

    model = tf.keras.Model(inputs=inputs, outputs=[velocity, steering])

    model.compile(
        optimizer='adam',
        loss={'velocity': 'mse', 'steering': 'mse'},
        metrics={'velocity': 'mae', 'steering': 'mae'}
    )

    return model
```

```
ChatGPT:

Great question   both models look well thought out, but the first works well for your robot
    while the second "basically does not work at all." Let's dive into **why**  that might be.

---

###    Key Differences and Potential Causes

1. **Output Structure and Loss Balancing**

- **Model 1**  outputs a single vector of two values (`Dense(2, activation='tanh')`)   
    presumably `[steering, velocity]`   and uses a **single loss**  (`'mse'`).

- **Model 2**  splits the outputs: `steering` with `tanh` and `velocity` with `sigmoid`, using **
    two separate heads and two losses**.

   *Model 1* lets the network learn the optimal tradeoff between predicting steering and velocity
    , which is often important in robotics where these outputs are correlated.
     *Model 2* may suffer from **unbalanced gradients**  if the magnitudes of velocity and
    steering losses differ significantly   especially since `velocity` uses `sigmoid` and is
    thus bounded between [0,1], while `steering` uses `tanh` in [-1,1].

Even if both losses are MSE, the optimizer might focus disproportionately on one head (often
    whichever has higher error magnitude), destabilizing training.
**Fix:**  Try using `loss_weights={'velocity':x, 'steering':y}` to balance losses if you stick
    with two heads. But usually combining outputs into one `Dense(2)` and decoding later is
    simpler and more stable.

---

2. **SeparableConv2D vs Conv2D**

- **Model 1**  uses standard `Conv2D`, while **Model 2**  uses `SeparableConv2D`.

     In small data or simple real-world tasks (like self-driving toy robots), `SeparableConv2D`
    often underperforms due to reduced capacity. It's more efficient, but less expressive.
**Fix:**  Try replacing `SeparableConv2D` with `Conv2D` to test if this is the issue.

---

3. **Output Activations:**
```

– **Model 1:** ‘Dense(2, activation=’tanh’)‘ both values in [−1,1], nice for steering and velocity if normalized accordingly.

– **Model 2:** ‘tanh‘ for steering (fine), but ‘sigmoid‘ for velocity.

If your training data has **negative or >1 velocity values** (e.g., reverse, full throttle), ‘sigmoid‘ w o n t work well. It restricts output to [0,1].
Fix: Either:

– Normalize velocity to [0,1] and rescale after prediction, or

– Use ‘tanh‘ or ‘linear‘ if you expect a wider range.

---

4. **Complexity of the Head Structure**

– **Model 1** uses one shared dense head: simpler and more generalizable.

– **Model 2** splits the head and uses more layers.

This may overfit or underfit depending on the data volume. If your dataset is small, the model may struggle to learn effectively from the extra complexity.

---

5. **Regularization**

– **Model 1** uses ‘kernel_regularizer=’l2’‘ in the dense layers.

– **Model 2** has no L2 regularization.

L2 regularization helps generalization and prevents overfitting, especially when paired with dropout.

Summary Fixes to Try in Model 2

2. **Combine velocity and steering into one head like in Model 1.**

4. **
Replace ‘SeparableConv2D‘ with ‘Conv2D‘** , at least initially.

6. **
Use ‘tanh‘ or ‘linear‘ for both outputs** , or make sure velocity data is scaled to [0,1].

8. **Add L2 regularization** to dense layers.

10. If keeping dual heads: **
If keeping dual heads: use ‘loss_weights‘** to balance training.

Would you like me to modify your second model to try these suggestions out directly?

You:

the goal with model two was to make the model more efficient since it has to run in a real time application

ChatGPT:

Got it you're trying to **optimize for inference speed and efficiency** , which totally makes sense for real-time control on a self-driving robot.
Heres a breakdown of **why Model 2 might still underperform despite being more efficient** , and how you might tweak it to **preserve performance while improving runtime** .

Efficiency vs. Expressivity Tradeoff

What you did right:

- Used `SeparableConv2D`, which dramatically reduces parameter count and computation (great for edge devices).
- Used `GlobalAveragePooling2D`, which replaces large dense layers and reduces overfitting.
- Split the outputs for cleaner postprocessing and possibly more targeted learning.

But the drop in performance likely comes from a **stack of small, compounding issues** that hurt learning, especially early in training.

Targeted Fixes While Keeping Model 2 Efficient

Heres how to **keep the efficiency** but fix the training issues:

---

1. **Use a Single Output Head Temporarily**

Even though you want split outputs, try training with:

```python
x = layers.Dense(128, activation='relu')(x)
x = layers.Dense(64, activation='relu')(x)
output = layers.Dense(2, activation='tanh')(x)
```

And then **split the output manually in inference** :

```python
velocity, steering = output[:, 0], output[:, 1]
```

Why? This helps the network **co-learn the relationship** between steering and velocity they re often coupled in real driving. You can go back to split heads later once you get it to train well.

---

2. **Got it you're trying to **optimize for inference speed and efficiency** , which totally makes sense for real-time control on a self-driving robot.
Heres a breakdown of **why Model 2 might still underperform despite being more efficient** , and how you might tweak it to **preserve performance while improving runtime** .

Efficiency vs. Expressivity Tradeoff

What you did right:

— Used ‘SeparableConv2D‘, which dramatically reduces parameter count **and** computation (great **for** edge devices).

— Used ‘GlobalAveragePooling2D‘, which replaces large dense layers **and** reduces overfitting.

— Split the outputs **for** cleaner postprocessing **and** possibly more targeted learning.

But the drop **in** performance likely comes **from** a **stack of small, compounding issues** that hurt learning, especially early **in** training.

——

Targeted Fixes While Keeping Model 2 Efficient

Heres how to **keep the efficiency** but fix the training issues:

——

1. **Use a Single Output Head Temporarily**

Even though you want split outputs, **try** training with:

```
```python
x = layers.Dense(128, activation='relu')(x)
x = layers.Dense(64, activation='relu')(x)
output = layers.Dense(2, activation='tanh')(x)
```
```

And then **split the output manually **in** inference** :

```
```python
velocity, steering = output[:, 0], output[:, 1]
```
```

Why? This helps the network **co—learn the relationship** between steering **and** velocity theyre often coupled **in** real driving. You can go back to split heads later once you get it to train well.

——

2. Use ‘tanh‘ **or** ‘linear‘ **for** velocity**
If your target velocity values arent strictly **in** [0,1], ‘sigmoid‘ will be a bottleneck.
Use ‘tanh‘ (**if** values are normalized to [−1,1]) **or** ‘linear‘ **if** you want raw output.

——

3. **Got it you're-trying-to-**optimize-for-inference-speed-and-efficiency**-,-which-totally-makes-sense-for-real—time-control-on-a-self—driving-robot.
Heres-a-breakdown-of-**why-Model-2-might-still-underperform-despite-being-more-efficient**-,-and-how-you-might-tweak-it-to-**preserve-performance-while-improving-runtime**-.

——

##- -Efficiency-vs.-Expressivity-Tradeoff

###- -What-you-did-right:

—- -Used-‘SeparableConv2D‘,-which-dramatically-reduces-parameter-count-and-computation-(great-for-edge-devices).

—- -Used-‘GlobalAveragePooling2D‘,-which-replaces-large-dense-layers-and-reduces-overfitting.

—- -Split-the-outputs-for-cleaner-postprocessing-and-possibly-more-targeted-learning.

But the drop in performance likely comes from a **stack of small, compounding issues** that hurt learning, especially early in training.

Targeted Fixes While Keeping Model 2 Efficient

Heres how to **keep the efficiency** but fix the training issues:

---

1. **Use a Single Output Head Temporarily**

Even though you want split outputs, try training with:

```python
x = layers.Dense(128, activation='relu')(x)
x = layers.Dense(64, activation='relu')(x)
output = layers.Dense(2, activation='tanh')(x)
```

And then **split the output manually in inference**:

```python
velocity, steering = output[:, 0], output[:, 1]
```

Why? This helps the network **co-learn the relationship** between steering and velocity theyre often coupled in real driving. You can go back to split heads later once you get it to train well.

---

2. **Got it you're trying to **optimize for inference speed and efficiency** , which totally makes sense for real-time control on a self-driving robot.
Heres a breakdown of **why Model 2 might still underperform despite being more efficient** , and how you might tweak it to **preserve performance while improving runtime** .

Efficiency vs. Expressivity Tradeoff

What you did right:

- Used ‘SeparableConv2D‘, which dramatically reduces parameter count and computation (great for edge devices).
- Used ‘GlobalAveragePooling2D‘, which replaces large dense layers and reduces overfitting.
- Split the outputs for cleaner postprocessing and possibly more targeted learning.

But the drop in performance likely comes from a **stack of small, compounding issues** that hurt learning, especially early in training.

Targeted Fixes While Keeping Model 2 Efficient

Heres how to **keep the efficiency** but fix the training issues:

---

1. **Use a Single Output Head Temporarily**

Even though you want split outputs, try training with:

```python
x = layers.Dense(128, activation='relu')(x)
x = layers.Dense(64, activation='relu')(x)
output = layers.Dense(2, activation='tanh')(x)
```

And then **split the output manually in inference**:

```python
velocity, steering = output[:, 0], output[:, 1]
```

Why? This helps the network **co-learn the relationship** between steering and velocity theyre often coupled in real driving. You can go back to split heads later once you get it to train well.

---

2. Use 'tanh' or 'linear' for velocity**
If your target velocity values arent strictly in [0,1], 'sigmoid' will be a bottleneck.
Use 'tanh' (if values are normalized to [-1,1]) or 'linear' if you want raw output.

---

3. Try 'depthwise_separable_conv' only for later layers**

The first layer often needs to capture richer features. Try this hybrid:

```python
x = layers.Conv2D(32, (5,5), activation='relu', padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2,2))(x)

x = layers.SeparableConv2D(64, (3,3), activation='relu', padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2,2))(x)
```

This gives you a solid initial representation and then trims compute cost later.

---

4. **Add L2 Regularization**

This can help stabilize training especially when paired with batch norm and dropout:

```python
layers.Dense(128, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(1e-4))
```

---

5. **Use learning rate warmup or lower initial LR**

If you're finding training unstable, try this:

```python
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
```

Or even better: use a scheduler that ramps up learning rate gradually in the first few epochs.

Bottom Line

To balance **real-time performance and good training**:

- Start simple: get it to train well with lightweight tweaks.

- Profile the network (e.g., with TensorFlow Lite) once it works.

- Then make structured tradeoffs for runtime (e.g., quantization, pruning, etc.).

Would you like me to write a revised `create_model()` that follows this balance lightweight but good performance?

You:

just out of curiosity, could I use the convolutional layers from the good model, keep them unchanged and hold the weights fixed, and then append the two heads?

ChatGPT:

Yes, absolutely and thats actually a **very common and smart approach** in deep learning! Its called **feature extraction** or using a **frozen backbone**.

What Youre Suggesting:

You're asking:

> Can I **reuse the convolutional base from the good model**, freeze its weights, and then **attach the two-output heads** (steering and velocity) for efficiency?

Yes, you can. Here's how it works:

Why This Might Work Well

- The convolutional base has **already learned good visual features** from your data (edges, shapes, road structure, etc.).

- Freezing it:

  - Reduces training time.

  - Prevents overfitting (especially on small datasets).

  - Keeps inference-time speed manageable.

- Then, a small custom head can **learn to decode** those features into steering and velocity.

How to Do It in Code

Step 1: Get the trained base model

Lets say your original "good" model is called `old_model`.

You can extract and freeze the base:

```python
base = tf.keras.Sequential(old_model.layers[:-3])  # Exclude the Dense layers
base.trainable = False
```

Or, if using functional API:

```python
base_input = old_model.input
base_output = old_model.get_layer(index=-4).output  # GlobalAveragePooling2D layer
base_model = tf.keras.Model(inputs=base_input, outputs=base_output)
base_model.trainable = False
```

Step 2: Add your two heads

```python
x = base_model.output

# Steering head
s = layers.Dense(128, activation='relu')(x)
s = layers.Dense(64, activation='relu')(s)
steering = layers.Dense(1, activation='tanh', name='steering')(s)

# Velocity head
v = layers.Dense(128, activation='relu')(x)
v = layers.Dense(64, activation='relu')(v)
velocity = layers.Dense(1, activation='tanh', name='velocity')(v)

model = tf.keras.Model(inputs=base_model.input, outputs=[velocity, steering])

model.compile(
    optimizer='adam',
    loss={'velocity': 'mse', 'steering': 'mse'},
    metrics={'velocity': 'mae', 'steering': 'mae'}
)
```

Extra Option: Gradual Unfreezing

Once the heads start learning well, you can **unfreeze the base and fine-tune** (just a little!) for better performance:

```python
base_model.trainable = True
model.compile(...)  # Re-compile with lower learning rate
```

This is especially effective if you're using transfer learning-like strategies.

Bonus: Real-Time Performance

Even with the larger convolutional base:

- You only train the lightweight heads.
- For real-time use, convert the full model (frozen base + trained heads) to **TensorFlow Lite** , **prune** , **or** **quantize** to optimize speed.

Let me know **if** you want the full code to freeze your trained base **and** add heads cleanly!

7.2 Appendix B: Training Data

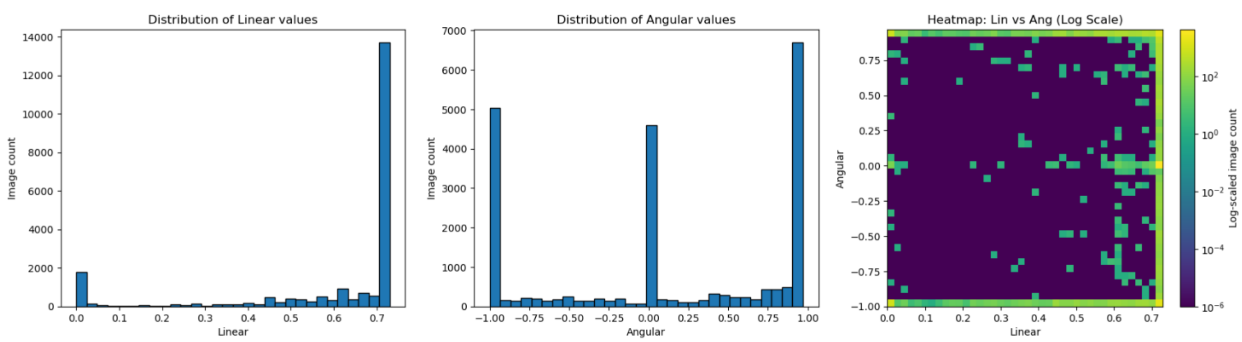


Figure 14: Training Data Distributions for Gravel Section

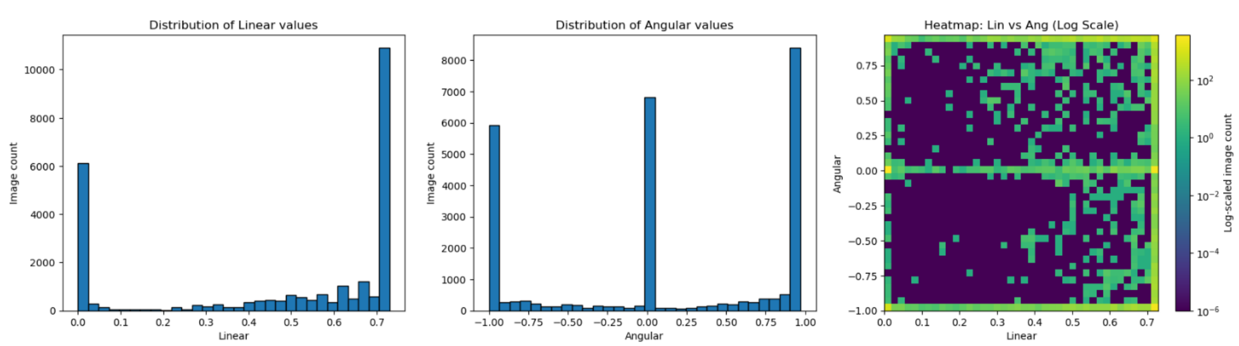


Figure 15: Training Data Distributions for Off-Road Section

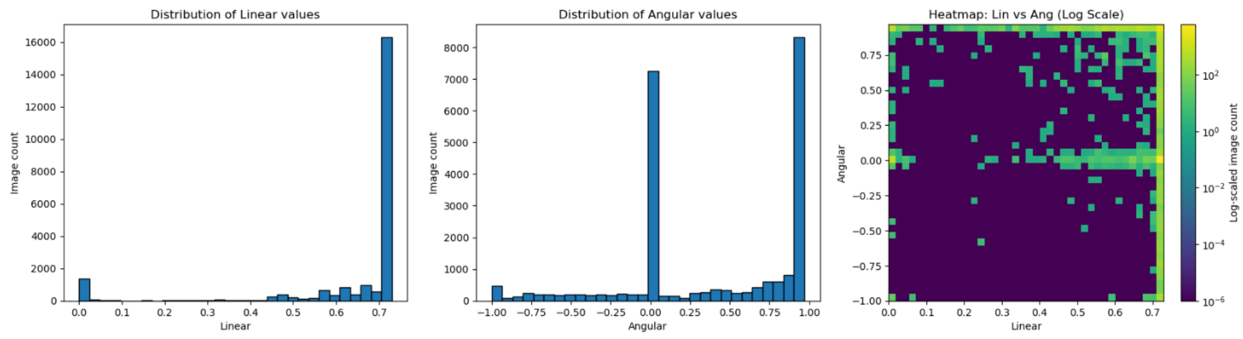


Figure 16: Training Data Distributions for Hill Section