Duckietown PIDPlus Project

IFT6757 - Machine Learning Project

Simon-Olivier Duguay Navid Hassan Zadeh

December 24, 2024

Contents

1 Introduction

Autonomous navigation requires reliable and adaptable algorithms to navigate complex environments. A simplified problem in this domain is lane following. To address this problem, the traditional lane-following algorithm for Duckiebot includes line detection, ground projection, lane filtering, and a PID controller. Although this approach is lightweight and fairly robust, there are cases where it fails to meet expectations.

The goal of this project is to improve lane-following with PID performance using a new machine learning-based measurement model. Specifically, the objective is to have a model that can predict the distance of the robot to the center of the lane, $\hat{y}_t$, given only the robot's camera image.

In this project, after capturing the data from the Duckiebot's camera feed, we apply some preprocessing steps to remove unwanted noise and extract interesting features from the video feed. The images are then ran through Convolutional Neural Networks (CNNs) or recursive LSTM-based Neural networks (RNNs), trained using the Gym-Duckietown simulator. The model will then output a predicted distance to the center of the lane. To smooth the output values of our model, we implement a kalman filter. The values are then fed into a basic PID controller to achieve smooth control.

We trained six different models (two CNN models and four RNN models) with different settings and compared their performance using a custom-made benchmark dataset. We tested all models on the simulation.

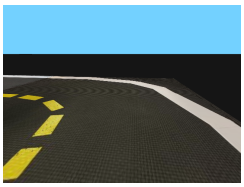
1.1 Motivation

The motivation for this project is to use machine learning to improve the traditional lane-following algorithm. By integrating a CNN or RNN to predict the lane position directly from images, this approach aims to provide a simpler and potentially more robust alternative, while still relying on the simplicity and strength of a basic PID control.

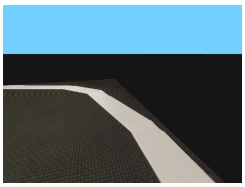
2 Dataset

2.1 Data Creation

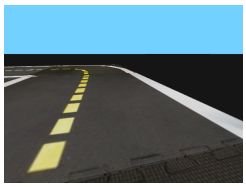
Three custom datasets with a total of 27,000 images were created using the Gym-Duckietown simulator. For this, we manually drove the robot in the simulation environment and recorded the necessary information to train our models. At each frame, we captured three pieces of information, namely the image from the camera, a label representing the deviation (distance) from the center of the lane, and the action values (speed and angular velocity) representing the action taken by the robot at each frame. In addition to taht, we created a benchmark dataset to evaluate our models. All of our datasets are available to download from the following link: https://1drv.ms/u/s!AmxJyID0MPIzlZ1eQO8Wp9isMPlmOg?e=r5OoxG



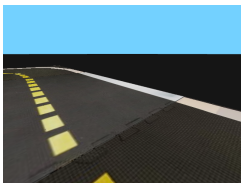
(a) 0.00293298m



(b) 0.05127094m



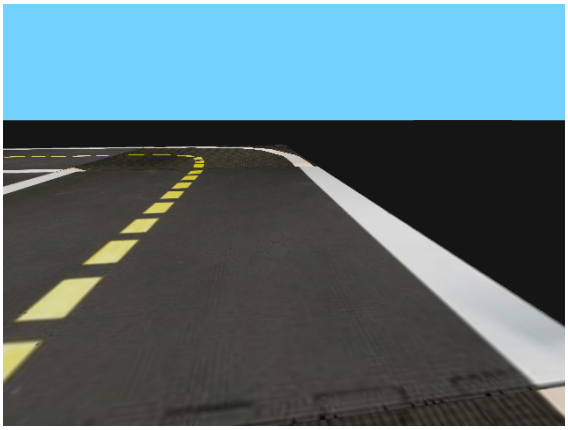
(c) -0.06203061m



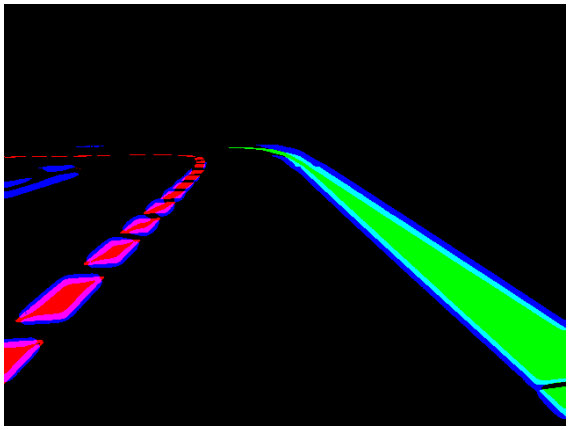
(d) -0.16589936m

Figure 1: Four examples from the dataset and their true label given by the simulation (i.e. distance from the center of the lane in meters). Note that positive label values mean to the right of the lane's center, and negative values mean to the left of the lane's center.

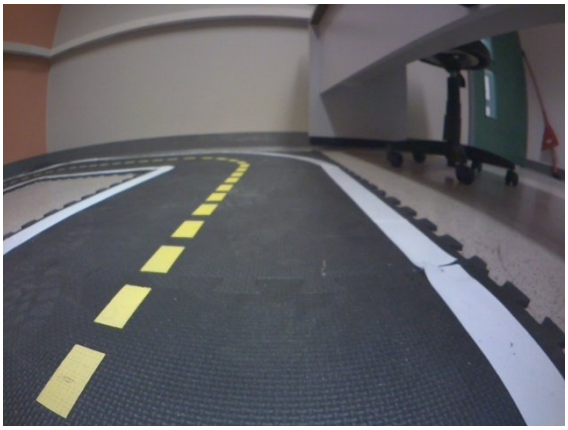
2.2 Preprocessing



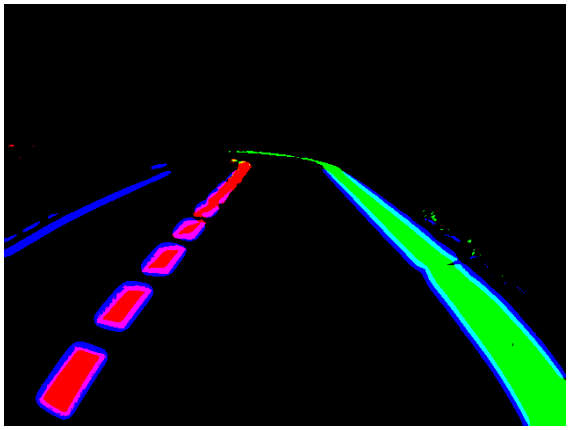
(a) Original Image (Simulation)



(b) Preprocessed Image (Simulation)



(c) Original Image (real-world)



(d) Preprocessed Image (real-world)

Figure 2: Two examples of images from the simulation and real-world duckietown environments and their preprocessed versions.

Preprocessing is an important step for any machine learning project. It ensures that input data is clean, consistent and in a format suitable for the model. Usually it consists of removing noise, normalizing values and extracting relevant features, impacting the model's

ability to learn patterns and make more accurate prediction.

In our case, the preprocessing adds another important value. It allows us to fill the gap from simulation data to real-life data. While still far from perfect, the simulation data becomes similar to the real-life pictures after transforming the images with masks and blurs. This allows our model to find some common ground and transfer the knowledge acquired with the help of the simulator to the real world. For comparison, two examples are provided in Figure 2 to show the image preprocessing and its effect on simulation vs. real-world images.
For preprocessing, the images underwent the following steps:

1. **Image Conversion and Color Space Transformation:**

   The input image is first converted to a NumPy array. The image is also converted to grayscale for edge detection and to HSV (Hue, Saturation, Value) format for color-based filtering. These transformations are essential for detecting lane markings (white and yellow).

2. **Noise Reduction and Edge Detection:**

   A Gaussian blur with a standard deviation of 4.5 is applied to the grayscale image to reduce noise and stabilize edge detection. Horizontal and vertical gradients are computed using Sobel operators, and their magnitudes are combined to create an edge map. A threshold is applied to highlight significant edges and remove the noise.

3. **Color and Horizon Masking:**

   Color masks are created using HSV thresholds to isolate white and yellow lane markings. These masks are further refined by cropping specific regions of the image: both the yellow mask and the white mask are cropped by 40% of the image width, with the yellow mask cropped on the right and the white mask cropped on the left. Additionally, a horizon mask is applied to exclude the top one-third of the image height, which removes irrelevant background information and is particularly useful for the real-world robot.

4. **Mask Combination and Multi-Channel Output:**

   The processed edge map, white mask, and yellow mask are combined into a three-channel image. The first channel represents the edge magnitude, the second channel highlights white lanes, and the third channel highlights yellow lanes. This final image is used as input to train the model.

3 Model Architectures

3.1 CNN Model

Our initial test was to use a Convolutional Nerual Network (CNN) with a series of convolutional, pooling and fully connected layers. In our experiments, we tested two different

versions of this model. These are called: CNN1 (without image preprocessing) and CNN2 (with image preprocessing). The architecture starts with five convolutional layers taht extract spatial features from the input image. Each convolutional layer uses ReLU activation and a max pooling layer. We also added a dropout layer after the five convolutions to prevent overfitting. At the end we added one or two MLP layers, with ReLU activation function. The number of neurons or layers were decided after some trial and error to optimize its performance. Finally, the output goes through a tanh function and scaled to the range of [-0.5, 0.5].

3.2 CNN+RNN Model

We created a custom CNN+RNN architecture to evaluate the impact of a different model. This model architecture incorporates the previous CNN model to process the images and extract their spatial features. Then an LSTM-based RNN model is added to take into account temporal dependencies of the input data and also capture the contextual information. We made four different versions of this model. For the first two models we fed the flattened CNN features in sequences of consecutive frames as input to our RNN model. We did that without preprocessing (RNN1) and also with preprocessing for images (RNN2). For the other two models, in addition to a sequnce of CNN features, we added another input to the RNN model. This other input is a fourier-transformed representaion of the action (i.e. speed and angular velocity) taken at each of the time steps. (NOte that for each image, we are feeding the action taken at the time the previous image was recorded to enable deploying this model in real time)

Our reasoning behind incorporating the action inputs which are speed and angular velocity was that these information should prove very useful in enhancing the temporal context between frames. These control commands taken by the robot directly influence the observed lane positions in subsequent frames. Therefore, ideally, our aim is to enable the model to learn about the relationship between the vehicle's motion commands and the changes in the lane's position with respect to the robot, which should lead to a more accurate prediction by the model. For example, suppose the robot is 5cm to the right of the lane's center and the robot's action values have been to go left at a particular speed in the previous few time steps. Purely based on these actions, the model can already have a good idea about the range of positions it might find itself in the future frames and therefore limit the domain of its prediction significantly. At the very least the model knows, for example, that if the robot keeps going left, it is highly unlikely that the next prediction would be anything larger than 5 cm to the right of the lane's center. The idea of using these action values in addition to the camera images made a lot of sense to us given that predicting based on a single frame can be unstable and a waste of useful information already freely available to us as we know the dynamics of the problem at hand.

To represent the action inputs effectively we applied the following transformation to encode each action variable into a richer and higher dimensional feature space. Specifically, each action (e.g., speed or angular velocity) is transformed using sine and cosine functions at multiple frequencies:

$$f_{\text{Fourier}}(v) = \left[\sin(2^0 v), \cos(2^0 v), \sin(2^1 v), \cos(2^1 v), \ldots, \sin(2^{n-1} v), \cos(2^{n-1} v)\right]$$

where v is the action value (e.g., speed or angular velocity), and n is the number of frequencies used. In our implementation, we used 6 frequencies for both the speed and angular velocity. This resulted in a concatenated vector of size 24 given that we have two action values and each one of them gave a vector of size 12.

By combining the transformed action features with the CNN-extracted spatial features, the model was expected to ideally gain the ability to incorporate temporal dependencies and vehicle dynamics into its predictions for better and more consistent predictions.

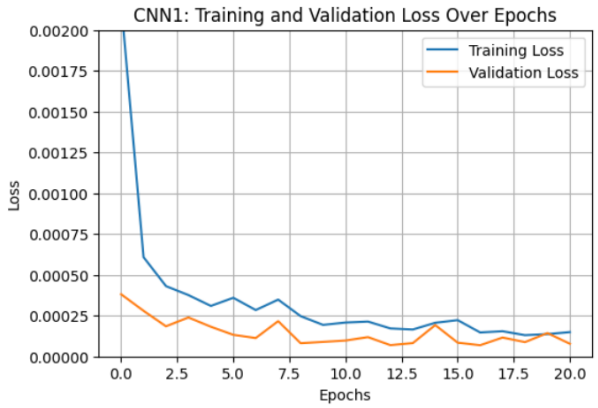
4 Training

The models were trained using the Mean Squared Error (MSE) loss function. We used an Nvidia RTX4080 for the training of all our models. In our study we used two CNN models: one with image preprocessing applied and the other without preprocessing. Additionally, four RNN models were trained, two of which processed only sequences of images, while the remaining two used both image sequences and action values as inputs of the RNN model. For each of these scenarios, models were trained both with and without image preprocessing. Please note that the sequence length chosen for training all of the RNN models was 20 images (and actions) per sequence. The table 1 below summarize the detailed configurations of each model as well as some of their training parameters and their training time.

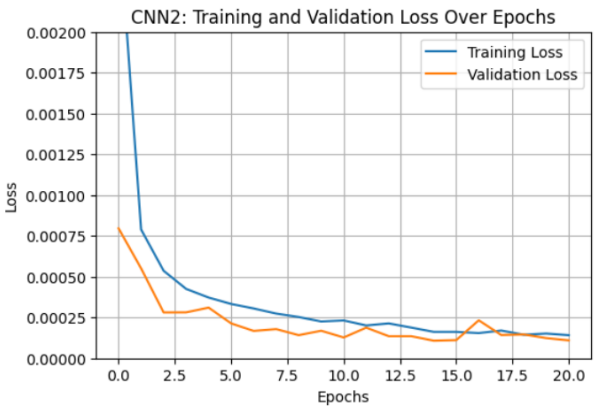
Model	**Input Type**	**Pre processing**	**Epochs**	**Batch Size**	**Optimizer**	**Layers**	**Train Time (min)**
CNN1	Single Image	No	20	16	Adam	5 CNN (conv2d, maxpool)	50 min
CNN2	Single Image	Yes	20	16	Adam	5 CNN (conv2d, maxpool)	89 min
RNN1	Sequence of Images	No	31	10	Adam	5 LSTM + 5 CNN	62 min
RNN2	Sequence of Images	Yes	20	10	Adam	3 LSTM + 5 CNN	82 min
RNN3	Sequence of Images + Actions	No	31	10	Adam	3 LSTM + 5 CNN	67 min
RNN4	Sequence of Images + Actions	Yes	20	10	Adam	5 LSTM + 5 CNN	114 min

Table 1: Configurations and Training Time for CNN and RNN Models

The following Figures 3 and 4 show the the training and validation loss for each of the 6 models we trained.

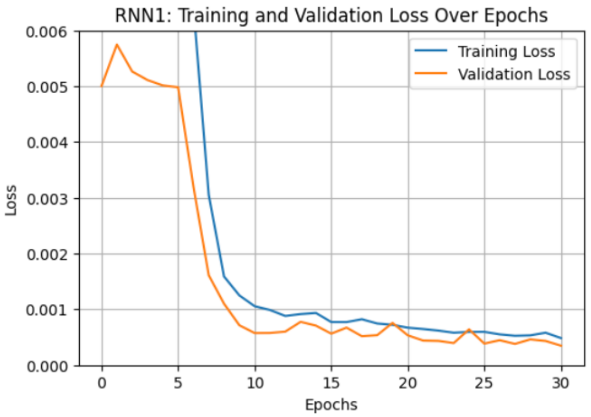


(a) CNN1: Training and validation loss of the CNN1 model (without image processing)

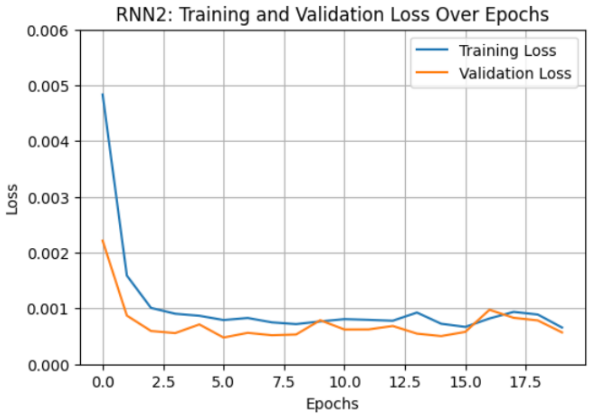


(b) CNN2: Training and validation loss of the CNN2 model (with image processing)

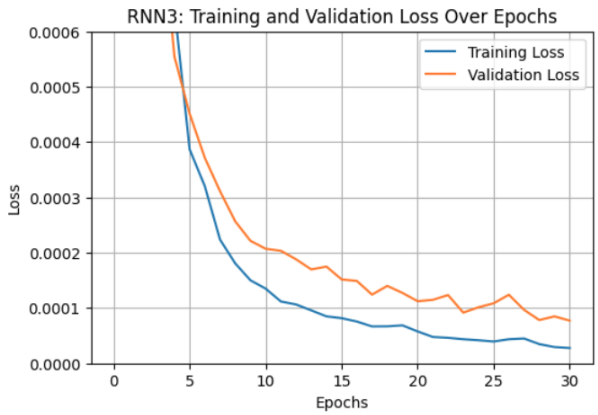
Figure 3: Training and Validation loss of the CNN models



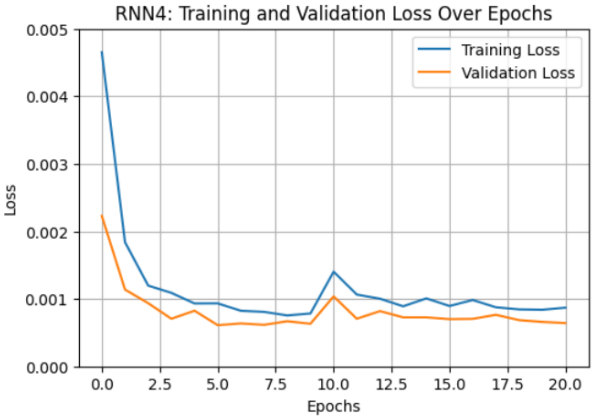
(a) RNN1: Training and validation loss (without image preprocessing, trained on sequence of images)



(b) RNN2: Training and validation loss (with image preprocessing, trained on sequence of images)



(c) RNN3: Training and validation loss (without image preprocessing, trained on sequence of images and actions)



(d) RNN4: Training and validation loss (with image preprocessing, trained on sequence of images and actions)

Figure 4: Training and validation loss for the RNN models

5 Benchmark

We created a benchmarking system to compare the models in our experiment. For this, we made a dataset consisting of 1,000 consecutive images. The performance of each model was evaluated by comparing their predictions with the true labels. The benchmark score for each model was calculated as the average of the absolute differences between the model's predictions and the true labels. The lower this score, the better. This score provides a quantitative measure to compare the accuracy of the models. The dataset used for benchmarking, along with all other datasets used in this experiment can be downloaded through a link provided in Section 2.1.

6 System Implementation

6.1 System Overview

The system developed for this project involves many components communicating and interacting with each other. The goal is to predict the distance to the center of the lane, $\hat{y}_t$, for then enabling reliable lane-following behavior.

Here are the key components :

1. **Camera Feed and Preprocessing**

   The Duckiebot captures a live feed from its front camera, which is then processed with many masks and filters to remove noise and extract relevant features.

2. **Machine Learning Model**

   The CNN or RNN model, trained on simulated data from the Gym-Duckietown simulator takes the preprocessed image as an input, and outputs $\hat{y}_t$, a value in the range of [-0.5, 0.5] representing it's distance from the center of the lane.

3. **Kalman Filter**

   To improve the reliability of the model's prediction, a Kalman Filter smooths the output values. The goal of the filter is to achieve smoother control, since there is a new prediction coming out of the network every frame.

4. **PID Controller**

   The filtered lane positions are then fed into a PID Controller, whose objective is to calculate the necessary motor commands to keep the Duckiebot in the center of the lane. It ensures smooth navigation by minimizing deviations from the target position.

6.2 ROS Architecture

The components of the system are integrated using ROS nodes to manage communication. Even though we are using some already integrated nodes from the duckiebot, we implemented two custom nodes for this project : The *model_node* and the *main_control_node*. We also implemented a topic for the communication between these nodes : *prediction*

- **Model Node**

  The model node subscribes to the *image* topic, which is sent by the Camera node. It also publishes to our own topic, called *prediction*, containing the output of the model. It is the node responsible for running our Machine Learning Model

  Once the component receives an image from the Camera node, it runs our preprocessing function before running a forward pass of our model. It then publishes the output to the *prediction* topic.

- **Main Control Node**

  The main control node subscribes to the *prediction* topic sent by the Model node. It publishes a $WheelsCmdStamped$ message to the $wheels\_cmd$ topic.

  From the prediction, this node is responsible to pass the data through the kalman filters and feed the values into the PID controller. It also sends the commands to the wheels which allows the duckiebot to move.

6.3 Model Deployment

The deployment of AI models directly onto a Duckiebot causes problems. Everything runs on a Jetson Nano, and there isn't enough RAM available for running pytorch and the core ROS nodes at the same time. The solution is then to run our nodes locally on our laptops. We rely on the internet connection for recieving and sending information with the core nodes. We then simply had to upload our weights into the Docker container and load our model with these weights.

Although this solution worked great, it remains highly sensitive to the network connectivity, which can be unpredictable and influenced by various factors, such as the number of people and ducks using it in the lab.

6.4 Control Integration

With the predicted positions, we smooth out the values with our Kalman filter before feeding them into the PID controller. A significant portion of the work in this stage invovled tuning the controller. The three primary parameters to adjust are k_p, k_i, and k_d. To simplify the process, we kept k_i at 0 and focused on tuning k_p first, followed by k_d.

Ultimately, we achieved a reasonably good calibration, but the tuning process required a considerable amount of time. This is an aspect that can always be further refined and adjusted for better performance.

7 Results

In this section, we present the results of our experiments. We evaluated both CNN and RNN models on the simulation, and additionally tested the CNN model on a real robot.

Our experiments gave promising results, and we successfully demonstrated the effectiveness of our approach.

7.1 Benchmark Results

Table 2 summarizes the performance of all our models on the benchmark dataset. The benchmark score (performance) is calcualted by averaging the absolute difference of the real distance values and predicted distance values for each model on all the 1000 images in the benchmark dataset.

Model	Preprocessing	Training Data	Benchmark Performance
CNN1	No	Single Images	0.0054951
CNN2	Yes	Single Images	**0.0049837**
RNN1	No	Sequence of Images	0.0148485
RNN2	Yes	Sequence of Images	0.0182638
RNN3	No	Sequence of Images + Actions	**0.0037412**
RNN4	Yes	Sequence of Images + Actions	0.0177134

Table 2: Benchmark performance and additional information about CNN and RNN models. These scores represent average prediction error. The lower the bettter.

Figures 5 and 6 shows the examples of predictions for the CNN models on individual images, while Figures 7, 8, 9, and 10 show the results for the RNN models for a consecutive sequence of images.

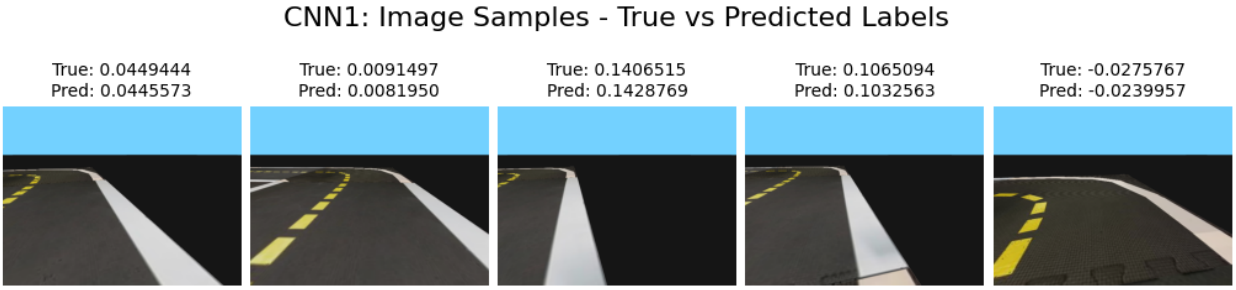


Figure 5: Five different predictions of CNN1 Model (No preprocessing, trained on single images)

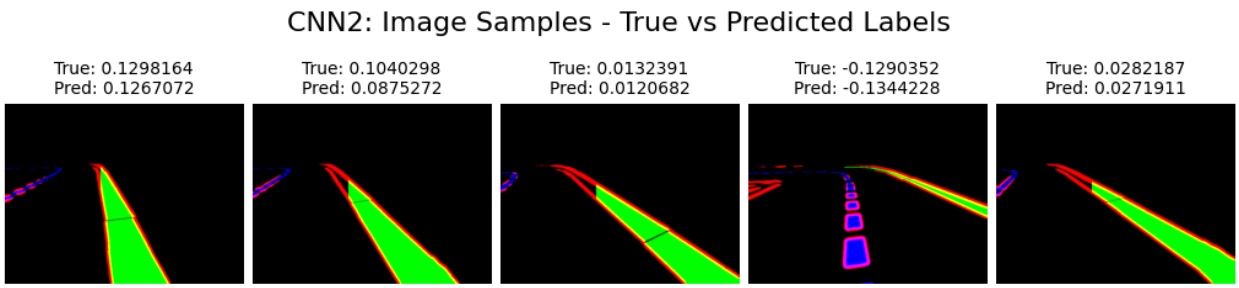


Figure 6: Five different predictions of CNN2 Model (With preprocessing, trained on single images)

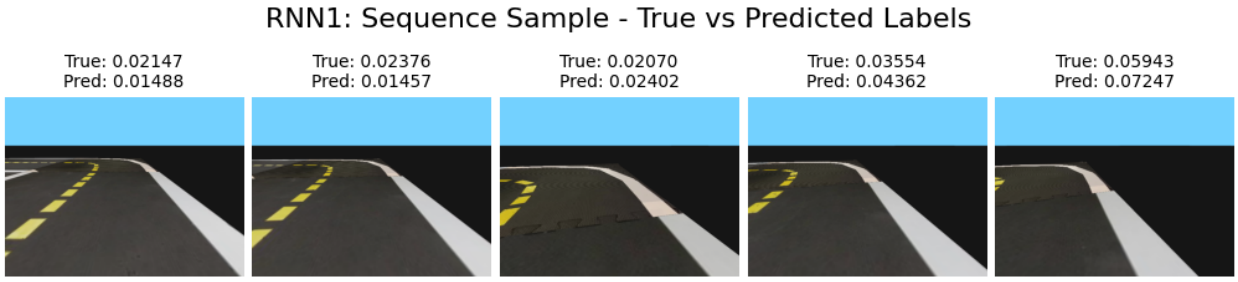


Figure 7: Prediction results for RNN1 Model (No preprocessing, trained on image sequences)

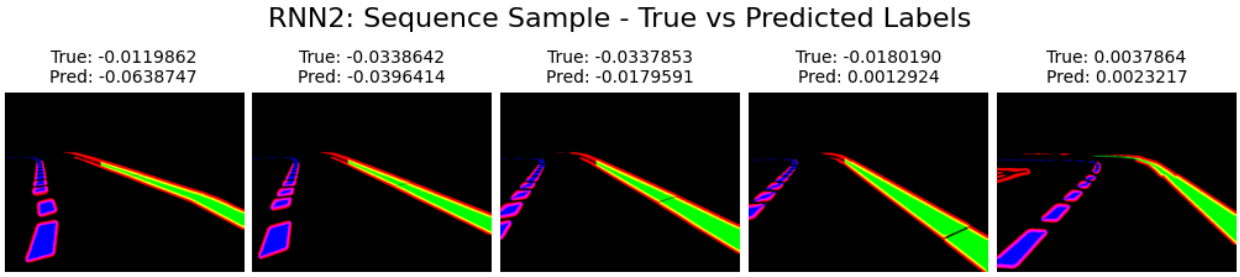


Figure 8: Prediction results for RNN2 Model (With preprocessing, trained on image sequences)

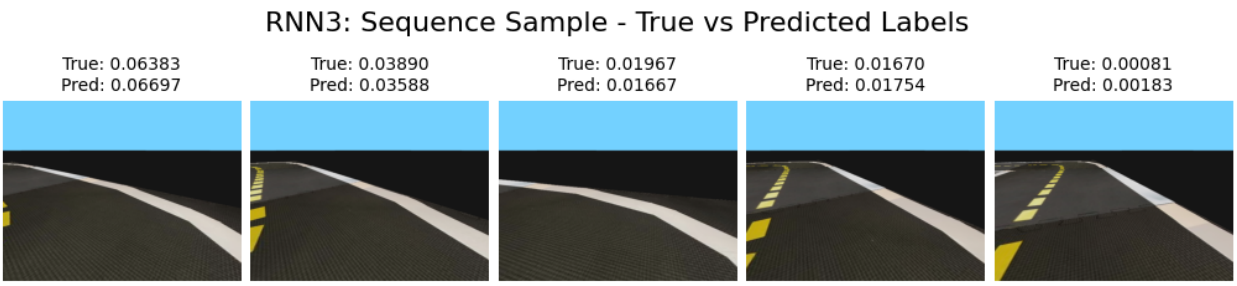


Figure 9: Prediction results for RNN3 Model (No preprocessing, trained on image and action sequences)

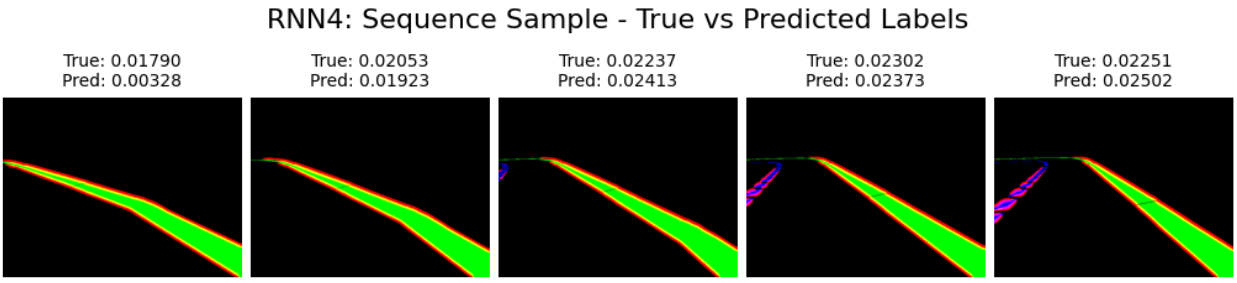


Figure 10: Prediction results for RNN4 Model (With preprocessing, trained on image and action sequences)

7.2 On the simulation

We tested all of our six models on the simulation. While their performance varied and some of them had a bit of a zig zag movement, they all succeeded in enabling the robot to turn around the loop with a relatively stable maneuvering. We didn't explore the PID parameter turning much in the simulation as the restults were already satisfactory. But with more optimized parameters, the movements should become smoother and less zig-zagy. The following link contains the video demonstrations of all the models on the simulation environment: https://1drv.ms/f/s!AmxJyID0MPIzlZ1XlUqCfTuRyIBysg?e=ewO5Lw

7.3 On the Real Bot

The real-world demonstration had mixed results. While the simulation was successful and worked almost flawlessly given a good tuning of the PID parameters, the real-world experiments faced certain issues. In our real duckiebot tests since our models were relateively large, we ran them on a computer. We sent and received the inputs and outputs of the model over the network with the help of ROS. With this approach, our biggest problem was the latency issues which limited our testing capacity. We only managed to test the CNN

model with image preprocessing (CNN2). After some testing and a lot of PID paramter tuning, we obtained some promising results, but we also ran into some issues during our tests. The Duckiebot's movements were a bit jerky, often over-steering and making excessive corrections. This often resulted in the robot going off course if operated for an extended period of time. This problem could be in part due to the latency in the transfer of images and other data on the network.

Despite these limitations, the system works with decent performance. It is dependent on the quality of the network connection and sensitive to real-world noise in the image, but it is capable of running 2 or 3 loops before encountering issues. We also noticed it stays more centered in its lane and crosses rarely into the other lane, something that the traditional lane-following algorithm struggles with.

While our project does not offer a replacement for the traditional algorithm, it demonstrates that using a machine-learning-based measurement model can work on Duckiebots and that there is a lot of potential in that area to find a more robust solution.

Here is a link to a small demonstration of our system running on a Duckiebot : PIDPlus demonstration

8 Discussion

Our proposed approach to PID's control variable measurement demonstrates promising integration and utilization of Machine Learning tools with classical PID control for lane-following tasks. Our CNN and RNN models enable us to give good predictions of the position of the robot. While the models' predictions are not totally accurate and sometimes are off by a few centimeters, they seem to easily and correctly predict positive- and negative-ness of the disance. (ie. whether the robot is on the right or left of the lane's center).

Our training was not done with the exact same model settings and architecture for different models as we tried to optimize it for each model though trial and error. In our results, we found that the CNN2 model (with preprocessing) could easily outperform the CNN1 model (without preprocessing). The ability of our CNN model to train well seemed to significantly depend on the MLP at the end of the CNN architecture. We used a 2 layer mlp and we used different number of neurons in them to optimize the performance and training of each model separately. With a longer training time, it seems that a deeper or wider MLP could results in more accurate predictions at the cost of a slower convergence. Similary, our RNN model parameters were selected by trial and error to optimize their performance. In the RNN results, it seemed the non preprocessing (RNN1 and RNN3) versions of the model outperform the preprocessing versions (RNN2 and RNN4). However, they seem to require a longer training in general. As expected, we noticed that our models' performances were highly dependent on their choice of architecture and it is possible that some optimization of hyperparameters of the current models could change their performance and comparison results. Due to the long time needed for training the models, we weren't able to do a more comprehensive hyperparameter optimization.

9 Reproducibility

All of the steps to reproduce every part of this project are available on their respective github pages, in the *README.md*.

Data collection and Training and Simulation: PIDPlus Training
System Integration and Ros nodes for real-world testing : PIDPlus

10 Conclusion

This project demonstrated the potential of machine-learning-based measurement models for lane-following tasks in autonomous driving. It proposed a solution that could help the traditional Duckiebot's lane-following algorithm overcome some of its limitations. While the proposed system achieved good enough precision in simulation, real-world performance had mixed results. It highlighted some challenges such as hardware limitation, network connectivity issues and sensitivity to noise. However, the system demonstrated some improvement over the traditional method, such as reducing crossovers to the other lane. This work emphasizes the potential of machine learning for these kind of problems and can pave the way toward more robust, efficient and practical real-world applications.

References

L. Paull et al., "Duckietown: An open, inexpensive and flexible platform for autonomy education and research," 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 2017, pp. 1497-1504, doi: 10.1109/ICRA.2017.7989179. keywords: Cameras;Roads;Robot sensing systems;Image color analysis;Education;Lighting;Calibration,