Imitation Learning and Direct Perception for Autonomous Driving

by

Rocky Liang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical and Mechatronics Engineering

Waterloo, Ontario, Canada, 2020

© Rocky Liang 2020

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis presents two learning based approaches to solve the autonomous driving problem: end-to-end imitation learning and direct visual perception. Imitation learning uses expert demonstrations to build a policy that serves as a sensory stimulus to action mapping. During inference, the policy takes in readings from the vehicle's onboard sensors such as cameras, radars, and lidars, and converts them to driving signals. Direct perception on the other hand uses these sensor readings to predict a set of features that define the system's operational state, or affordances, then these affordances are used by a physics based controller to drive the vehicle.

To reflect the context specific, multimodal nature of the driving task, these models should be aware of the context, which in this case is driver intention. During development of the imitation learning approach, two methods of conditioning the model were trialed. The first was providing the context as an input to the network, and the second was using a branched model with each branch representing a different context. The branched model showed superior performance, so branching was used to bring context awareness to the direct perception model as well.

There were no preexisting datasets to train the direct perception model, so a simulation based data recorder was built to create training data. By creating new data that included lane change behavior, the first direct perception model that includes lane change capabilities was trained.

Lastly, a kinematic and a dynamic controller were developed to complete the direct perception pipeline. Both take advantage of having access to road curvature. The kinematic controller has a hybrid feedforward-feedback structure where the road curvature is used as a feedforward term, and lane deviations are used as feedback terms. The dynamic controller is inspired by model predictive control. It iteratively solves for the optimal steering angle to get the vehicle to travel in a path that matches the reference curvature, while also being assisted by lane deviation feedback.

Acknowledgements

My rewarding journey through graduate school and my development as an engineer can be attributed to many and I want to take this opportunity to thank them all. Advice, guidance, and support from my supervisor Dr. Dongpu Cao has been invaluable during my time at Waterloo. I would also like to thank my labmates, who have provided an enjoyable learning environment that encouraged collaboration and innovation. Of course, I would like to thank my parents who made all this possible.

Table of Contents

List of Tables viii

List of Figures ix

List of Abbreviations xii

1 Introduction 1

1.1 Relevant Works 4

1.1.1 Imitation Learning 4

1.1.2 Direct Perception 6

1.1.3 Training Strategies 7

1.2 Tools and Methodologies 9

1.2.1 Deep Learning Frameworks 9

1.2.2 Simulation 10

2 Imitation Learning 12

2.1 Non Branched Conditional Network 13

2.1.1 Introduction 13

2.1.2 Context Usage 15

2.1.3 Data Balancing 16

2.1.4 CNN Bias Correction 19

2.1.5 Image Augmentation . . . 20
2.1.6 Production Considerations . . . 25
2.2 Branched Conditional Network . . . 25
2.2.1 Introduction . . . 25
2.2.2 How Branching Works . . . 26
2.2.3 Branch Training . . . 27
2.2.4 Temporal Learning . . . 29
2.3 Results . . . 30

3 Data Recorder 32
3.1 Recorder Client . . . 34
3.2 Labels . . . 36
3.2.1 Data Format . . . 36
3.2.2 High Level Command Handling . . . 36

4 Direct Perception 43
4.1 Motivation . . . 44
4.1.1 Comparison to Imitation Learning . . . 44
4.1.2 Comparison to Mediated Perception . . . 45
4.2 Affordance Selection . . . 46
4.2.1 Affordances for Lateral Control . . . 46
4.2.2 Affordances for Longitudinal Control . . . 49
4.3 Affordance Predictor . . . 50
4.3.1 Model . . . 50
4.3.2 Inference . . . 53
4.3.3 Offset Sensors . . . 53
4.4 Results . . . 55

5 Controller **59**

5.1 Kinematic Controller 59

5.1.1 K Stanley Controller 59

5.2 Dynamic Controller 61

5.2.1 Single Track Lateral Motion Model 61

5.2.2 Controller Derivation 63

5.2.3 Limitations 68

5.3 Results 73

6 Conclusion **78**

6.1 Summary 78

6.1.1 Imitation Learning 79

6.1.2 Direct Perception 79

6.1.3 Controller 80

6.2 Future Work 80

References **84**

APPENDICES **90**

A Code **91**

A.1 Branch Training Code 91

List of Tables

2.1 The high level commands used as contexts and their corresponding one-hot form . . . 16

2.2 Testing results for trained imitation learning models . . . 30

3.1 Average real world FPS of the recorder with the simulation set to 15 FPS, when using different libraries to save images . . . 35

3.2 Key labels saved by the recorder . . . 37

3.3 Labels for each nearby vehicle and pedestrian . . . 38

3.4 The high level commands used as contexts for the affordance predictor and their corresponding one-hot form . . . 40

4.1 Affordances predicted by the network . . . 50

4.2 Validation performance of different backbones with all other parameters held constant . . . 51

4.3 Results of direct perception in different scenarios . . . 58

5.1 Speeds above which oscillation and failure happen with each controller . . . 74

List of Figures

1.1 Qualitative plot that compares complexity vs. explainability for all 3 paradigms 3

1.2 High level workflow in policy creation 11

2.1 Input images to the network and saliency maps from select channels in different stages of convolution. It can be seen that each channel focuses on different things, e.g. some detect lane markings and some detect obstacles 14

2.2 Steering probability distribution at intersections. This thesis aims to train a policy that allows the user to select which "peak" to use, or which turn to take 15

2.3 Path of a vehicle traveling northbound (up), color coded by driver intention. Blue means follow lane, green means left turn, red means right turn 17

2.4 Steering and speed distributions of 5000 randomly sampled datapoints from the Conditional Imitation Learning dataset. Directly applying gradient descent to unbalanced data like this would only teach the network to output the most common values 18

2.5 Steering distribution of three batches of 512 datapoints obtained with the bin counter, each with different levels of thresholding 20

2.6 Non branched conditional imitation learning network 21

2.7 (a) Not enough augmentation can lead to overfitting and worse performance within original data range (green shaded portion) (b) Adequate augmentation covering a wider numerical range, better fit within original data range . 23

2.8 Images from the dataset after augmentation. (a) shows examples of slight augmentation (b) shows examples of severe augmentation, which provided less validation loss 24

2.9 Branched conditional imitation learning network 27

3.1 The data recording setup 33
3.2 Architecture of the data recording pipeline 35
3.3 Saved data organization 37
3.4 Button layout for activating different high level commands during data recording 38
3.5 As the user activates different high level commands, the waypoint queue (in green) shifts to their appropriate locations. (a) and (b) show the waypoint queue shifting for left and right lanechanges, (c) is in follow mode 41
3.6 User interface of the data recorder 42

4.1 The 3 paradigms in autonomous driving 44
4.2 A generalized affordance set will allow the direct perception pipeline to control different vehicles to drive the same way with minimal vehicle specific development 47
4.3 Illustration of heading and crosstrack errors 48
4.4 The direct perception pipeline, including the branched affordance predictor and controller 52
4.5 Simulated sensors can be offset from the main camera to generate datapoints with larger error states 54
4.6 Images from cameras yawed by -8 degrees (left), 8 degrees (right), and the main camera (center) 54
4.7 Common failure modes for direct perception seen in testing 56
4.8 Distance predictions vs. labels 57

5.1 The K Stanley controller attempts to match vehicle path curvature to road curvature, as well as minimize heading and crosstrack errors 60
5.2 Top down view of a right front tire during a right turn. When a rolling tire is pointed in a direction different than its velocity vector, lateral force is generated 62

5.3 A qualitative curve of lateral force vs. tire slip angle. Before a threshold slip angle, the relationship is linear 62

5.4 Curvature derivation based on driving around in a circle 65

5.5 Steering loss when the optimal steering angle is near 10 degrees 67

5.6 Loss curve and steering angle during the optimization process, which happens every timestep 67

5.7 Lateral controller based on the single track dynamic model 68

5.8 Step steer response of the single track lateral model. With its linear tire assumption, it provides some unlikely results even with a steering input of only 3 degrees 70

5.9 Longitudinal velocity and steering angle sweep of tire slip angle 71

5.10 Passenger tires' lateral grip typically leave the linear region at a slip angle of 5 degrees. This figure shows conditions of when the tire slip angles are under (green) and above (red) 5 degrees 72

5.11 Lane deviation trace during and after a high speed turn using the three different controllers 75

5.12 Lane deviation trace during and after a high speed obstacle avoidance maneuver using the three different controllers 76

5.13 Lane deviation trace during and after a lane change using the three different controllers 77

List of Abbreviations

CNN Convolutional Neural Network 5

DP Direct Perception 2

FC Fully Connected 4

GPS Global Positioning System 49

GRU Gated Recurrent Unit 29

HLC High Level Command 5

IL Imitation Learning 2

LQR Linear Quadratic Regulator 45

LSTM Long Short Term Memory 5

MAE Mean Average Error 27

MPC Model Predictive Control 5

MSE Mean Squared Error 27

PID Proportional Integral Derivative Controller 45

RL Reinforcement Learning 2

RNN Recurrent Neural Network 7

Chapter 1

Introduction

Autonomous driving is projected to be a $20bn market by 2025. It has the potential to completely transform our day to day lives. Once mass adoption begins, it will impact public infrastructure, urban planning, and food service among others, and it will create ripple effects that will be felt by industries far disconnected from transportation and mobility.

According to [14], there are three paradigms in autonomous driving: mediated perception, behavior reflex, and direct perception. The modular pipeline that consists of separate perception, planning, and control modules belongs to the mediated perception group, and as of now it is the most common approach. In mediated perception, the perception component builds a comprehensive reconstruction of the world from sensory stimulus. This can include information on all static and dynamic obstacles within a defined range of the vehicle. Then, the planning and control modules act on that digital reconstruction to get the vehicle to its goal. This requires a large amount of compute power onboard. The second paradigm, direct perception, only extracts what is necessary from the environment. A learning based perception policy is used to obtain **affordances**, which are variables that have meaningful attributes relating to the task at hand. The set of affordances are then used by a rules based controller to generate appropriate actions. The third type, behavior reflex, is a one-shot conversion from sensory stimulus to control action. Machine learning models are trained to to show appropriate behavior for a given measurement which for example can be camera images, radar readings, or lidar point clouds. The information learned from training is stored in a policy, or control law, that converts sensor stimulus directly to vehicle actuator commands. Training is done offline, then the trained policy can be run on vehicles with much less computation. This approach requires massive amounts of data to train, and there is the difficulty of proving safety guarantees with trained models. As

this stimulus to action mapping is typically stored in a neural network, it is also difficult to understand why it takes the actions that it does. Figure 1.1 is a comparison of the three in terms of complexity and explainability.

Reinforcement Learning (RL) is one way to train the models mentioned above. It gains knowledge about a given task by trial and error. Training a model by RL for any practical task typically involves millions of these trial and error attempts at solving the given task before a usable policy is learned. As RL is done in simulation, transferring the model from working well in simulation to working well in the real world presents another challenge. Imitation Learning (IL), another learning based approach that does not require as much compute time, can provide benefits of both RL and the pipeline; the immediate real world applicability of traditional perception-planning-control, and the onboard efficiency of running a trained model without the need for extensive training. This approach trains a policy to generate appropriate actions based on measurements of system and environment states, and what is appropriate is defined by expert demonstrations. For example, if this is to be applied to a manipulator arm, a human will operate the arm to complete desired tasks, while manual control input and sensory feedback are recorded. Once all tasks have been recorded, a policy can be trained to imitate the recorded state-action mapping.

The fault with this particular workflow is that collecting human expert demonstrations is costly, especially for the driving task. For a policy to be knowledgeable enough to maneuver a car through all possible driving scenarios, even when the task is carried out in a constrained environment such as on freeways, the amount of scenarios the expert demonstrations need to include make manual collection of human driving data highly restrictive. Its end-to-end nature also makes it difficult to explain why it takes the actions that it does. Perception, planning, and control are done by a single model and it is not clear where along the model the transitions between these tasks happen. In an effort to combat this, we can train a policy that only focuses on parts of the input that matter in a method called direct perception. Instead of building a mapping from sensor input to actuation signals, direct perception aims to map sensor input to one or more vital clues about where the agent is relative to the environment. These clues, or affordances, are then fed to a motion controller to produce actuation signals.

Contributions:

This thesis developed context-aware imitation learning and Direct Perception (DP) models for the autonomous driving task. The imitation learning model that performed the best

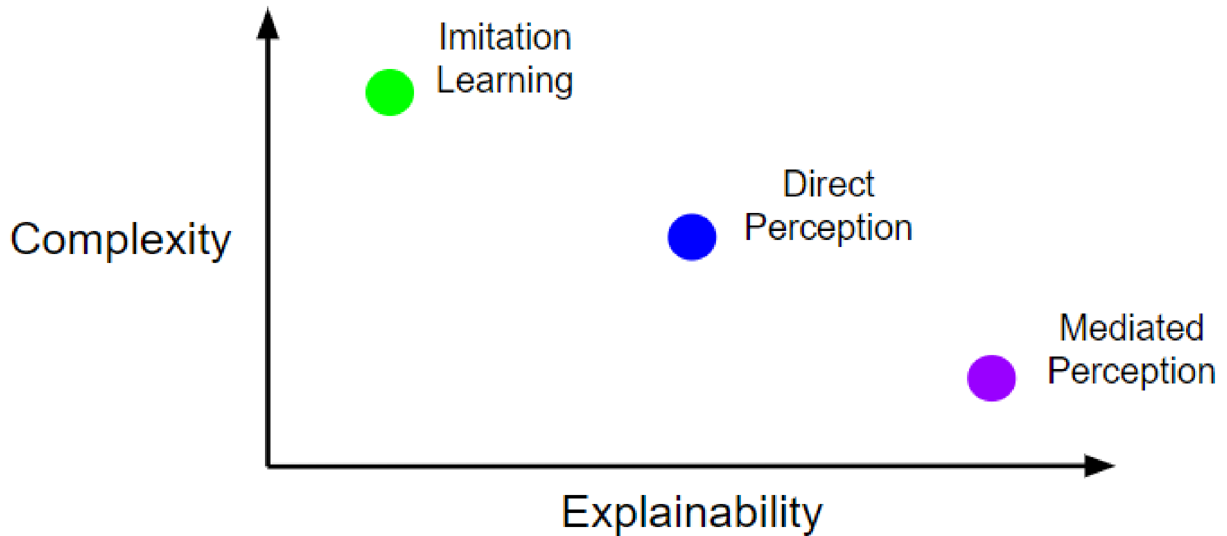


Figure 1.1: Qualitative plot that compares complexity vs. explainability for all 3 paradigms

used branching for outputting actions for different contexts, and contained a recurrent component to make use of temporal information in the input data. Instead of acting on every single sensor snapshot that comes in, it takes the change in the environment into account in its decision making process. The direct perception model is split into two parts: the affordance predictor and a controller. The predictor is the learning based component that takes in sensor stimulus and outputs affordances. It also uses branching to distinguish between different contexts. A data recorder was built to generate training data for the predictor. Lane change capability based purely on vision was developed by adding the lane change intention as a context.

Two lateral controllers based on road curvature were developed. One uses the kinematic relationship between the vehicle and the road to generate steering angles, and the other uses a dynamic vehicle model to project future vehicle path and optimizes that path to match the reference path dictated by affordances. Although both have a lane deviation feedback component, they reduce the reliance on the feedback term by generating steering angles based on road curvature and allows for more stable operation. The dynamic controller takes tire slip into account, making it more reliable at high speeds where often ignored effects in the dynamics of vehicles become significant. By using these physics derived controllers to output control actions, the behavior of the vehicle is appropriately constrained and any

errors that surface are isolated to the perception portion of the model; allowing debugging efforts to be more focused.

1.1 Relevant Works

1.1.1 Imitation Learning

Imitation learning aims to learn a desired policy, or behavior, from expert demonstrations. This behavior can be represented by anything from a linear function to a neural network, and this representation is selected based on the task to be performed and the level of abstraction of the task[45]. IL has been experimented with in the context of autonomous driving since 1989[47], when it was used to drive the Autonomous Land Vehicle In a Neural Network (ALVINN) of CMU. ALVINN used a 3 layer Fully Connected (FC) neural network to store its policy. The network consumes single channel image and laser rangefinder data that has been flattened to an input size of 1217x1, and predicts road curvature as well as a boolean on whether the drivable area is lighter or darker than non-drivable area in the current image. At the time, ALVINN researchers already realized that having a wide variety of data is key to successfully training a policy that's reliable regardless of driving conditions. To circumvent the difficulty of collecting a large enough dataset in the real world, they built a data generator that created simulated images and rangefinder readings for training. The vehicle was able to drive on sunny days down single lane roads at a speed of half a meter per second.

Muller, Urs, et al. trained a similar policy that converted image inputs to steering angles[44]. They built upon the work of ALVINN researchers by using a convolutional neural network. The application was to drive an RC car through off road courses, which inherently has larger scenario variances compared to on road driving. The network uses a pair of images from a stereo camera to make steering predictions. According to the paper, fitting the network to output steering angles from stereo images allows the network to replace handcrafted heuristics. Feature design and camera calibration are no longer needed. It has the added benefit of potentially performing better than pure stereo matching by making use of cues other than stereo disparity in the input. The authors also mention the importance of having large amounts data that show enough diversity in conditions. During data collection, the authors took care to show consistent behavior when avoiding obstacles, and also refrained from making turns when no obstacles were present. The vehicle was able to navigate itself through unseen paths of varying conditions.

[40, 46, 27] are more recent works that have applied imitation learning to the driving task. [40]built a racing policy that converts front facing images to driving actions in the form of probabilities for 36 discrete steering angles, and 3 discrete longitudinal control modes (full gas, full brake, coast). Expert demonstrations was collected using the in game autopilot of Assetto Corsa, a racing simulator. By analyzing the actions of this demonstrator, it can be seen that while the steering histogram resembles a normal distribution, throttle and brake were massively unbalanced. Despite the unbalanced data, the policy has shown performant driving when deployed back to the same vehicle in game. However, it does not work well when it was given a different car to drive. [46] used Model Predictive Control (MPC) as the expert demonstrator to solve the racing task. [27] combined IL and RL for training. Reward shaping was used to improve the performance of an IL trained racing policy. The reward was provided by a separate network that was trained on a set of images that was human labeled with a reward according to agent states including speed, orientation relative to the road, etc.

While the above 3 experiments trained policies for a racing task where the objective is to achieve the fastest lap times, there have also been studies on applying IL to on road driving [9, 6, 29, 39]. This task is more complex than racing in the sense that the policy could see a much larger diversity in the scenarios it experiences. [9] used a simple Convolutional Neural Network (CNN) to map the image from a front facing camera to the an inverse turning radius. This policy showed good performance when deployed for highway driving. [6] used a Long Short Term Memory (LSTM) network to fit sequences of images to steering angles. It was not tested on a real vehicle. However, the ability for an LSTM based network to obtain temporal relationships between each frame allowed it to perform well on a test dataset. One of the challenges in imitating steering behavior is that in real life driving, steering is held straight a majority of the time. Especially true for highways, this means strategic sampling is needed to make sure the network sees enough examples of turning. [29, 39] employed dataset trimming to remove overrepresented steering data before training. Although this method provides a more balanced dataset, many datapoints with near zero steering angle were discarded; resulting in the network having less examples to learn from and makes it more prone to generalization errors.

As demonstrated by [44], data that show consistent behavior is useful for the network to quickly learn to imitate the behavior. However, this approach to IL is unfeasible for tasks that have multimodal action distributions for a single scenario or when it's impossible for the source of expert demonstrations to be consistent (i.e. a large amount of human demonstrators). [17] presented the concept of conditional imitation learning to get around this issue. During data collection, a High Level Command (HLC) is recorded along with

sensor inputs and control signals. The HLC is an instruction for the policy to use when it finds itself in an ambiguous situation (left, right, straight, etc.). Two ways of incorporating the HLC into the policy were tested. The first was to use it as an additional input to the neural network, and the second was to use it as a switch for a branched network. During inference, the output of the branch corresponding to the currently selected HLC would be executed. The latter has a 10% higher success rate. [25] expanded on the idea of high level commands by adding an LSTM component to a non-branched policy network. With HLC as an input, the policy network with LSTM has a 25% higher completion rate than a non-LSTM, non-branched version, although it performed worse in sharp turns. While there is no correlation between the performance and interpretability of end-to-end models, for safety critical tasks, it is important for engineers to have insight into the model's behavior. [20] made progress towards interpretable imitation learning by incorporating an attention model into their network. The attention block is a skip connection that gets element wise multiplied then summed with the output of the parallel FC + softmax layer, forming a weighted sum of the region of interest features generated by the backbone. Models trained by supervised learning, which imitation learning is a subset of, are vulnerable to adversarial attacks. [10] created a simulation infrastructure which they used to analyze adversarial attacks on imitation learning driving models. Various disturbances were placed on the sensor input image and their effects on the model's action were studied. Ways to redirect vehicles driven by imitation learning models were identified, which sets the groundwork on building robust models that can defend against these types of attacks.

1.1.2 Direct Perception

Direct perception has shown to be effective at a simplified driving task. In [14], a network was trained to infer driving affordances from the image produced by a front facing camera using only a few thousand training images. The affordance set from this study includes distance to other vehicles, heading error, and distance to each lane in a three lane road. [11] successfully made use of a variational autoencoders (VAE) [34] to perform direct perception. The proposed network has one input and two heads. Input images from a drone were compressed into a small latent representation by an encoder. A decoder is trained to reconstruct this latent representation back into the original image, and a policy network is trained to map the latent representation to a set of affordances that describe the pose of gates the drone needs to maneuver through. By using the latent representation, which is a compacted version of the input image, as the input to the policy network, this approach has the ability to generalize across different domains (lighting, background scene, etc).

[53] used real street imagery to train a model to predict a set of affordances that includes distance to intersection, heading angle, and number of lanes. Although not enough to fully operate a vehicle, it provides an additional source of information to supplement information from an HD map. On a vehicle that relies on HD map to drive, it can be used as a redundant safety system.

In direct perception, the problem of multimodal scenarios still has to be dealt with. [50] builds upon the branched architecture of [17] to perform conditional direct perception. Each branch outputs a set of affordances. Branch selection is also done with respect to HLC, and the selected affordances are sent to a controller to calculate the control signals. [15] also created a conditional model, however the affordances predicted are future waypoints. It was achieved in a two stage process. First, a privileged agent with access to ground truth was trained to perform the prediction task, then it was used to provide on-policy supervision to train an unprivileged agent that only uses the image from a front camera. Another study that defined waypoints as affordances was [7]. A post-perception 2D representation of the surrounding environment was used to predict future waypoints. Perception results of the roadmap, traffic signs, obstacles, and planned route are rendered into pixel format in top down view, then these images are fed into a feature net to obtain a latent representation. An agent Recurrent Neural Network (RNN) uses this feature set, along with previous agent state and previous agent heatmap, to predict the next state and next heatmap. Note that the output is future waypoints, and no control or perception is involved. This body of work can be though of as learning based path planning, where a ideal path is given and a drivable path that takes road obstacles and traffic signs into account is generated.

1.1.3 Training Strategies

One of the downsides of supervised learning for behavior cloning is the covariate shift problem. What a policy experiences during training cannot possibly cover all possible scenarios it might encounter during deployment. In other words, the training data distribution is not the same as that of scenes in deployment. Given that CNNs are also prone to generalization errors for small changes in input [4], and small changes between training images and real life images are common (e.g. the color of a pedestrian's shirt, the outline of a vehicle), there will be inference errors at every time step. If this error is allowed to stack up over time, the vehicle can reach an unexplored area of the state space. Then, the vehicle will be acting on undefined behavior which can be catastrophic. There are various attempts at dealing with this problem. [9] applied extensive data augmentation to the training data so the policy has

a better chance at generalizing [51]. The augmentation was done by geometric shifts to the images and offsets to the associated turning radii. Aside from a front facing camera, two additional cameras that were yawed slightly to the left and right were also used in data collection. The turning radii attached to frames from those cameras were given an offset, emulating corrective behavior. Aside from geometric shifts, color space transforms can also be used [5]. Changes in brightness, shade, and pixel wise dropout can simulate real world phenomenons such as sensor occlusion and sun glare.

Augmentation techniques developed in the medical field are also suitable for autonomous driving tasks, as the types of augmentation have safety critical implications. [23] proposed a method using GANs to synthesize medical images which could possibly do well in generating driving scenes. [55] discussed an algorithm that could learn the best augmentations to use, a useful tool for the driving task where selecting augmentations is already a challenge.

There are also online approaches to combat covariate shift. [48] introduced the DAgger algorithm, which works to alter data distribution as needed. An initial dataset is collected and used to initialize a policy. This policy is then partially deployed back to the agent for further data collection. The policy is trusted until the agent is about to fail, and at that point, an expert policy intervenes until the agent is back in an acceptable state. The dataset collected in this manner is used to update the initial policy, and this process is repeated until a satisfactory policy is obtained. This was applied to the driving task and compared with offline or standard IL in [46], with an MPC serving as the algorithmic expert. This process "patches the holes" of earlier datasets by guiding the policy in areas where it does not know how to react, or does not perform well. It also enforces behavior that is already good, since parts of the new dataset comes from earlier iterations of the itself. The online IL proved to be more reliable, as occasionally using expert actions led the vehicle to a previously unexplored area. The task was running a high speed RC car on a dirt track. Due to this environment, ground truth motion was more stochastic and plant dynamics was more nonlinear than typical robotics tasks. This speaks to the robustness of the DAgger approach. With standard DAgger, there is the downside of query efficiency. Querying the expert, whether it be algorithmic or human, is costly. When the expert does not have feedback for their actions, their performance can also degrade. [58, 33] Have proposed ways to mitigate this problem by using multiple policies at once.

Learning from imbalanced data is still an intense area of research [36], and is a challenge that needs to be overcome for this thesis. Under day to day driving, the steering angle of a vehicle is held at near zero most of the time. Data recorded in this manner is highly imbalanced towards small steering angles. If learning algorithm that is naive towards data

imbalance is directly applied to this type of data, it will only learn to optimize for labels that appear with high frequency and never learn to turn. [9, 29, 39] employed data preprocessing to remove the center steering bias. The collected dataset was clipped to have a more even dsitribution before it was trained on. [17] used no preprocessing, but rather obtained an even distribution by limiting bin count during batch sampling. Both approaches perform well. However, they do not address the issue of multiple imbalanced dimensions.

1.2 Tools and Methodologies

section 1.1 provided a high level overview of the problem this thesis attempts to solve, to build policies for sensorimotor control by either end to end imitation learning or direct perception. This section serves as an introduction to some of the specific tools used to accomplish learning based policy creation, as well as how they are used in this thesis.

1.2.1 Deep Learning Frameworks

Keeping track of so many weights and biases in large neural networks is not easy, so deep learning frameworks are used to make it more manageable. Keras and Pytorch were the two frameworks used in this study. These frameworks are libraries that speed up training and deployment of neural networks with a wide range of utility features and hardware acceleration. When using these frameworks, the user defines the architecture of the network by specifying the the types of layers, sizes of the layers, and how data flows through these layers. With this capability, prototyping different neural network architectures becomes much simpler.

When it comes to prototyping and fast iteration, Keras is one of the best suited tools. Initially a front end API for multiple deep learning frameworks, Google's TensorFlow machine learning framework has officially integrated it as its high level interface. Networks are defined by stacking layers on top of each other, and training is done in one line of code. Keras also has more support than PyTorch for running on embedded devices, which is relevant to this study as deploying trained models to RC cars can be a good way to validate the models. Keras was used to create and train the non conditional models discussed in section 2.1.

The author attempted to use Keras to train the conditional models from section 2.2 as well. However, with many of its features streamlined, it did not have the level of flexibility needed to support the conditional network training workflow, therefore PyTorch was used

instead. PyTorch is a low level framework that allows users to work with array expressions, which makes it suitable for unconventional training; although it is not as user friendly as Keras. At its core, PyTorch is a multidimensional array computation library with provisions for machine learning. Data is held in containers called Torch tensors, which are similar to Numpy arrays. To create a model, users would create a class that inherits PyTorch's base module class, initialize weights, then use a class method to define operations using helper functions, array expressions, and even pure Python. The training loop, including forward propagation, loss calculation, and backpropagation, must be implemented by the user.

1.2.2 Simulation

When working on software that controls physical systems, it is important to validate the software in simulation before deploying it to the real world where there can be safety implications. An experimental control pipeline like the one proposed in this thesis will likely make mistakes and require adjustment and iteration before it is ready. If early development work is done on a real car, damages incurred can be costly to repair and cause weeks of downtime. Simulations allow researchers to quickly iterate on concepts that may or may not work and view the results without the consequences of testing on a real vehicle.

To further highlight the importance of simulation for safety critical intelligent systems, all of the big players in the autonomous driving industry have dedicated simulation teams [7, 30]. These high fidelity simulations provide a virtual proving ground, as well as a source of data. As of mid 2019, Waymo has driven 10 billion miles in their in house simulator. For perspective, they have only done 20 million real miles on public roads [56]. While simulated miles do not fully represent real world miles, job postings for simulation realism suggest that closing the gap between simulation and the real world is a priority. To effectively train ML models, it's best to have data of all possible scenarios, and it is prohibitive to collect data of dangerous driving or crashes in real life. Simulations can generate these high risk situations without putting humans in harm's way.

This study uses CARLA [21], an Unreal engine based simulator built to support research in autonomous driving systems to collect training data. There are other similar tools available, however they do not offer the level of flexibility that CARLA does. The simulator uses a client-server model to access user commands. The user would create Python/C++ clients that call the CARLA API to control the simulation. Aspects of the simulated world that can be controlled include vehicles (ego and others), pedestrians, traffic light states, time of day, and weather. Sensors can be attached to any point on any controlled agent.

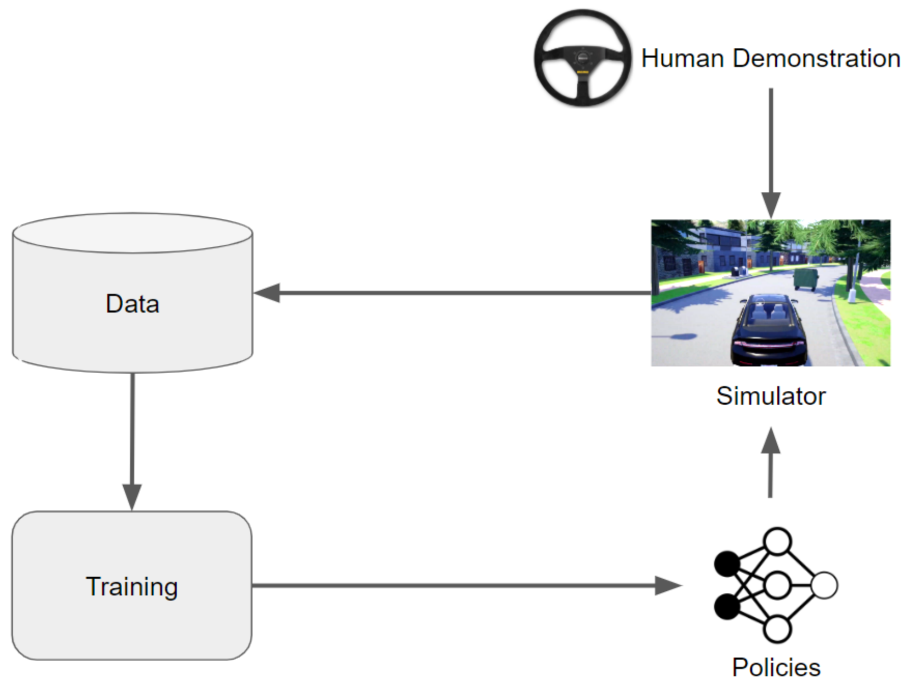


Figure 1.2: High level workflow in policy creation

This highly configurable simulation makes for an excellent testing platform for this study. Policies can be deployed back into a variety of different conditions to verify for robustness. Fig. 1.2 shows the different ways the aforementioned tools can be used to create driving policies. Data can be generated by human demonstration or by running a trained policy in the simulator. Then training takes place using the obtained data.

Chapter 2

Imitation Learning

Imitation learning aims to produce a state-action mapping, or policy, for the operation of an intelligent agent. For a self driving car, this means a policy that converts sensory input such as data from cameras, lidars, and radars, to vehicle actuation signals for steering angle and wheel torque. In the engineering spirit of achieving desired results with the least amount of resources, I will challenge myself to only use the information from a single onboard camera. The goal of this thesis in terms of IL is to produce a policy in the form of (2.1) with the following criteria:

- Uses a single camera and forward speed
- Adheres to high level commands
- Avoids pedestrians and other vehicles

$$\pi(o, c) = \hat{a} \tag{2.1}$$

The dataset used for this study is the Conditional Imitation Learning Dataset [17]. It contains sequences of 88x200 images, and each image is paired with vehicle states and user inputs recorded at the time at which the image was taken, including HLC. There are over 600,000 images, and the dataset is 24GB in total. It was recorded in a single lane road suburban environment with a variety of different vehicles and pedestrians present, and there is dynamic weather which includes rain, sun, and overcast. The IL policy π this study aims to develop is a neural network that uses the image as observation o, the HLC as context

c, and outputs driving signals as predicted action $\hat{a}$. This can then be formulated as a supervised learning problem where the optimization goal is to minimize the loss function $\mathcal{L}$ as shown in equation (2.2).

$$min \sum_{i}^{n} \mathcal{L}(\pi(o_i, c_i), a_i) \tag{2.2}$$

The optimization would be solved by gradient descent in the batch form, where a gradient is calculated for each datapoint i in a batch of n datapoints. The batch of gradients are then averaged to obtain the final gradient to be applied to the network's weights. Ideally, the process alters the policy π such that its prediction for each input pair (o, c) has minimal difference from the label a.

A fully functional self driving car requires perception, planning, and control. The policy being explored by this thesis covers perception and control. Planning is assumed to be abstracted away into a separate module. This can be onboard or cloud based, and it would communicate the result of street level path planning (think directions from Google Maps) in the form of aforementioned context c.

As modern vehicles have complex systems to manage energy usage, a policy that directly outputs throttle and brake percentages is likely not practical for production. With the current state of technology, if the goal is to develop a vision based driving solution for production vehicles, the policy should output values that can be used by low level controllers that are present in modern vehicles. These controllers would then modulate the brake and throttle based on the setpoint provided by the policy. However, this study is meant to be a proof of concept of an entirely end to end policy, to explore the feasibility of sensorimotor control architectures for robots in general.

2.1 Non Branched Conditional Network

2.1.1 Introduction

As neural networks are universal function approximators, the initial attempt at creating a policy described by (2.1) is to let a neural network fuse the information from o and c together, then process the fused information into control signals for the vehicle. A CNN feature extractor distills the image down to a feature vector that contains key information

about the scene. The longitudinal velocity of the vehicle is fed to a FC feature extractor to produce another feature vector representing the vehicle's longitudinal motion. The HLC, or context, is passed in as a one-hot encoding, as shown in table Table 2.1. These three vectors are concatenated, then passed through several more FC layers to calculate the predicted action, thus achieving end to end control. Even though this is a modular network with individual components for each input and output, and it performs both perception and control, it cannot be determined which parts of the network handle the perception task and which parts handle the control task.

As CNNs are highly adept at extracting features from images, they are used to process input images. At each convolution stage, kernels are passed through the image to uncover salient regions of the image. As training progresses, kernel values of the filters are updated and each filter learns to identify specific clues present in the image. Earlier convolution layers will focus on more abstract concepts such as colors and geometries, and later convolutions will learn to identify task specific items (Figure 2.1). In this case, they might become vehicle detectors or intersection detectors. When the filtered channels reach the last layer, they no longer resemble the original image, but are latent activation maps with internally learned meaning for each pixel. These channels are then flattened for downstream FC layers in the network.

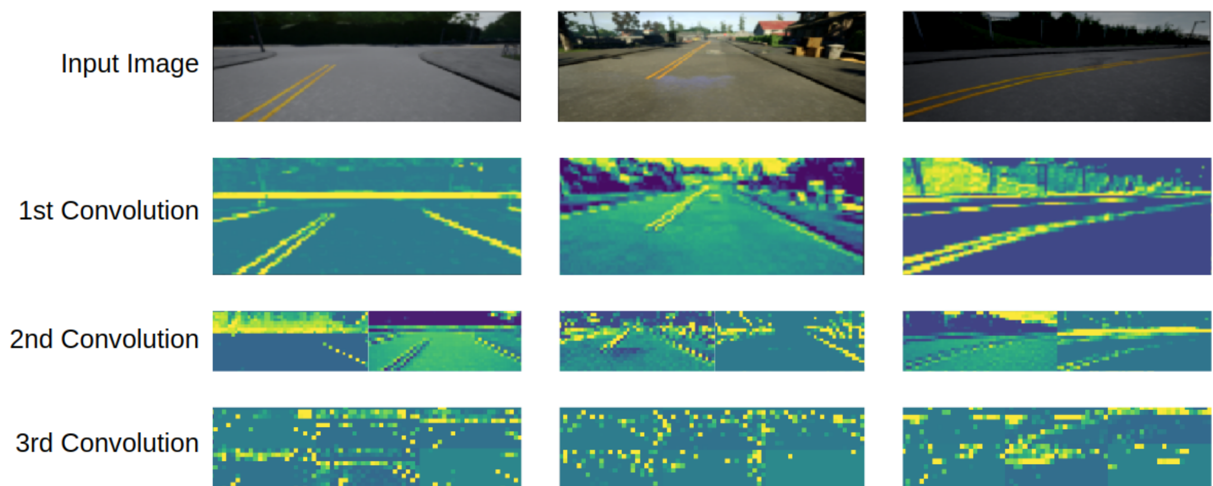


Figure 2.1: Input images to the network and saliency maps from select channels in different stages of convolution. It can be seen that each channel focuses on different things, e.g. some detect lane markings and some detect obstacles

In order for the policy to successfully navigate the vehicle on roadways, there has to be a way to dictate which turn to take when it arrives at an intersection. Without context, this decision would be unconstrained because the dataset contains demonstrations of all possible high level actions. Mixture density networks have the capability to learn multimodal data well, however, they can only return the learned distribution; there is no way to tell the network to make a turn or a lane change. How does the policy choose when there are multiple modalities in the scene? (Figure 2.2)

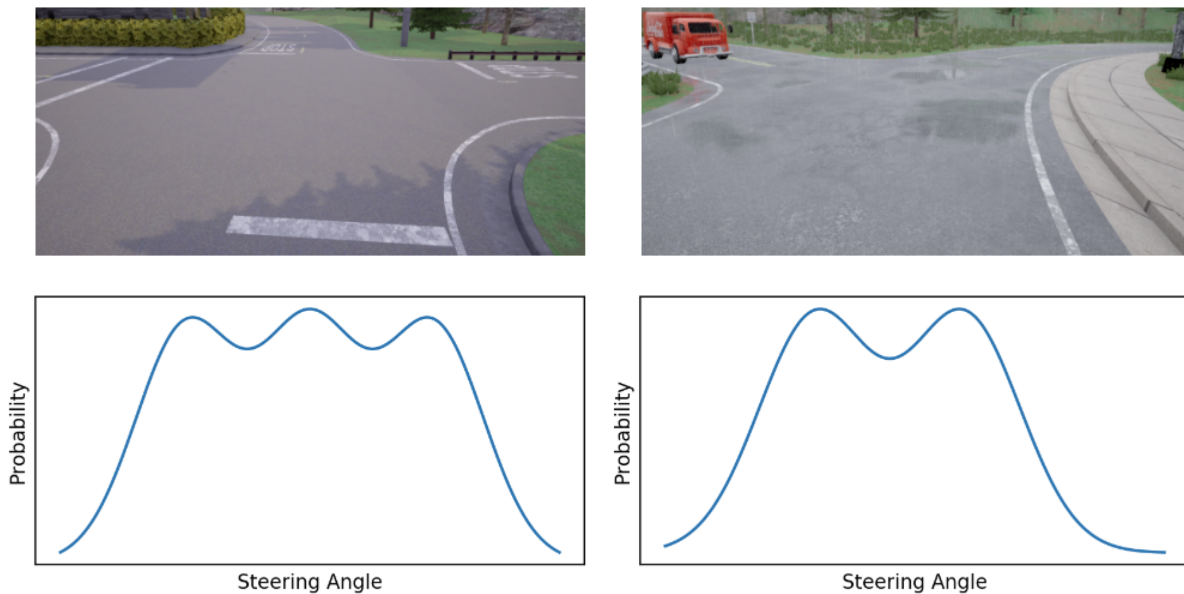


Figure 2.2: Steering probability distribution at intersections. This thesis aims to train a policy that allows the user to select which "peak" to use, or which turn to take

2.1.2 Context Usage

Even though the action distribution for each scene may be continuous, the context used to make the mode selection is not. As a neural network can only consume numerical data, context has to be in numerical form. Using integer values to represent each discrete context would be inappropriate as these contexts cannot be described by a continuous scale. Although discrete, integers encode more information than suitable for this task due to their

Table 2.1: The high level commands used as contexts and their corresponding one-hot form

High Level Command	One-hot Representation
Straight	$[1, 0, 0, 0]$
Right Turn	$[0, 1, 0, 0]$
Left Turn	$[0, 0, 1, 0]$
Follow Lane	$[0, 0, 0, 1]$

ordinal nature. One-hot encoding however, is a great fit as it does not weigh one category over another and there is no ordinal relationship between any of the encodings. Therefore, the HLC for each datapoint is one-hot encoded before being fed to the network.

To formally define the usage of HLCs, they are *instructions* for high level decision making. They have no control over physical vehicle motion, which is handled purely by the policy itself. If a vehicle is traveling down a road with no turns, it will continue down the road despite being issued a turning HLC. It will turn if and only if a turn is available, and a turning HLC is being issued. This means a turning HLC can be issued well ahead of the vehicle reaching a turn, as illustrated in figure Figure 2.3. The color coded path shows the driver intention, or HLC, at different points on the road. The intent to make a turn is what the imitation learning model needs to be aware of.

2.1.3 Data Balancing

The steering distribution in the dataset is highly concentrated at zero (Figure 2.4). If a network was trained on this data without any balancing, it would predict near zero values for any situation as it would have seen so few examples of turning during training. This is a challenge in most cases of learning from recorded driving data. With the way most road networks are designed, the amount of times when the steering wheel is significantly turned is disproportionately less than when it is held straight. Some have chosen to trim their dataset to more even distributions [29, 39], and some have chosen to augment their dataset by artificially shifting steering angles [9, 26]. The author believes that the trimming method has the disadvantage of stripping away a large amount of useful information from the dataset. Once trimmed, although the distribution is now tractable, many examples of small steering angle driving has been removed from the dataset, and the neural network has less data to generalize from.

If trimming comes with the downside of less generalization power, is there a method that

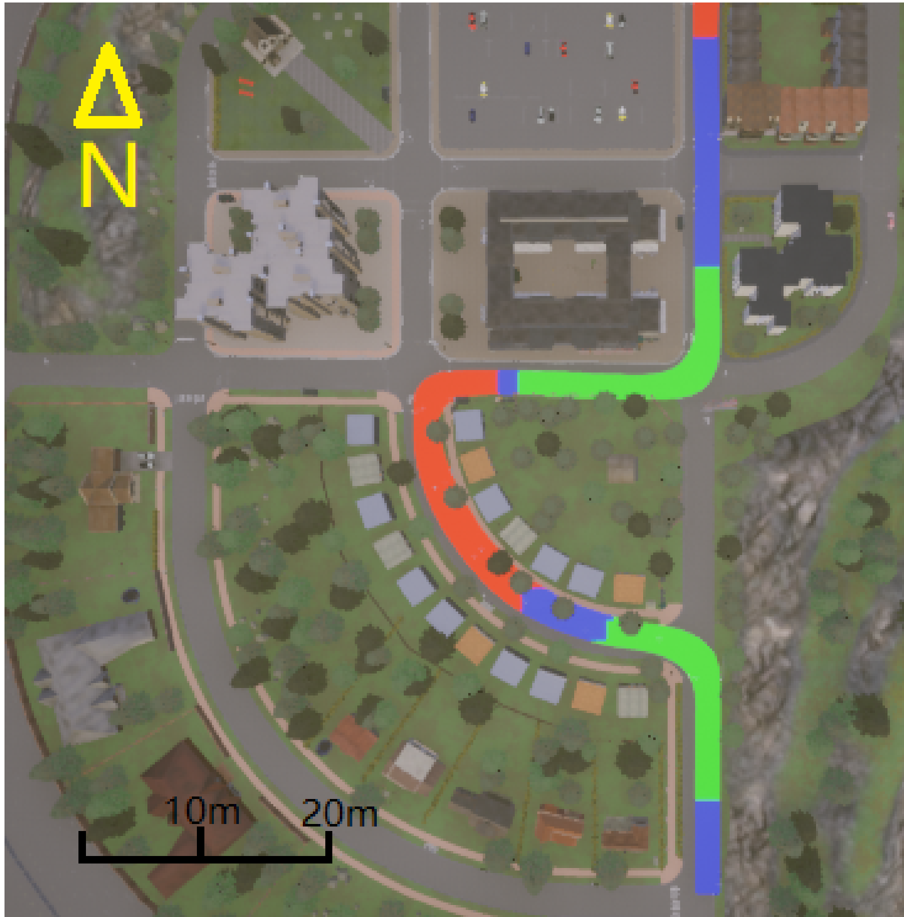


Figure 2.3: Path of a vehicle traveling northbound (up), color coded by driver intention. Blue means follow lane, green means left turn, red means right turn

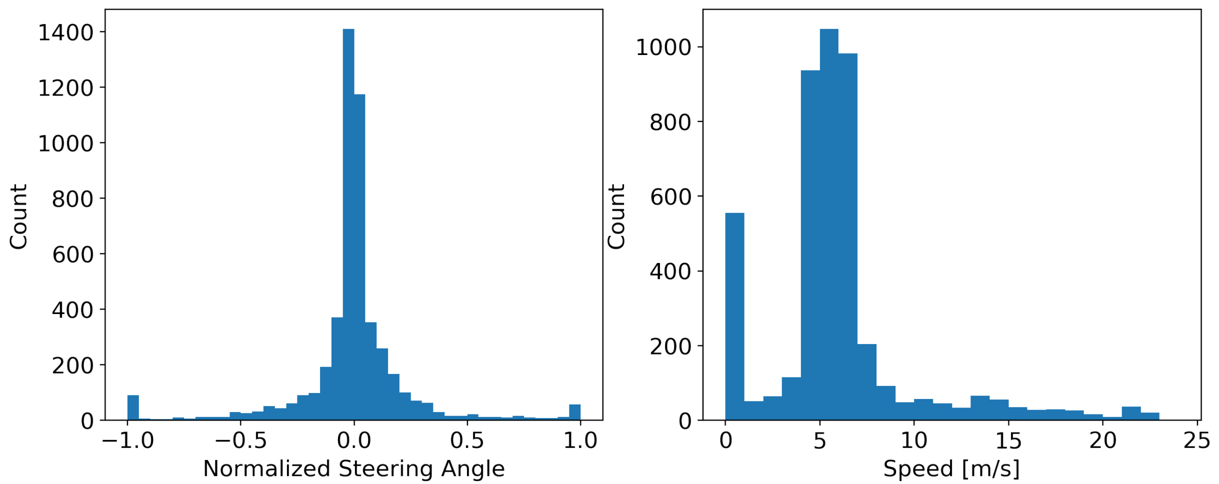


Figure 2.4: Steering and speed distributions of 5000 randomly sampled datapoints from the Conditional Imitation Learning dataset. Directly applying gradient descent to unbalanced data like this would only teach the network to output the most common values

allows us to not have to trim, while providing balanced data? Since training the network is done using batches of data at a time instead of the entire dataset at once, we can alter the distribution of each batch while retaining the entire dataset. A histogram is initialized and a threshold is set based on batch size. Datapoints are randomly sampled, and each one is added to the appropriate bin in the histogram. Once a bin reaches max capacity as per the threshold set earlier, newly sampled datapoints that belong to that bin are rejected. This binning process continues until the amount of datapoints that have been successfully binned equals the batch size.

With the bin threshold in place, each bin is limited in capacity, which results in a steering distribution that is no longer highly concentrated around zero. Figure 2.5 shows the balanced data distribution at different levels of thresholding. If the threshold is set too high, the effect of this sampling method would become negligible. If it is set too low, the sampling algorithm would take an unreasonable amount of time in sampling for datapoints that do not belong to the highly frequented bins. It was also found that continuing to lower the threshold past 12 percent does not lead to better training results. The dataset has artificially injected steering noise in an effort to train the network to have recovery behavior against veering off lane. Lowering the threshold too much means much of what is collected in each training batch is this artificial noise, decreasing the quality of data. Another parameter that requires some care in tuning is the width of the bins. It needs to be set such

Algorithm 1: Bin Counter

input : $batchSize, threshold, binUpperbound, binLowerbound, binDelta$
output: A balanced batch of data for training $batch$
initialize counter object bin with $binUpperbound$, $binLowerbound$, and $binDelta$;
initialize batch array $batch$ with $batchSize$;
for $i \leftarrow 1$ **to** $batchSize$ **do**
    randomly sample datapoint ds from database;
    $fullFlag \leftarrow bin$.checkIfFull(ds);
    while $fullFlag$ *is not true* **do**
        randomly sample datapoint ds from database;
        $fullFlag \leftarrow bin$.checkIfFull(ds);
    end
    bin.addCount(ds);
    $batch$[i] $\leftarrow ds$;
end

that the bin resolution is fine enough to show trends in the distribution. When bins are too wide, there is the risk of binning data that is only on one side of the bin.

2.1.4 CNN Bias Correction

As shown in figure Figure 2.4, vehicle longitudinal speed in the dataset has a long tail distribution. This is due to the data being collected in urban scenarios where the opportunity to drive safely at high speeds is rare. We could balance the data using the bin counter method described above, but since we are already balancing steering, adding an additional dimension to the bin counter would make training too slow. This is due to the GPU becoming underutilized as the sampling process on the CPU bottlenecks training.

Longitudinal speed is also an input to the network. Combined with the two dimensional balancing challenge mentioned above, it makes an enticing case to do nothing about the speed distribution. However, speed is given in the form of a single value, and the input image is tens of thousands of values! Speed information can easily get dwarfed among all the other numbers being crunched by the network. It could be difficult for the gradient descent process to settle on a set of weights that recognize the importance of that one

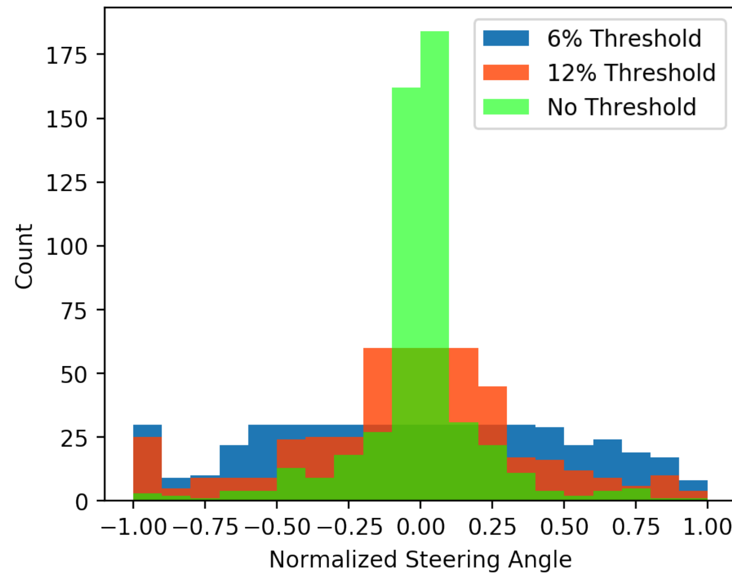


Figure 2.5: Steering distribution of three batches of 512 datapoints obtained with the bin counter, each with different levels of thresholding

input, which describes longitudinal motion. To get around this issue, we can force the CNN to infer speed from the image as well. This way, the policy can use both the CNN encoded vector and the speed encoded vector as a source of information for longitudinal motion.

To achieve this, a speed predictor is branched out from the CNN output vector. This module is trained to predict the current longitudinal speed, which is the same as one of the inputs to the network. The loss from this branch is weighted, then summed with the main loss. Notice that the speed decoder only uses the CNN output (Figure 2.6), and there is no flow of information from input speed to predicted speed.

2.1.5 Image Augmentation

A policy that controls a vehicle based on RGB image inputs must be robust to unseen situations. This means it must generalize well, and training with a high variety of data is the most straightforward way to help a network generalize. However, it is difficult to curate unbiased driving data that has a wide enough variety of situations, as things out of the ordinary simply does not happen often in real life driving. To solve this problem,

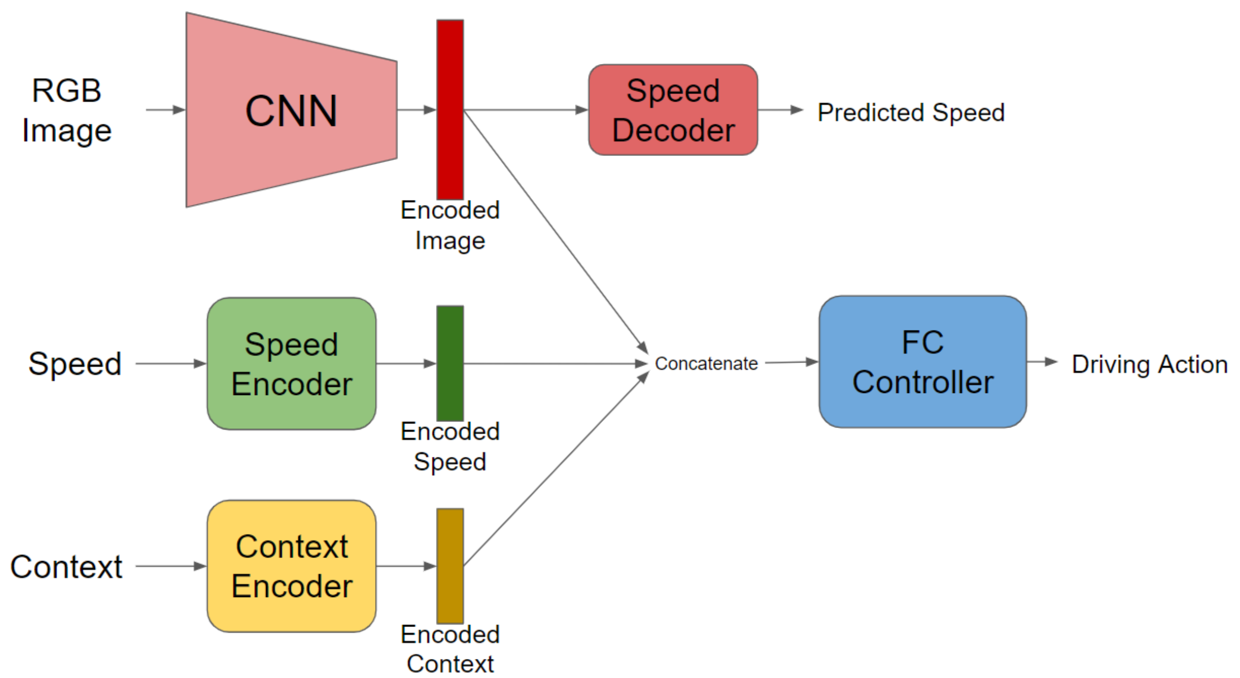


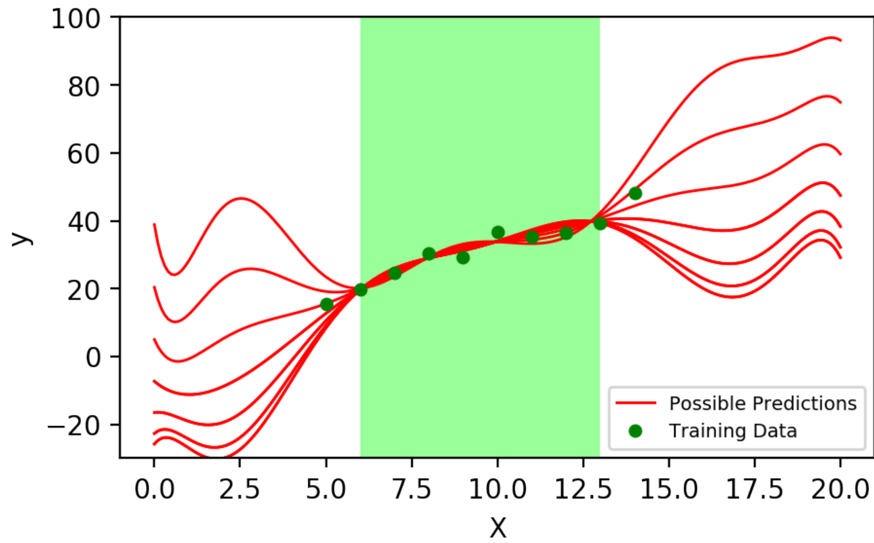
Figure 2.6: Non branched conditional imitation learning network

we can extend the bin counter balancing method outlined subsection 2.1.3 by designing a sophisticated labeling and querying system to generate training data with high situational variety. [3] describes such an approach.

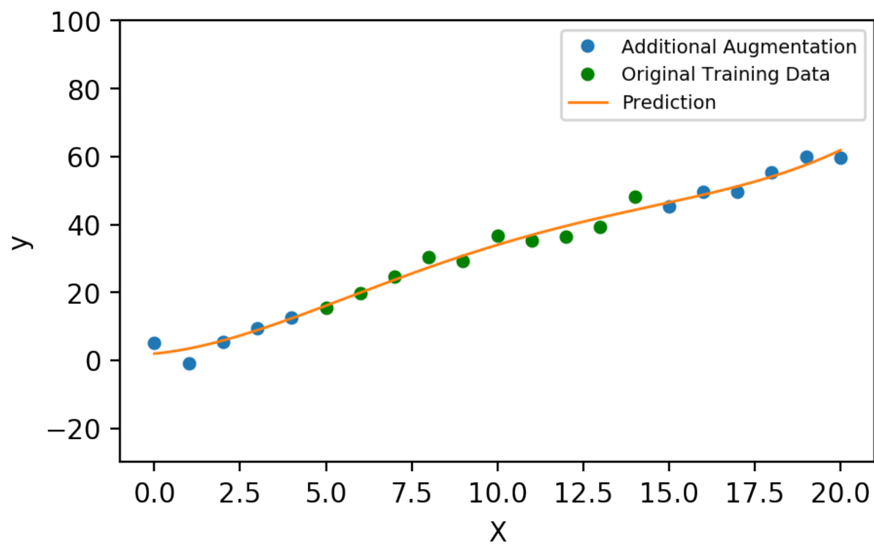
To achieve adequate generalization without the support of an engineering team, data augmentation can be used. This is a technique for making a dataset seem larger than it really is. For augmenting images, typical operations are geometric shifts, changes in shade, blurring, and brightness. Since neural networks rely on recognizing patterns and trends in the data to learn, these augmentations cannot infringe on the relationship between input and output. For example, the images cannot be flipped or rotated in any way if there are key clues in the way objects are oriented in the image. If done appropriately, augmentations can give an image dataset more diversity and thus helps the network to generalize better.

The training images were augmented by changing shade of color and brightness to account for varying lighting conditions. Pixel dropout and additive Gaussian blur were also used to model sensor failures. Even though some images turn out to have extreme shades of pink or blue that would probably never represent a real life lighting situation (Fig. 2.8 (b)), applying these severe augmentations actually leads to lower validation losses than only augmenting the images to look like what we think can be picked up by an onboard camera (less extreme shading, Fig. 2.8 (a)). The explanation behind this lies in the way a neural network fits itself to data. After training, the network will make appropriate predictions on data it has seen; but on data it has not seen, it cannot be proved that it will behave well. Training data belongs to a range on the numerical axis and training a network on this data is the equivalent of constraining its output for this range only. There are no constraints at all for input values outside of this range, so predicted outputs for "out of range" inputs can wildly differ from the ideal policy. This effect is demonstrated by Fig. . In other words, the network could overfit itself to only a certain range of data. Even though the network might not encounter these types of inputs during inference, this type of overfitting can have a negative effect on "in range" inputs as well. As seen in Fig. 2.7 (a), there are fluctuations even for the "in range" portion of the predictions.

By augmenting the inputs beyond what we can expect in real life, the network learns to stay on trend for a wider range of values (Fig. 2.7 (b)) and will provide a smoother fit for the in range data as well.



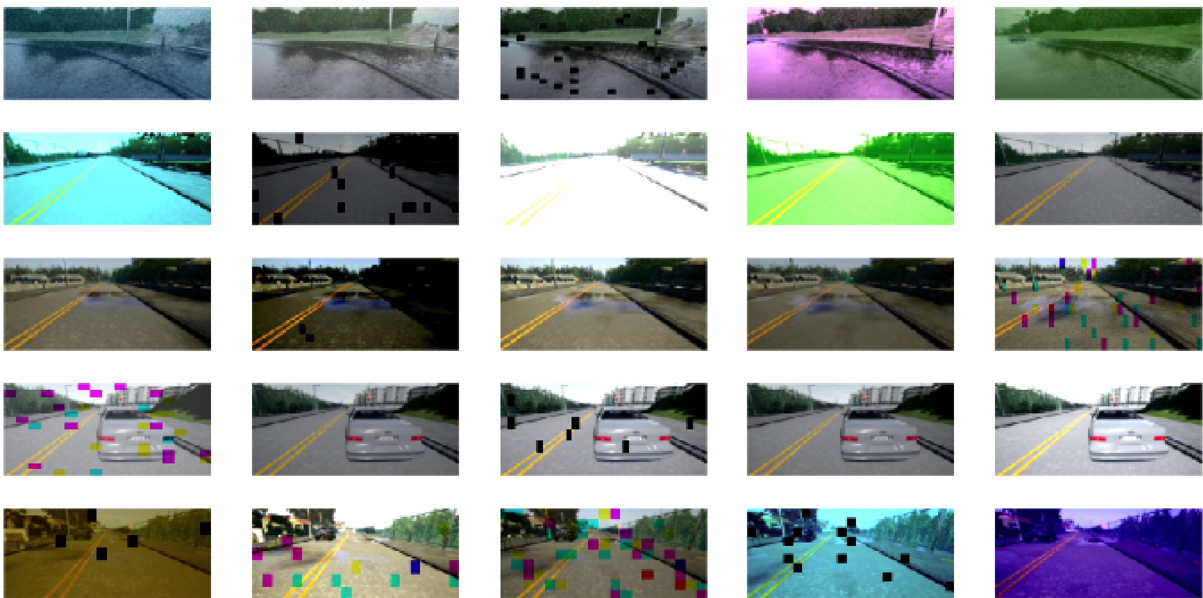
(a)



(b)

Figure 2.7: (a) Not enough augmentation can lead to overfitting and worse performance within original data range (green shaded portion) (b) Adequate augmentation covering a wider numerical range, better fit within original data range

(a)



(b)

Figure 2.8: Images from the dataset after augmentation. (a) shows examples of slight augmentation (b) shows examples of severe augmentation, which provided less validation loss

2.1.6 Production Considerations

Of course, a model trained purely on simulation data could not be trusted with real human lives. CNNs are sensitive to small fluctuations in input [4]; and even though simulations are getting increasingly more life like, there is still a gap between simulation and the real world that needs to be addressed. Before the policy gets deployed to a real vehicle, the knowledge it learned from the simulation domain has to be transferred to the real world domain to improve reliability.

There are several ways to make this knowledge transfer, the simplest and oldest of which is fine tuning. Fine tuning is the process of training a neural network with a small, production representative dataset that has already been trained on a larger dataset. Only weights from the last few layers are updated, with the remaining layers frozen, during fine tuning. This is done to preserve the general, abstract concepts that apply to both datasets. Fine tuning is typically used when there is little production representative data, but large amounts of data from another similar task.

Another approach is to learn the mapping from one domain to another using an encoder-decoder structure [11]. Once the mapping network is trained, it is used in conjunction with the policy network trained on simulation data. The mapping network converts real world images to the simulation domain, which is then used by the policy network to make predictions. The drawback of this approach is having to maintain an additional network. However there is no performance lost in the domain transfer since the pipeline will perform at the same level as seen in policy training validation, assuming the mapping network provides an accurate simulation domain reconstruction.

2.2 Branched Conditional Network

2.2.1 Introduction

As mentioned in subsection 2.1.1, this policy is being trained to complete a multimodal task. It has to decipher the multimodal nature of the environment, as well as that of possible control actions. Even though neural networks are excellent function approximators, training is more straightforward when they are not presented with difficult tasks such as multimodal learning. Breaking the task into unimodal subtasks might bring better performance.

Each of the HLCs used in the non branched model typically represents a peak in the action distribution. When there are no turns available, the peaks are aligned, forming a single Gaussian like distribution. When the vehicle is at an intersection, the peaks will diverge to represent the different routes available. For simplicity, this study assumes the vehicle will only ever encounter roads that split into three or less other roads. This covers T intersections, Y intersections, highway on/off ramps, roundabouts, and many other unique scenarios a vehicle might encounter; and can easily be expanded to cover even more niche intersection types if sufficient data was available.

To decompose the overall sensorimotor task down to unimodal subtasks, the network will have multiple control branches at the end, each corresponding to a specific HLC. This means each control branch will not have to comprehend the multiple routes available in the input scene. The straight/follow branches will always output actions that drive the car along the current road, and the turning branches will do the same until there is a turn available; at which point they will take the turn.

2.2.2 How Branching Works

The portion of the network before branching, or the trunk, are similar to the branched network. Non conditional portions of the network will be referred to as the trunk from here on. A CNN feature extractor processes input images to produce a input vector. An FC sensory module processes longitudinal speed to produce a motion vector. These two vectors are concatenated, then passed to the sensor fusion module before branching into the four action branches. The HLC is no longer an input to the network, but rather a switch used to select which branch to train or use. At the policy's interface level of abstraction, it works the exact same way as the non branched network. A planning module would communicate the current HLC to the policy, and the policy would output the appropriate action. The difference is in how the HLC is used to affect the output. During inference, instead of passing in the HLC's one-hot encoding to the network, the encoding becomes a switch to select which branch is used.

One of the downsides of the non branched network is sometimes it does not comply with the HLC. The branched architecture poses a "hard" request for a change in behavior when there is an HLC change, which is more reliable than the non branched version. This is due to the output branch of each HLC using a different set of weights (Fig. 2.9), weights that were trained using only state-action pairs that correspond to that HLC (more on training later); whereas the non branched network is more of a "soft" request for change in behavior.

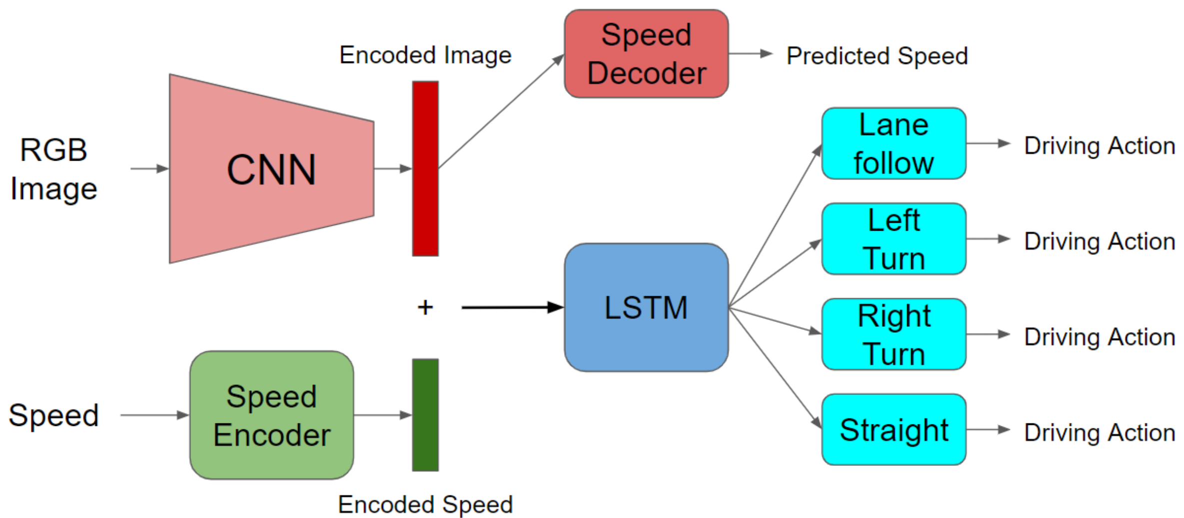


Figure 2.9: Branched conditional imitation learning network

For the non branched network, how the predicted action changes when a new HLC is given is completely up to the data and specificities of training. If the network has not learned to generalize well, it could be unresponsive to HLC changes, making it a less robust solution compared to branching.

2.2.3 Branch Training

During training of the branched network, data in the training batch dictate which branch to update. Each datapoint, based on its HLC, produces gradients for one action branch and the trunk of the network. This is done so each branch is strictly only trained by data for one type of HLC. To help understanding this process, consider a training batch with a batch size of two. One datapoint has the left turn HLC, and the other has the straight HLC. Once the batch is forward propagated, the Mean Squared Error (MSE) or Mean Average Error (MAE) loss function is applied individually with each datapoint and its corresponding branch. The loss for the left turn datapoint is calculated with the left turn branch, and the straight datapoint with the straight branch. The other branches do not incur any loss. This loss is normalized, then summed together with any unconditional output from the trunk, then backpropagated through the network, ending the training process for this batch. Weights of branches other than left turn and straight remain unchanged.

To implement this branch training process in PyTorch, it was discovered that the most straightforward way is to calculate losses between each action label and all branches, then mask off the losses from unactivated branches. The action label a is repeated height wise by the number of branches there are, and the prediction $\hat{a}_i$ from all branches are concatenated together. These two vectors are now of equal height. An element wise delta is calculated between them, which is then either squared or applied the absolute operation based on whether MSE or MAE loss is being used. The resulting vector is the element wise loss between each branch's prediction and the label in consideration. Since we only want to train one branch, we need to mask off the losses of the remaining branches. Based on HLC, a one-hot/n-hot mask m is generated to bring the losses of those branches to zero before a height wise sum is calculated. This is then normalized by the amount of labels there are in a to arrive at the branch loss.

Algorithm 2: Branch Training

Result: Train individual branches for a network that has n conditional branches and m outputs per branch

initialize network $model$;
initialize data generator $batches$;
for $batch$ in $batches$ **do**
    unpack $batch$ into input X, label y, and context c;
    $\hat{y}_{1 \to n} \leftarrow model(X)$;
    broadcast label y n times into y_{large};
    $y_{diff} \leftarrow$ element wise difference between y_{large} and $\hat{y}_{1 \to n}$;
    if *MAE Loss* **then**
        $elemLoss \leftarrow y_{diff}$;
    else if *MSE Loss* **then**
        $elemLoss \leftarrow$ square(y_{diff});
    generate one-hot or n-hot $mask$ based on context c;
    $masked \leftarrow mask \otimes elemLoss$;
    $loss \leftarrow$ sum($masked$) / m;
    $gradients \leftarrow$ backpropagate($loss$, $model$);
    $model$.update($gradients$, $learningRate$);
end

2.2.4 Temporal Learning

Adding a recurrent component to the network allows it to learn temporal dependencies. This brings the decision making process of the policy one step closer to that of a human driver. When we drive, we do not base our actions on a snapshot of the perceived environment, but rather a sequence of time from the past until the current moment. For example, seeing a leading vehicle get closer and closer, we are able to estimate how fast we are approaching that vehicle and decide on the correct amount of throttle or brake application to avoid crashing. Conversely, if we were only given a single image and asked to control our car's longitudinal motion, we would not do so well. We are able to extract valuable information from how the environment changes over time. Even though we are only able to perceive an instantaneous moment at a time, our memory of the past gives us more to work with. Consider another scenario: we are given an image taken from the perspective of the driver. The vehicle is traveling along a road with no other cars, and a speed limit sign is clearly in view. Based on this picture alone, we do have enough information to control the vehicle's longitudinal motion. This means decision making in different scenarios can require analyzing perception history of varying lengths into the past. Therefore, we need a policy container that supports the following:

- Learning temporal relationships in data
- Decision making according to memory
- Retaining and discarding memory as needed

RNN is a common network structure useful for temporal learning. Simple RNNs work by applying the same weights and biases to every timestep in the time series input, usually with the goal of predicting either the next timestep or another time series variable. This type of network maintains a hidden state that gets passed from one timestep to the next, serving as memory. However, there is no way to control how long this memory stays effective, especially in longer sequences. Gated Recurrent Unit (GRU) and LSTM however, are networks with a feature called gating to help with remembering longer sequences. GRUs have a reset gate and an update gate that use both the current timestep's input and the hidden state to modulate its hidden state, they control when information is remembered and when it is thrown away. LSTMs have a similar way of processing time series data, but it maintains an additional state called the cell state. This additional state helps it maintain a longer memory than GRUs. The benefit of GRUs is they have less tensor operations,

therefore they can be trained in faster than LSTMs. For automotive applications, training speed is less of a concern as the network would be trained in massive datacenters. For all reasons listed, LSTMs will be used in the policy network for temporal learning.

2.3 Results

The branched architecture performed better than the non branched architecture. Table 2.2 shows completion and infraction data from the two models after a hundred runs of a 1 km path in the Town03 map with dynamic weather. Town03 depicts an urban area with roads up to four lanes wide. Note that the model has not been trained on imagery from this environment.

The training data has some major differences from Town03. It only contains single lane roads, and there are no dashed lane lines. The intersections in the training data are also much smaller. If the models perform well in Town03, it would be an indication that they have generalized what they learned from the training data to larger roads. The most common infraction mode for the non branched model was disobeying HLCs. For the branched model, it was running into the side of the road. As they have not seen any dashed lines in training, crossing them was another common infraction mode, and more so for the non branched model. When driving down a two lane road, it often appeared to see the solid lines on the left and right extremities as road boundaries and straddled the dashed lane line. The branched model also crossed dashed lines albeit less frequently, and it tended to stick to the rightmost solid lane line it can see. Another noteworthy observation was the models' penchant for road shoulders. The training data did not have road shoulders, and the models might have learned to recognize curbs as road boundaries. During testing, both models would sometimes cross a solid lane line to drive along the shoulder, meaning they see curbs as having a higher priority than solid lane lines. Lastly, weather still seems to affect the models. For the exact same scenario bar different weather, the models still

Table 2.2: Testing results for trained imitation learning models

Backbone	Completion %	Minor Infraction/km	Major Infraction/km
Branched	59	5.3	1.6
Non Branched	35	5	1.1
Branched No Aug.	41	8.2	2.3
Non Branched No Aug.	19	12.9	4

perform differently. More work will need to be done on improving generalizing capabilities of these models.

Chapter 3

Data Recorder

The CARLA Conditional Imitation Learning dataset does not contain labels needed to train a direct perception network. Retroactively obtaining those labels is impossible since they are dependant on dynamic states of the simulated city. For this reason, a new dataset has to be collected to train the affordance predictor. Due to prior work on this thesis being based on the CARLA simulator, this new dataset will also come from CARLA. There are no preexisting tools to record data from recent (0.9.x) versions of CARLA, therefore one was created to generate the dataset needed to train the affordance predictor. A secondary goal for the recorder was to be an easy to use general purpose tool for ADAS/self driving researchers to collect data of any common sensor modality. Figure 3.1 shows the equipment used when recording data.

Figure 3.1: The data recording setup

3.1 Recorder Client

The CARLA simulator itself launches as a server, and users would create clients to control the simulation and visualize its contents. The recorder client created for this thesis first sets up the simulation agents, then collects user input from a steering wheel or keyboard to control the ego vehicle and saves recorded data in a main loop. The game library Pygame is used for visualization (Figure 3.6) and reading in user inputs, and the computer vision library OpenCV is used to process and save recorded images. To run the data recorder, a bash script launches the CARLA server as well as clients that handle simulation weather, NPC management, and the main recorder client itself (Figure 3.2).

Once the recorder client is up and running, it spawns the ego vehicle, and the user defined sensor configuration. Sensors can be placed anywhere on the vehicle or in a static location. The main loop of the recorder reads in user inputs, sends them to the vehicle, takes a snapshot with all the spawned sensors, calculates affordances using waypoint information, saves all relevant data, and lastly "ticks" the simulation, which propagates the simulation world to the next timestep. The tick is needed as the simulation is run in synchronous mode, meaning the simulation holds the current timestep until all processes in the recorder's main loop are complete before jumping to the next. This is to ensure all saved sensor snapshots and control actions from each timestep are time synced. The alternative to this is asynchronous mode, where the simulation and the recorder each try to run at their own specified update rates. Even if both target update rates are set to the same value, the variable execution speed of the main loop's conditional code will lead to sensor snapshots and saved labels coming from different times, and models trained from this data could learn undesirable behavior from theses artifacts.

Since we wish to try recurrent components in the affordance prediction network, setting the simulated world update rate is mainly driven by time delta between each sensor reading. For fairness of comparison, the refresh rate must allow training models that operate at the same frequency (7.5 Hz) as the imitation learning model. 15 Hz was chosen since it allows training 5, 7.5, and 15 Hz models by dilated sampling or simply using all the data. A simulation refresh rate higher than 15 Hz is unnecessary since consecutive images would be too similar and it would slow the real world refresh rate down to a point such that the recording task becomes extremely tedious for human demonstrators. The operation that brought the largest overhead in the recorder loop was writing images to disk. Initially, running the sim at 15 Hz presented a challenge for a powerful (as of early 2020) gaming desktop. After switching from Matplotlib to OpenCV for saving images, the recorder ran smoothly. Preliminary trials saved images in the .jpeg format for a small speed increase,

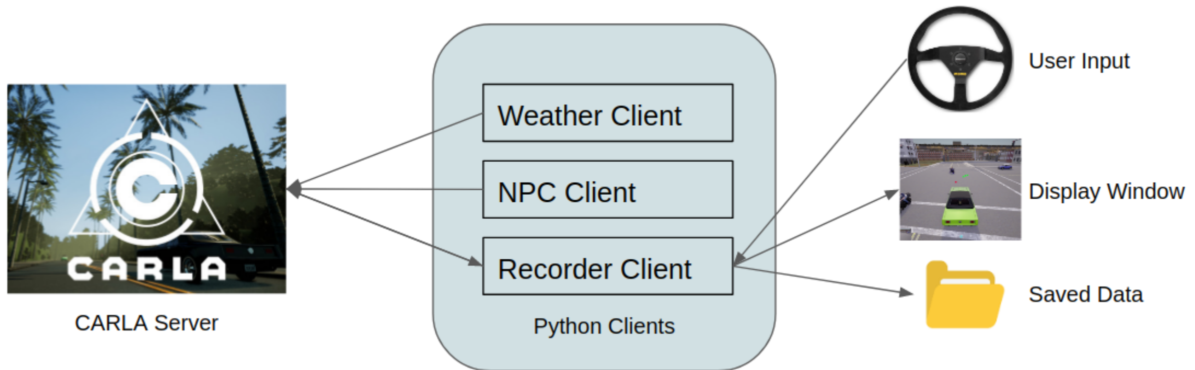


Figure 3.2: Architecture of the data recording pipeline

Table 3.1: Average real world FPS of the recorder with the simulation set to 15 FPS, when using different libraries to save images

	Real World FPS (.jpeg)	**Real World FPS (.png)**
OpenCV	18	13
Matplotlib	14	12
Pillow	12	8

and the final experiment saved images in .png since it is a lossless format. Write speed comparisons are shown in Table 3.1.

The values in Table 3.1 were obtained with a minimal sensor configuration of just a single low resolution (88x200) camera. The modern vehicle can have more than a dozen sensors of various types onboard, and using this many sensors in this recorder would make it unbearably slow; and this is not conductive to the goal of making this an all purpose data recorder for autonomy research. Python runs on a single thread by default, but there is a threading module available to make use of additional threads. A direction for future expansion is to refactor the data saving code to multithreading.

As the recorder is built to be a general purpose tool for all kinds of automotive research, it needs to be easily accessible. For the work done in this thesis, user commands were sent to the recorder client through a video game steering wheel. As not everyone has this equipment readily available, PS4 controller and keyboard support has been added to lower the barrier to entry for using the recorder. An additional feature is the switching between user driven and CARLA autopilot driven to reduce the reliance on human demonstrators during data

collection, and to facilitate the study of human/vehicle control handover scenarios [41, 54].

3.2 Labels

3.2.1 Data Format

Aside from saving sensor data, numerical and categorical labels are also saved. The recorder is configured to save motion states of the ego vehicle, driving inputs from the user, and information on vehicles within a preset radius of the ego vehicle. These labels were selected with functional breadth in mind. Aside from imitation learning and direct perception, the recorder can generate datasets that enable training of detection, classification, localization, semantic segmentation, and more. Of course, what information is saved can also be easily changed by the user.

Each time a recording session begins, a new recording directory is generated which is named the current time. Within the recording directory, there is one subdirectory for each sensor and another subdirectory for labels. Each simulation tick incurs a .json file in that subdirectory which contains all labels for that timestep. Figure 3.3 shows an example of three recordings, each with three RGB cameras and one semantic segmentation camera. Table Table 3.2 is an abbreviated list of items saved as labels. For ease of access, most items are stored at the root level of the .json file, except information on nearby vehicles and pedestrians. Under the "Nearby Vehicle Information" and "Nearby Pedestrian Information" fields, each car and pedestrian has their own entry, and their contents are shown in Table 3.3.

3.2.2 High Level Command Handling

During data recording, in addition to driving the vehicle, the human demonstrator has to relay their driving intent to the simulator, so each datapoint has an HLC just as in the CARLA Conditional Imitation Learning dataset used in the previous chapter. The stream of HLCs from the human demonstrator cannot stop, but having to constantly hold buttons makes data recording difficult, so the "follow lane" HLC is active by default until any other HLC is signaled. The other HLCs are signaled by the driver pressing assigned buttons on the steering wheel or controller (Figure 3.4). HLCs are highly correlated to the actual driving tasks of steering/pedal modulation, which the human demonstrator is doing anyways, so

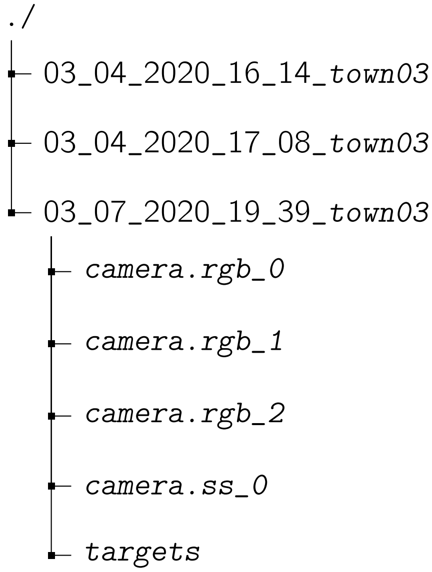


Figure 3.3: Saved data organization

Table 3.2: Key labels saved by the recorder

Label	**Unit**
Global Cartesian Coordinates	m
Global Velocities	m/s
Global Accelerations	m/s^2
Global Rotation Angles	deg
Global Rotation Rates	deg/s
Chassis (Local) Velocities	m/s
User Driving Inputs	various
Affordances	various
Nearby Vehicle Information	-
Nearby Pedestrian Information	-

Table 3.3: Labels for each nearby vehicle and pedestrian

Label	**Unit**
Global Cartesian Coordinates	m
Global Heading	deg
Distance to Ego Vehicle	m
Agent ID	-

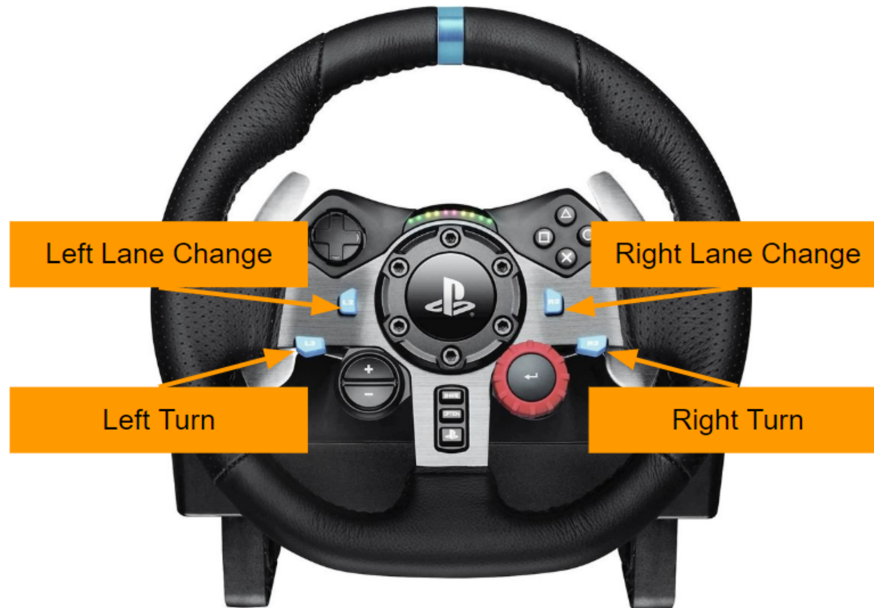


Figure 3.4: Button layout for activating different high level commands during data recording

why are the HLCs not bound to the driving tasks such that they don't have to be explicitly signaled by the human demonstrator?

The answer to this question lies in the data imbalance issue discussed in subsection 2.1.3. From the meager amount of times the steering wheel is turned by a significant amount away from zero (Figure 2.4), it can be inferred that in natural driving, there is also an imbalance in driver intention. Most of the time, we do not have the intention to turn or make lane changes. If turning and lane change HLCs are only activated during actual turns and lane changes, there would be a massive class imbalance between the default follow lane HLC and the rest. To solve this imbalance problem, as opposed to the previous chapter, there is an additional variable to work with: the data. Creating a new dataset allows control over data distribution. To eliminate data imbalance from the source, the human demonstrators are instructed to drive in ways such that each affordance label has an even distribution. Detailed instructions are as follows.

1. Avoid holding the steering wheel straight for long, steer in a sine wave down a straight lane
2. Follow vehicles at different distances
3. Intermittently activate turning/lane change HLCs even when it is not possible to carry out those actions
4. Intermittently activate lane change HLCs but do not make the lane change

Since the direct perception model does not learn the behavior from the demonstrations but rather learns the mapping from sensory stimuli to affordances, the direct perception pipeline will not reproduce the erratic driving behavior from the demonstrations. (1) flattens the heading error distribution by putting the vehicle into nonzero heading orientations respective to the road. (2) flattens the distance-to-front distribution. (3) Trains the network to be aware of when certain actions are impossible and breaks the correlation between driving tasks and HLC, which is necessary so the car only carries out an action when it is possible to do so. (4) flattens the crosstrack error distribution by having the vehicle drive along at nonzero crosstrack errors.

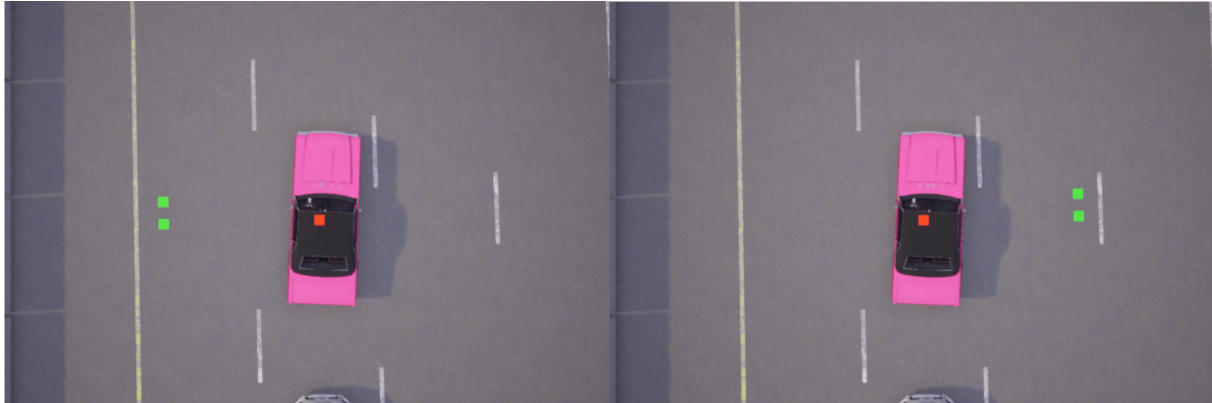
The conditional affordances are calculated using the vehicle's current pose and the pose of the waypoint being tracked, or the star waypoint. If the current HLC is follow lane, the star waypoint follows the vehicle and stays in the center of the lane being occupied.

Table 3.4: The high level commands used as contexts for the affordance predictor and their corresponding one-hot form

High Level Command	**One-Hot Representation**
Follow Lane	$[1, 0, 0, 0, 0]$
Right Turn	$[0, 1, 0, 0, 0]$
Left Turn	$[0, 0, 1, 0, 0]$
Right Lane Change	$[0, 0, 0, 1, 0]$
Left Lane Change	$[0, 0, 0, 0, 1]$

Heading error is the angular difference between vehicle heading and star waypoint heading, and crosstrack error is the lateral distance between the star waypoint and the vehicle's geometric center, with the waypoint's transform as the origin. This introduces a slight discrepancy as the curvature based steering controller derived in subsection 4.2.1 is based on a kinematic vehicle model with the reference point being the middle of the rear axle, instead of the geometric center of the car. However, on most passenger cars, the distance between these two points are about a meter, so this presents no significant effect on data quality or vehicle control. If a longer vehicle such as a bus was being used as the ego vehicle in the recorder, the crosstrack error calculation can be modified to suit by first finding the global transform of the rear axle midpoint, then calculating the lateral distance between it and the star waypoint.

When a turn or lane change HLC is selected, the waypoint fetching mechanism checks to see if a turn or lane change is possible at the current location. If it is, the star waypoint jumps to the requested spot, altering the heading and crosstrack error (Figure 3.5). Otherwise, it keeps following the ego vehicle in the current lane. This is implemented using a queue of waypoints. As the vehicle drives, new waypoints are pushed to the queue and waypoints that are now behind the car are popped. Selection for the next waypoint that gets pushed to the queue is based on current HLC. If the vehicle strays too far from the queue, the entire queue is cleared and repopulated from the the new vehicle location.



(a) (b)

(c)

Figure 3.5: As the user activates different high level commands, the waypoint queue (in green) shifts to their appropriate locations. (a) and (b) show the waypoint queue shifting for left and right lanechanges, (c) is in follow mode

Figure 3.6: User interface of the data recorder

Chapter 4

Direct Perception

This chapter details the development of a conditional direct visual perception model that conducts scene understanding on sensory input and context. This approach belongs to the direct perception class of robot control, and can easily be adapted to accept more sensor modalities. The result of scene understanding is then used by a physics derived controller to drive the vehicle. A similar context set as the imitation learning model is used, with additional contexts of left and right lane change. Since a new dataset is being recorded to train this task, this is a good opportunity to add these two contexts to form a more complete intelligent driver.

Figure 4.1 visualizes the three paradigms mentioned in the introduction; mediated perception, behavior reflex, and direct perception. Keep in mind that aside from sensory stimulus, all three modules must be receiving some type of external context. For simplicity this was not shown in the diagram. The end-to-end imitation learning policy networks discussed in chapter 2 belong to the behavior reflex group. Before getting in to the design of the direct perception approach, some drawbacks of imitation learning and mediated perception are presented.

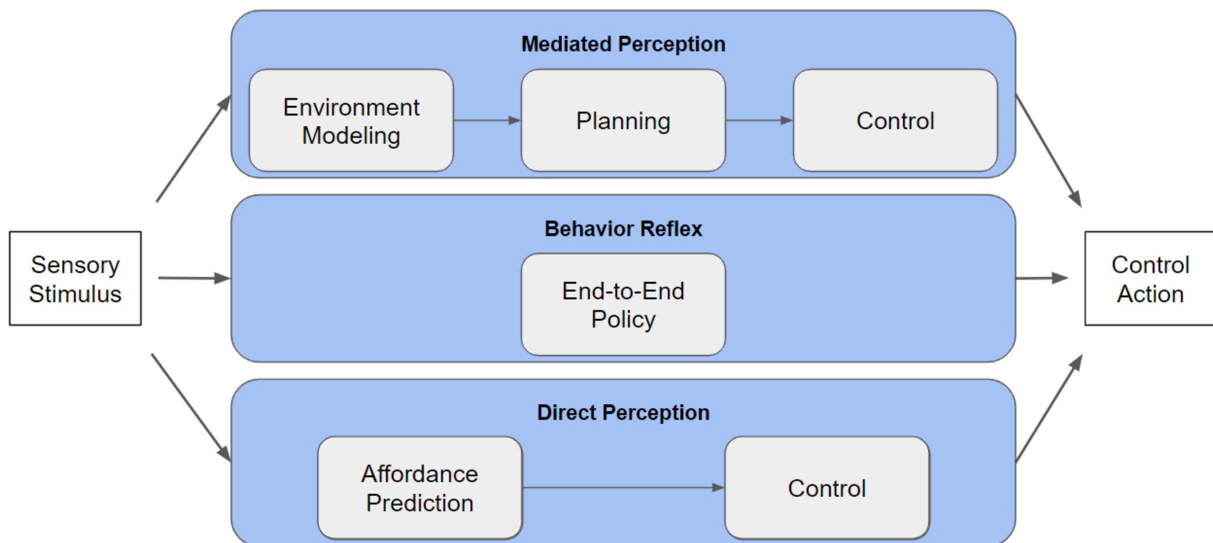


Figure 4.1: The 3 paradigms in autonomous driving

4.1 Motivation

4.1.1 Comparison to Imitation Learning

Imitation learning policies cover the entire onboard pipeline from perception to control. Making the control portion of the pipeline learning based has several disadvantages, the main one being explainability. For any physical system that is run by a computer, the control module is the interface between the digital world and the real world. If the operation of this module is difficult to explain and debug, proving that it will demonstrate defined behavior at all times will be a challenge; therefore it is also difficult to justify developing it all the way to production.

The second disadvantage of end-to-end policies is the inefficiency in scaling it to many different types of vehicles. Every car has its own unique set of dynamics, and running the policy on one car with the policy having been trained by data from a different car could lead to passenger discomfort or unsafe driving. This problem can be partially solved by having the policy predict variables that lead to the same physical reaction (e.g. turning radius instead of steering angle, or acceleration instead of throttle/brake percentage). However, there are still differences between the way each model of car acts on actuation signals that

make this an imperfect solution.

Another major drawback for end-to-end policies is the inability to tune the controller's behavior. The control portion is embedded into the overall policy and there is no way of abstracting it away to a separate module. With classical control methods such as state space feedback, gains can be adjusted to operate differently, and is often done to achieve a desired response time or energy efficiency. With a neural network controller, this type of parameter tuning is no longer feasible and final behavior is up to the training data. The network would have to be conditioned to behave in different ways, but that involves creating conditional datasets for all the behaviors, which is tedious. To provide an example for the easy-to-adjust advantage that classical controllers have, for adaptive cruise control, if a rules based controller such as a compound PID is used, it can be easily tuned both in development and production, on various settings such as follow distance and time gap.

4.1.2 Comparison to Mediated Perception

The control module itself can actually be very similar (Proportional Integral Derivative Controller (PID), Linear Quadratic Regulator (LQR), MPC) for both direct perception and mediated perception approaches. The difference is in how sensory stimulus is processed before reaching the control module. Mediated perception typically tries to build a full understanding of the environment. The pose and motion of all perceivable obstacles are obtained by fusing the stimuli from different sensor modalities. Common modalities are vision, lidar, and radar. Once the environment is modeled, the planning module creates a path through the environment for the control module to track.

All theses operations in a mediated perception pipeline require a large amount of onboard computation. This means the vehicle will be less efficient. With tightening emissions standards globally and the advent of electric vehicles where range anxiety is still an issue, having a power hungry system is highly undesirable.

Although the perception module in a mediated perception pipeline is more explainable compared to affordance prediction due to its modular nature, many subcomponents of the perception module are still learning based with no more explainability than an affordance policy. Affordance prediction also a simpler task than environment modeling, since it is trying to obtain only variables that are essential to vehicle operation.

4.2 Affordance Selection

This thesis aims to develop an affordance predictor that predicts a set of affordances that can be used by a classical controller to drive a vehicle. For a fair comparison with the imitation learning policy, the same policy inputs will be used: a single RGB image from a front facing camera, and the current longitudinal velocity. The affordances it provides must be usable by a controller to control both longitudinal and lateral motion of a vehicle. They also must be general enough for this direct perception pipeline to be easily scaled to cars with different dynamics.

4.2.1 Affordances for Lateral Control

To comply with the requirements outlined above, the set of affordances must lead to steering angle and throttle/brake calculation, and the controller that takes in the affordances to generate control actions must be easily tuned to exhibit different behavior for different application specific requirements. When humans drive, we use visual cues to decide how to steer the vehicle. However, we are not calculating a precise front wheel steering angle, then carefully adjusting the steering wheel to reach that angle. Rather, we have a mental image of where the car should be, and we move the wheel in a way that directs the car down our mental goal path. Every vehicle also responds differently to the same front wheel steering angle. For these two reasons, the affordance set should include variables that enable a controller to manipulate cars with vastly different dynamics to trace out the same path.

For different cars to trace out the same path, they need to achieve the same turning radius. Therefore, turning radius is selected as one of the affordances. A small practical problem that arises from this is that when the road is straight, the turning radius is infinity, and that would lead to numerical instability when training the affordance predictor. As suggested by [9], we can simply take the inverse of turning radius, curvature, and use it instead. To fully describe the vehicle's deviation from the ideal position, heading error and crosstrack error can be used (Figure 4.3). Heading error is defined as the angular difference between lane heading and vehicle heading, and it can be used directly as a part of a control law for steering angle. Crosstrack error is defined as the lateral distance between the vehicle and the point on the ideal path closest to the vehicle. These two physical values have been successfully extracted from RGB images by numerous studies in the past [32, 50], so they are suitable as affordances as well.

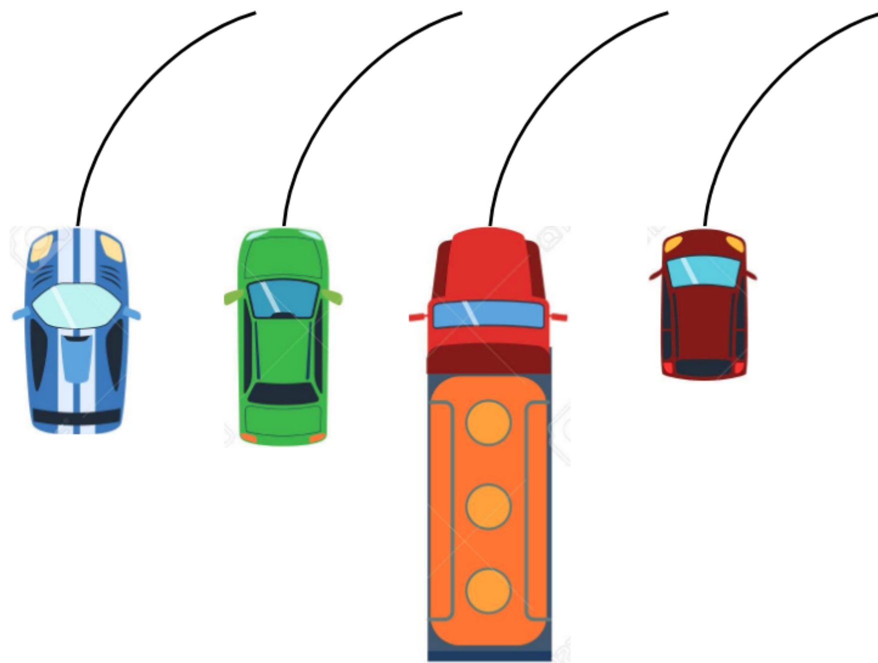


Figure 4.2: A generalized affordance set will allow the direct perception pipeline to control different vehicles to drive the same way with minimal vehicle specific development

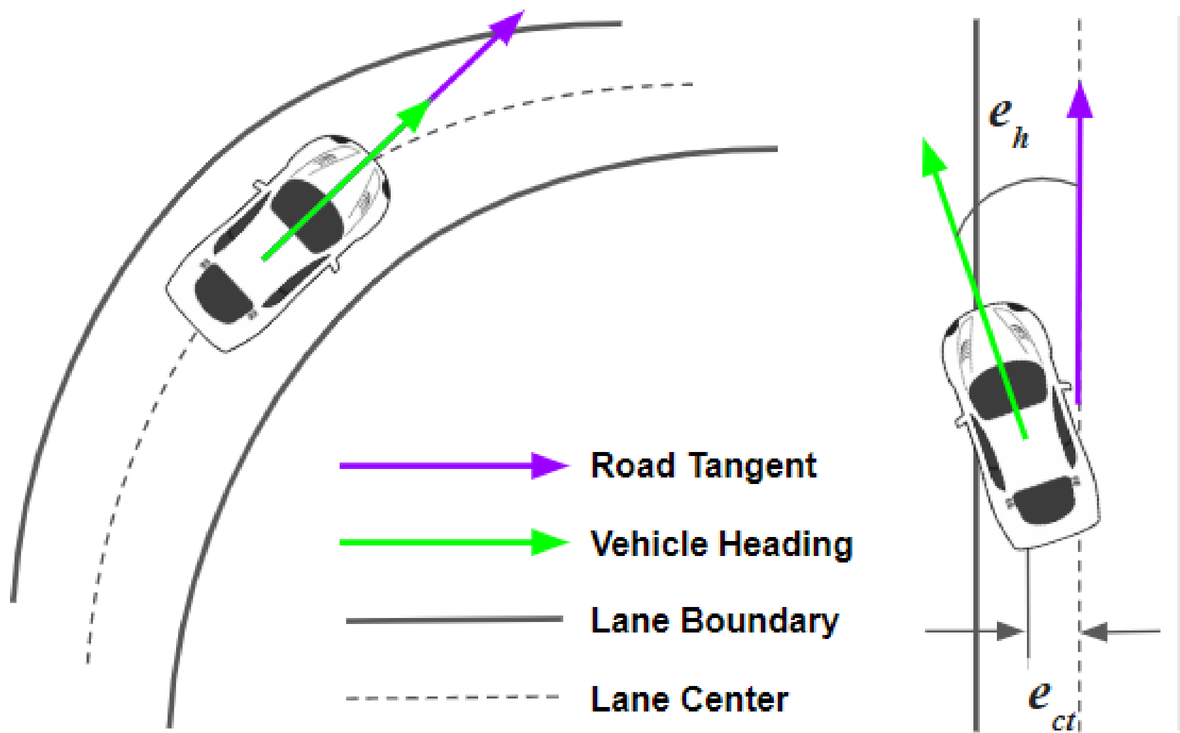


Figure 4.3: Illustration of heading and crosstrack errors

A point worth mentioning is that crosstrack error and heading error define an instantaneous error state, while curvature is not an error but a reference, and it describes a path instead of a positional error. When an error based feedback controller is used, its output works to reduce the error. When a curvature based controller is used, its output does not affect the next perceived curvature, making it a feedforward controller. If all three affordances are used to steer the vehicle, it would be a hybrid feedforward/feedback controller. The fact that curvature defines a path also opens up the possibility for predictive controllers, since the steering input can be optimized to trace out a path of a given curvature. To summarize, the affordances for lateral control have been selected as curvature, crosstrack error, and heading error. A framework to steer different vehicles using the same set of affordance predictor weights (Figure 4.2) has been proposed.

4.2.2 Affordances for Longitudinal Control

A car following model similar to [50] is used for longitudinal control, and affordances are selected to enable this controller. If there are no obstacles in front, the car is controlled to track a reference speed v^*. Given the current vehicle speed v lower than the target speed v^*, we implement the optimal velocity tracking model [14] as

$$v = K_v v^* (1 - \exp - \frac{K_f}{d_f v^* - d}) \tag{4.1}$$

where d_f is the distance to the preceding vehicle, K_v, K_f and d are parameters. In the case where the driving speed v is more than 10 km/h larger than the target speed limit v^*, we set the throttle value to zero and the brake value similar to [50]

$$brake = K_b \frac{v}{v^*} \tag{4.2}$$

We set $K_b = 0.35$ in our experiment based on the vehicle we simulated. $brake$ is a measure of braking effort with its value between 0 and 1, and would be converted to a brake pressure in implementation. When there is an obstacle in front of the ego-vehicle in a very short distance, we set full brake in order to prevent crashes.

Based on the above longitudinal control scheme, the only affordance needed is distance to the vehicle in front. In a world where connectivity in vehicles is already prevalent, the task of obtaining reference speed can be offloaded from the affordance predictor as it can easily be requested based on Global Positioning System (GPS) information or occupant intentions.

Table 4.1: Affordances predicted by the network

Affordance	**Unit**	**Conditional**
Road Curvature	m^-1	Yes
Heading Error	deg	Yes
Crosstrack Error	m	Yes
Distance to Car in Front	m	No

Table 4.1 show the list of all affordances used to control the vehicle. The conditional affordances are based on the HLC of a specific datapoint, and the unconditional affordance is always true regardless of HLC. The network needs to be able to distinguish between obstacles that are simply in its field of view and obstacles that are actually obstructing the ego vehicle. To achieve this, the training data needs to be labeled in such a way that the predictor only lowers the distance affordance if an obstacle is occupying the same lane. Obstacles that have a large enough lateral distance or heading delta to the ego vehicle are omitted from distance calculations, so vehicles in another lane or oncoming vehicles (especially during a turn) do not bring the ego vehicle to a sudden stop. This is implemented in the data recorder that was detailed in chapter 3.

4.3 Affordance Predictor

4.3.1 Model

A similar branched architecture as in section 2.2 is used. There are branches at the end for conditional predictions and an additional branch for non conditional predictions. Each conditional branch outputs the same set of conditional affordances listed in Table 4.1. There is also a speed output that forces the CNN to encode speed information to regularize against imbalanced speed data. Branch loss is computed using algorithm 2, and summed with losses from unconditional outputs. Figure 4.4 shows a diagram of the entire control architecture, including the network.

The branched neural network was built in a much more modular way after lessons learned from building them for earlier parts of this thesis. This new way of model construction makes the tasks of swapping model components and scaling data much more efficient. The model in section 2.2 is monolithic, meaning all foundational network elements such as FC and CNN layers are built into a single network class. This manner of model construction poses a large

Table 4.2: Validation performance of different backbones with all other parameters held constant

Backbone	**Validation Loss**
ResNet18	0.02
MobileNet	0.15
ShuffleNet	0.08
ResNext	0.06

impedance on architecture and parameter search, since the class source code has to be modified for every new attempt in the search. To remedy this issue, numerical parameters can be added as arguments for the class constructor method, but there is an even better solution that allows more flexibility. A shell class for the network is built, and its constructor method takes in a Python dictionary as an argument, and each entry in the dictionary defines a component of the network. This expedites the process of changing large network components. There is no longer a need to dig into the source code. For example, the image processing portion of the model is easily interchangeable between different CNNs (ResNet, NasNet, MobileNet, etc) simply by changing the CNN entry in the Python dictionary, which is loaded from a configuration file. The shell class gathers building blocks of the network from the dictionary, and parameters for these building blocks are loaded from another dictionary that holds numerical values. Making use of the modularity of the network, different proven CNN structures were tested as the image feature extractor, or the backbone, and results are shown in Table 4.2.

Aside from network parameters, this second dictionary also holds scaling factors for the training data. The labels used all have different scales, which can lead to suboptimal training. The activation functions in neural networks have a narrow activation zone where their gradient is nonzero, usually around where the input is zero. When the output of network layers are outside this activation zone, the resulting gradient becomes so small that the weights cannot get updated. Scaling data mitigates this vanishing gradient problem. These scaling factors are used to scale the labels to similar means and variances during training, and also to scale network outputs back to their original units during inference. Batch normalization is also used to ensure proper distribution inside the network.

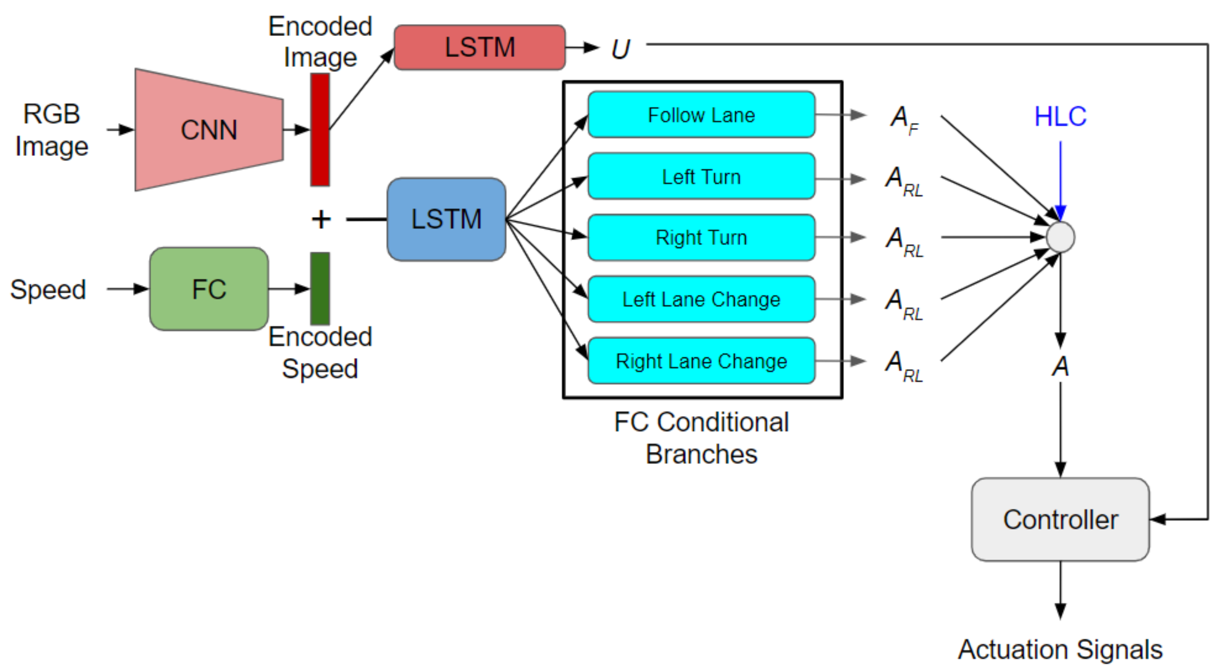


Figure 4.4: The direct perception pipeline, including the branched affordance predictor and controller

4.3.2 Inference

During inference, a mask is used to generate the final output. The outputs of all conditional branches are concatenated into a single $1x(n * m)$ vector, with n being the number of branches and m being the number of affordances in one branch. Based on current HLC, an n-hot mask is generated and applied to the branch outputs. Then, an interleave add operation is done on the vector to sum up the same affordance cell of all branches to form a $1xm$ vector which is the list of affordances based on the given HLC.

4.3.3 Offset Sensors

When we have control over the training data itself, another approach to tackle the imbalance issue is to add additional sensors that simulate different error states, inspired by [17]. There can be additional cameras that have a yaw angle or lateral translation deviation from the main camera which would be used during inference. During training, images from the offset cameras are paired with labels that represent those offsets, as if the vehicle was in a pose such that the main camera was receiving that view. For example, a camera that is yawed to the left by 30 degrees would get a heading error label that is 30 degrees more than that frame's original label. During training, sampling is done randomly out of all the available sensor snapshots. This method increases the amount of images that represent larger error states significantly, making it very powerful for generating more even label distributions. Note that if such a sensor is added, another sensor of the opposite rotation/translation also had to be added to even out the collected error states. Figure 4.5 does not show this symmetry for clarity.

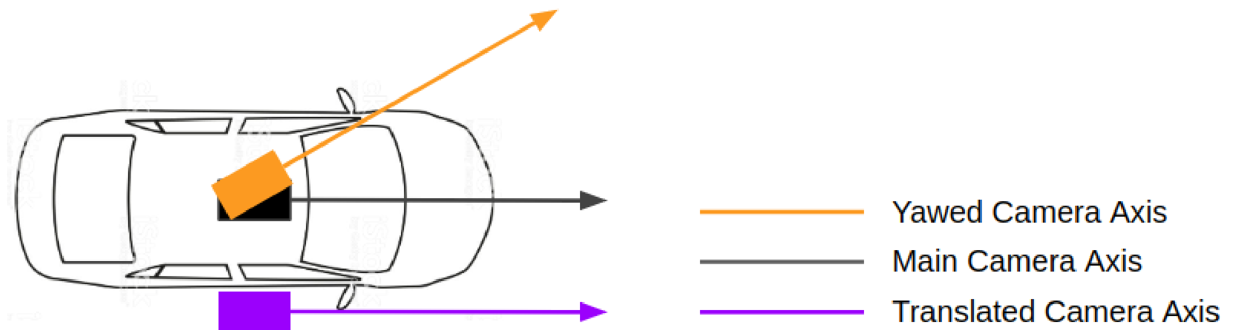


Figure 4.5: Simulated sensors can be offset from the main camera to generate datapoints with larger error states

Figure 4.6: Images from cameras yawed by -8 degrees (left), 8 degrees (right), and the main camera (center)

4.4 Results

Just as with the imitation learning models presented in the previous chapter, the direct perception approach is benchmarked in a variety of training and deployment conditions Table 4.3. It was deployed back to the same town where training data was recorded, and models trained with/without augmentation and offset sensor training were tested. Training without using offset sensor data presented a larger drop in completion rate than training without data augmentation. When deployed to scenes unseen during training, the model only had a 7 percent drop in completion rate in an urban environment compared to a 23 percent drop in a rural environment. This means the model has generalized to urban scenarios well, but more work needs to be done on improving generalizing across different domains.

Some common failure modes are shown in Figure 4.7. Although the affordance predictor was generally reliable in adhering to HLCs, there was an interesting failure mode in particularly wide intersections. The model sometimes did not turn if a turning HLC was issued when the vehicle was driving through an intersection. A possible cause was the camera field of view was not wide enough to provide visual cues that suggest there is a turn available. Increasing the camera's field of view or using multiple sensors could remedy the problem. Another issue was that the model does not seem to have any reservation about driving down the opposite lane. While there are examples of opposite lane driving (with appropriate heading error labels) in the dataset, their occurrence is minimal and perhaps more should be included.

Figure 4.8 shows distance predictions made by the network, which is crucial for safe longitudinal control. In top right and bottom left examples, the network has correctly predicted that the distance should be 50 (the default max value for when there are no obstacles in front) even though there is a vehicle in its field of view. When vehicles in view are not in our way, they are not seen as obstacles and do not affect the predicted distance.

Figure 4.7: Common failure modes for direct perception seen in testing

Pred = 1.83
Label = 1.26

Pred = 29.83
Label = 32.62

Pred = 50.0
Label = 50.0

Pred = 50.0
Label = 50.0

Pred = 12.83
Label = 13.40

Pred = 10.36
Label = 9.59

Figure 4.8: Distance predictions vs. labels

Table 4.3: Results of direct perception in different scenarios

Scenario	Completion %	Minor Infraction/km	Major Infraction/km
Training Town, Urban	87	2.3	0.6
Training Town, Urban, No Aug.	65	3.8	1.3
Training Town, Urban, No Offset Sensor Training	52	5.7	3.3
Unseen Town, Urban	81	3.1	1.2
Unseen Town, Rural	67	2.9	1.6

Chapter 5

Controller

The previous chapter described models that generate affordances for controllers, and reasoned why each affordance was used. This chapter introduces two lateral controllers that use these affordances to generate steering angles for the vehicle, one that is based on a kinematic model and one that is based on a dynamic model.

5.1 Kinematic Controller

5.1.1 K Stanley Controller

The Stanley controller, shown in equation (5.1), is a common controller used for lateral control, and it requires crosstrack error and heading error. These two physical values have been

$$\delta_s = e_h + \arctan\left(\frac{K_s(l_f + l_r)}{v_x}\right) \tag{5.1}$$

$$\delta_k = 90 - \arctan\left((l_f + l_r)\kappa\right) \tag{5.2}$$

For now, we will assume the vehicles are driving slow enough that the no slip condition holds true [31]. A simple deterministic relationship can be derived between ideal condition steering angle δ_k and road curvature κ. The arctan of total wheelbase $l_f + l_r$ times κ is

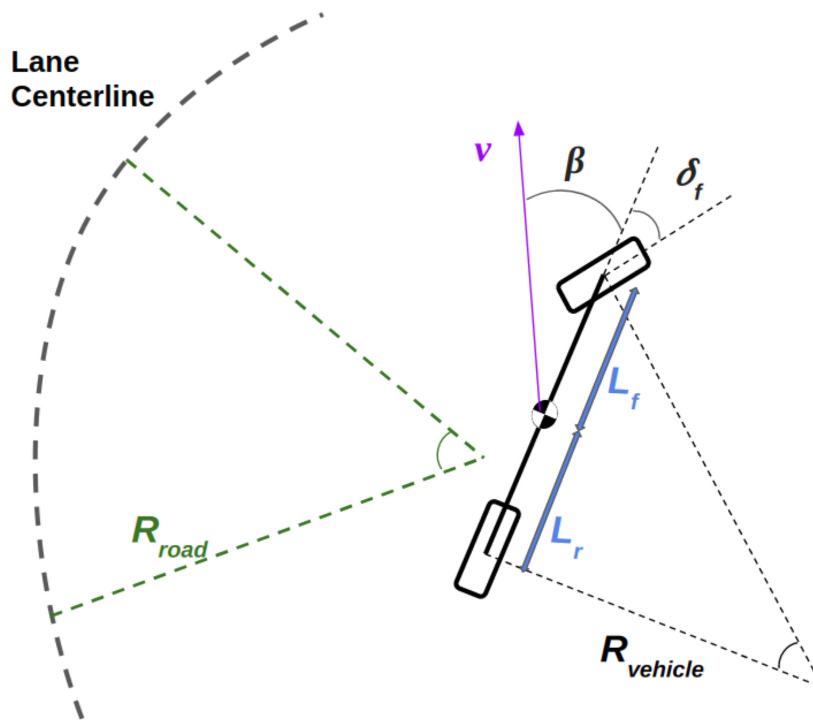


Figure 5.1: The K Stanley controller attempts to match vehicle path curvature to road curvature, as well as minimize heading and crosstrack errors

the complementary angle of the steering angle that would lead the car to trace a path of the given radius. δ_k is then summed with δ_s, the angle given by Stanley, to form the final steering control output. When the vehicle is traveling on a straight stretch of road, the Stanley controller is expected to correct minor deviations from the lane center line that results from road surface imperfections. When the vehicle is on a curved road, it is impossible for a vehicle controlled by Stanley to perfectly track the lane center line, as error has to accumulate before it can generate corrective steering commands. This is where the curvature affordance comes in. Knowing the vehicle's wheelbase, an ideal steering angle to track a curved path can easily be found. This will be used to generate a "baseline" steering angle for the car to track the lane curvature while the Stanley controller corrects any minor deviations (Figure 5.1). This structure also reduces reliance on the Stanley controller, which is prone to oscillations in large error scenarios.

5.2 Dynamic Controller

5.2.1 Single Track Lateral Motion Model

The K Stanley controller from subsection 4.2.1 is based on a kinematic model. It assumes no wheel slip, as if the vehicle is driving on rails. This works for simulations which have simplified physics. However, in real life at higher speeds, this assumption is no longer valid, and lateral vehicle motion needs to be analyzed in terms of forces generated by wheel slip. The lateral force generated by a tire is its coefficient of friction multiplied by normal force (vertical load on the tire). The coefficient of friction is a function of the tire's slip angle, α. In vehicle dynamics, slip angle can refer to the slip angle of the car, which is used as a measure of stability; or the slip angle of tires, which is used to analyze force generation. In this chapter, mentions of slip angle will be referring to tire slip angle, which is steering angle minus body slip angle, shown in Equation 5.3 and illustrated in Figure 5.2.

$$\alpha = \delta - \arctan\left(\frac{v_y}{|v_x|}\right) \tag{5.3}$$

The physical meaning of slip angle is the angle between the tire's direction of travel and the direction it is pointing in. When a car is driving in a straight line, there is no slip angle. When the steering wheel is turned at a fast enough longitudinal speed, the tires generate slip that results in the vehicle changing its heading. This force-slip relationship is somewhat analogous to how force is generated from deformation in a spring. Tire slip leads to deformations at the contact patch. Deformations cause shear stress which results in forces that act on the car. Unlike a spring, it is comprised of a stiffness and a friction component. At lower slip angles, lateral force comes purely from cornering stiffness, which is represented by the slope of the linear region. As slip increases, coulomb friction kicks in and contributes to the plateau portion of the force-slip curve, as shown in Figure 5.3.

In day to day driving, tires usually do not go into the nonlinear region, therefore it is an acceptable simplification to design a controller based only on the linear region. With this assumption in mind, a dynamic lateral motion model can be formulated for the direct perception pipeline to operate at higher longitudinal speeds. Equation 5.4 and Equation 5.5 show the linearized slip angle and tire models, where tire lateral force F_y is obtained by multiplying the combined (left and right tire) cornering stiffness C_a with slip angle. v_x, v_y are respectively longitudinal and lateral velocities of the vehicle, a is the distance from the front axle midpoint to the center of mass, b is the distance from the rear axle midpoint to the center of mass, w is yaw rate, and δ is front wheel steering angle.

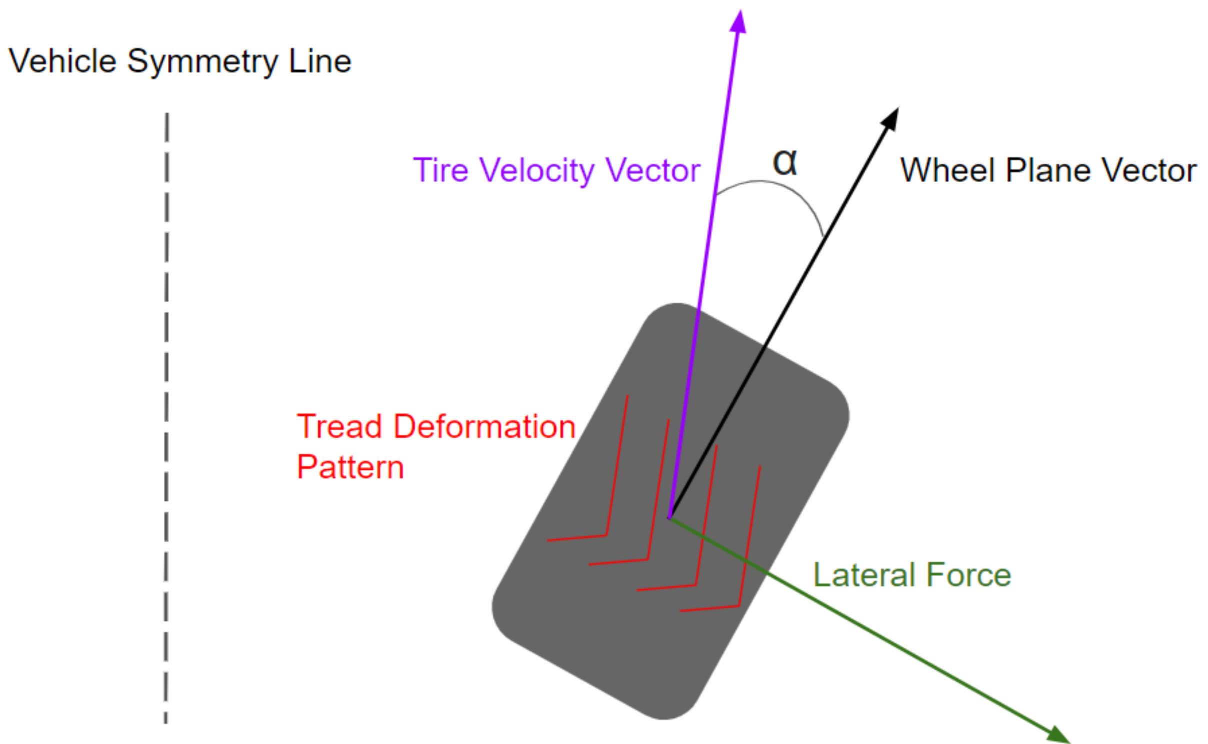


Figure 5.2: Top down view of a right front tire during a right turn. When a rolling tire is pointed in a direction different than its velocity vector, lateral force is generated

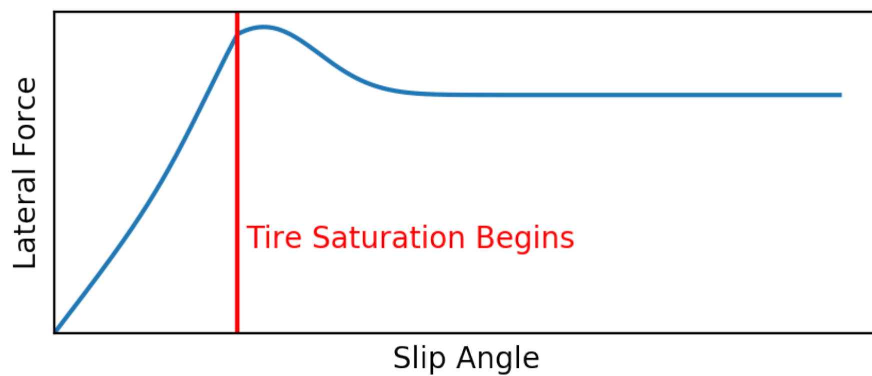


Figure 5.3: A qualitative curve of lateral force vs. tire slip angle. Before a threshold slip angle, the relationship is linear

$$\begin{aligned}\alpha_f &= \delta - \frac{v_y + aw}{v_x} \\ \alpha_r &= -\frac{v_y - bw}{v_x}\end{aligned} \tag{5.4}$$

$$\begin{aligned}F_{yf} &= C_{af}\alpha_f \\ F_{yr} &= C_{ar}\alpha_r\end{aligned} \tag{5.5}$$

$$\begin{bmatrix}\dot{v}_y \\ \dot{w}\end{bmatrix} = \begin{bmatrix} -\frac{C_{af} + C_{ar}}{v_x m} & -(\frac{aC_{af} - bC_{ar}}{v_x m} + v_x) \\ -\frac{aC_{af} - bC_{ar}}{v_x I_z} & -\frac{a^2 C_{af} - b^2 C_{ar}}{v_x I_z}\end{bmatrix} \begin{bmatrix} v_y \\ w \end{bmatrix} + \begin{bmatrix}\frac{C_{af}}{m} \\ \frac{aC_{af}}{I_z}\end{bmatrix} \delta \tag{5.6}$$

Knowing the tire model, lateral forces and yaw moment can be summed to obtain the lateral model (Equation 5.6) in continuous state space form $\dot{X} = AX(t) + BU(t)$. m is vehicle mass and I_z is yaw inertia. Lateral velocity and yaw rate are the states, and steering angle is the input. To create a controller using this model means solving for an optimal steering angle based on the model's outputs. The aim is to match the model's projections for vehicle states to what the affordances dictate they should be. The lane curvature affordance describes the short term ideal path of the vehicle. Even though a path can have varying curvature, a new control action gets recalculated every timestep so the controller is able to handle the constantly changing curvature.

5.2.2 Controller Derivation

The model is propagated forward in time to obtain projected waypoints. The waypoints are then used to calculate projected curvature. Unlike most MPC strategies with prediction horizons of ..., only a single timestep propagation is needed because this returns a differential state, which can be interpreted as the state difference between two timesteps. Knowing this difference is sufficient for curvature calculation under the constant curvature assumption. A loss function is calculated between the target curvature and the predicted curvature, and the change in loss with respect to the steering angle is calculated. This is the gradient for the steering angle. A solver uses this gradient to update the steering angle to bring it closer to the optimal value, which should guide the vehicle to trace out a path with the correct curvature.

The state transition and input matrices from Equation 5.6 (A and B matrices) are used to obtain the state derivatives (Equation 5.9). The states can then be realized by integrating it with dt (Equation 5.10). With state x being $[v_y, \omega]'$ and control input u being front wheel steering angle δ. The realized state is the prediction *one* timestep into the future. Knowing this prediction, the predicted curvature can be calculated. Those familiar with model predictive control will notice the lack of state space temporal stacking that is a part of the derivation for a standard MPC. This stacking process is done to compile a modified state space model that outputs a time series of future predicted states, resulting in large modified A and B matrices that are computationally intensive to work with. This particular controller does not need any temporal stacking because states from only a single timestep is needed for a transformation into curvature. Curvature defines a path, and this path is compared with the curvature of a reference path for loss and gradient calculation. In other words, the transformation from state to curvature replaces the prediction horizon of a standard MPC.

$$x = \begin{bmatrix} v_y \\ \omega \end{bmatrix} \tag{5.7}$$

$$u = \delta \tag{5.8}$$

$$\dot{x} = Ax + Bu \tag{5.9}$$

$$x = \dot{x}dt \tag{5.10}$$

When the predicted state x is obtained, it is used to calculate the curvature. Two points are used to help find this curvature. The first is the vehicle's current location, and the second is the vehicle's predicted location based on the predicted state. Under the constant curvature assumption, these two points lie on the same circle with its origin being the vehicle's instant center of rotation. The curvature κ is the inverse of the circle's radius r. If the triangle that is formed between these two projected points and the circle's origin is split into symmetrical halves (Figure 5.4), we can use known euclidean distance d and heading difference α between these two points to calculate radius r and in turn curvature κ.

$$d = \sqrt{(v_y dt)^2 + (v_x dt)^2} \tag{5.11}$$

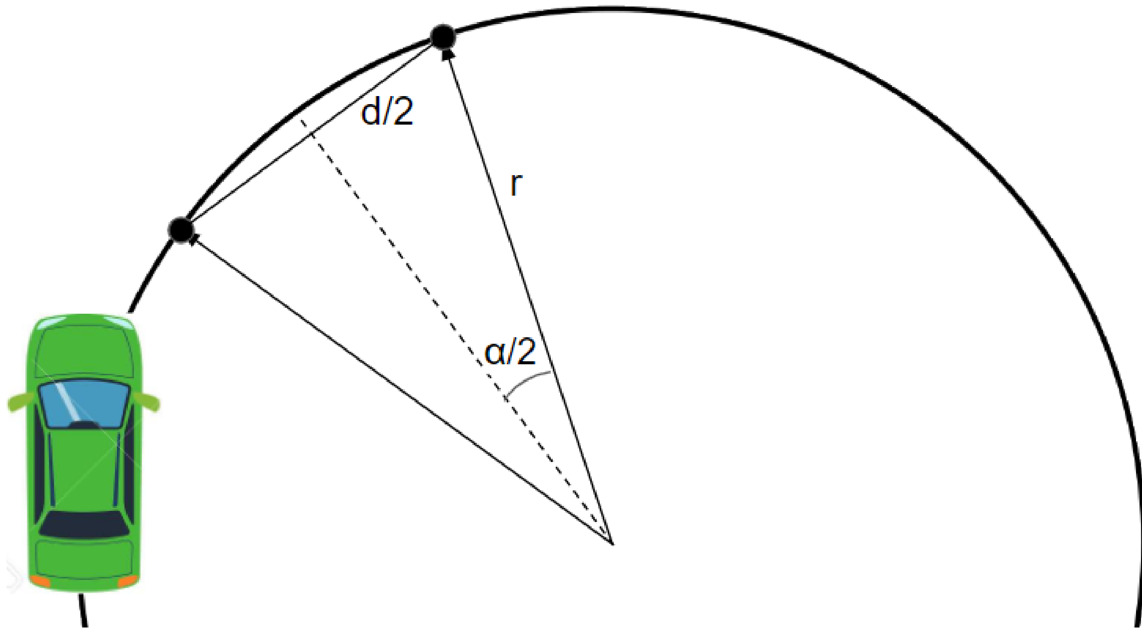


Figure 5.4: Curvature derivation based on driving around in a circle

$$\alpha = \omega dt \tag{5.12}$$

$$\kappa = \frac{2sin(\alpha/2)}{d} \tag{5.13}$$

$$\mathcal{L} = (k - k_{ref})^2 \tag{5.14}$$

Knowing the predicted curvature, a loss can be calculated between it and the reference curvature from the affordance set κ_{ref} using mean squared error Equation 5.14. Based on the computation graph from forward propagation of the modified model all the way to the MSE loss (Equation 5.9 to Equation 5.14), the gradient of curvature loss L with respect to steering input u can be calculated by chaining together the series of partial derivatives from Equation 5.15 to Equation 5.20, resulting in Equation 5.22. It can then be used to obtain the optimal steering input by iteratively calculating the loss gradient and updating the control input u.

$$\frac{\partial \mathcal{L}}{\partial \alpha} = -\frac{2cos(\alpha/2)(d\kappa_{ref} - 2sin(\alpha/2))}{d^2} \tag{5.15}$$

$$\frac{\partial \mathcal{L}}{\partial d} = -\frac{4sin(\alpha/2)(d\kappa_{ref} - 2sin(\alpha/2))}{d^3} \tag{5.16}$$

$$\frac{\partial d}{\partial v_y} = \frac{v_y dt^2}{\sqrt{(v_y dt)^2 + (v_x dt)^2}} \tag{5.17}$$

$$\frac{\partial \alpha}{\partial \omega} = dt \tag{5.18}$$

$$\frac{\partial \mathcal{L}}{\partial v_y} = \frac{\partial \mathcal{L}}{\partial d}\frac{\partial d}{\partial v_y} \tag{5.19}$$

$$\frac{\partial \mathcal{L}}{\partial \omega} = \frac{\partial \mathcal{L}}{\partial \alpha}\frac{\partial \alpha}{\partial \omega} \tag{5.20}$$

$$\frac{\partial \begin{bmatrix} v_y \\ \omega \end{bmatrix}}{\partial U} = B \tag{5.21}$$

$$\frac{\partial \mathcal{L}}{\partial U} = B_m^T \frac{\partial \mathcal{L}}{\partial \begin{bmatrix} v_y \\ \omega \end{bmatrix}} \tag{5.22}$$

As the MSE loss function is convex, there is only one optimum. The steering angle series U is updated by subtracting the gradient $\frac{\partial \mathcal{L}}{\partial U}$ multiplied by an update rate. The quadratic nature of the loss function ensures stability by decreasing each "step" of the optimization process as the search process approaches the optimum. Figure 5.5 shows the way loss changes as a function of steering deviation from the optimal angle of roughly 10 degrees. Starting from an initial guess or proposed steering angle, the loss gradient is iteratively applied to the proposed steering angle until the difference in steering angle from the previous iteration to the current iteration is below a certain small tolerance, or the max loop count has been exceeded. This optimization is shown in Figure 5.6.

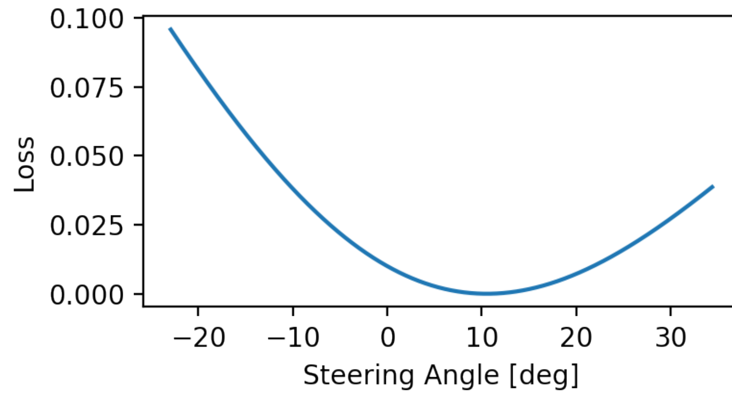


Figure 5.5: Steering loss when the optimal steering angle is near 10 degrees

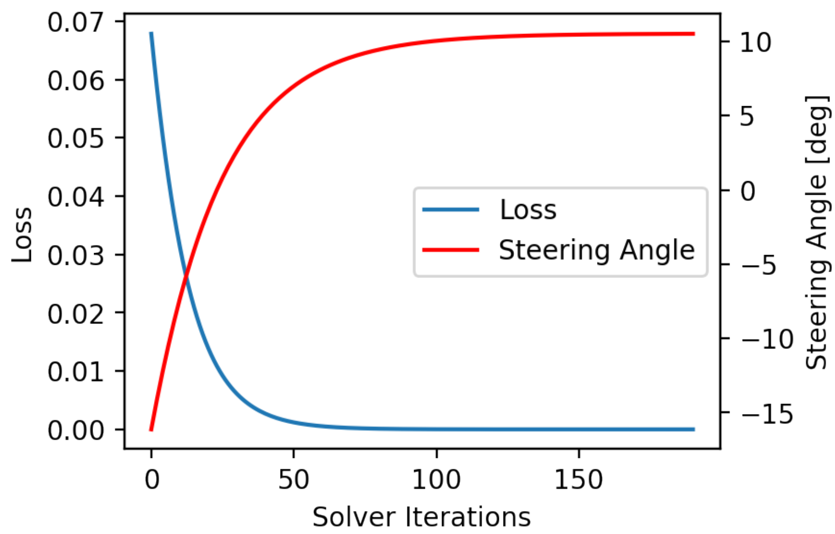


Figure 5.6: Loss curve and steering angle during the optimization process, which happens every timestep

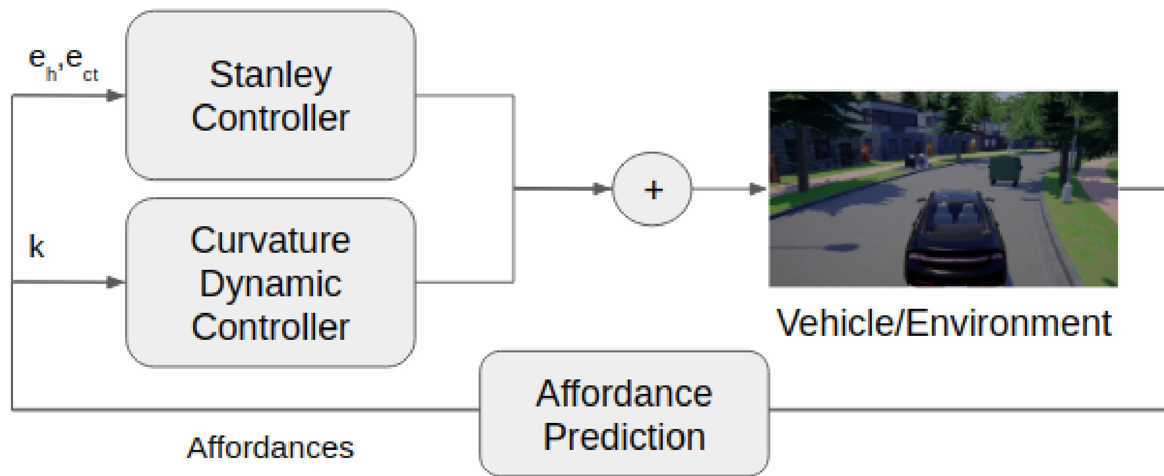


Figure 5.7: Lateral controller based on the single track dynamic model

So far, a controller for tracking lane curvature has been formulated. If the vehicle has a lateral offset with the center of the lane due to road cambers, wind, or other factors, it will not center itself using just this controller. There still needs to be a feedback component to minimize heading and lateral deviations. With the same structure as the K Stanley controller proposed above, crosstrack and heading error affordances are used by the standard Stanley steering controller while curvature is used by this dynamic model based controller. Their outputs are summed together to form the final steering command (Figure 5.7).

5.2.3 Limitations

The curvature dynamic control portion is based on a single track linearized model, so some physical effects are not accounted for. However, since the model will be used to solve for the optimal steering angle at every timestep, it having low state dimensionality and being a linear model makes it a suitable candidate for real time calculations in normal driving conditions. The most significant effects neglected by this model are listed below.

- Tire saturation
- Lateral load transfer

- Longitudinal load transfer
- Variable road surface conditions
- Change in contact patch size (camber gain)
- Steering geometry

Aside from the aforementioned plateau in lateral force as slip angle increases, this model also does not consider lateral and longitudinal load transfer. When a vehicle turns, its mass shifts to load the outside tires. The heavily loaded tires gain traction capability, but they amount they gain is less than the amount the inside tires lose. The decrease in overall lateral traction is not considered. Same goes for longitudinal load transfer, the changes in handling characteristics are not reflected in the model. Road conditions can vary, leading to changes in grip, and this is baked into C_a in this model, which is also fixed. As a wheel moves through its suspension travel, its camber angle changes based on suspension geometry. This also affects the amount of grip a tire can provide.

Out of all the effects neglected, tire saturation is the most significant, as all the other effects present a change in grip that trends with their severity, while tire saturation breaks the linear trend that exists the lower slip angle situations. Figure 5.8 shows the step steer input response of a vehicle traveling at 20 m/s predicted by this model. Vehicle parameters are representative of a mid size sedan. With the steering angle δ at 3 degrees, it predicts that the vehicle achieves a peak lateral velocity of more than 40 m/s. That's 144 KPH or 90 MPH in the lateral direction! To provide some scale, a three degree front wheel steering angle equates to roughly a 60 degree turn of the steering wheel. On a real car, this type of input would lead to severe understeer or severe oversteer situations. These inconceivably large lateral motion states are caused by the linear tire assumption.

Model inaccuracies due to tire nonlinearity do not happen until the tires leave their linear region (Figure 5.3). Tire grip typically tapers off at a slip angle α of around 7 degrees, but they start to exhibit nonlinear behavior when the slip angle is around 5 degrees. The dynamic model is trustworthy up until that happens. To determine what scenarios under which the model is inaccurate, a sweep of steering angle and longitudinal velocity is done to find the steady state slip angle in different driving scenarios (Figure 5.9). α is found by subtracting chassis slip angle β from the front wheel steering angle. With different vehicle parameters, the exact results would be different, however it is meant to reveal a general trend which further design choices can be based off of. Distilling the results of that sweep into feasible/infeasible regions using an α of 5 degrees as a threshold, we have Figure 5.10.

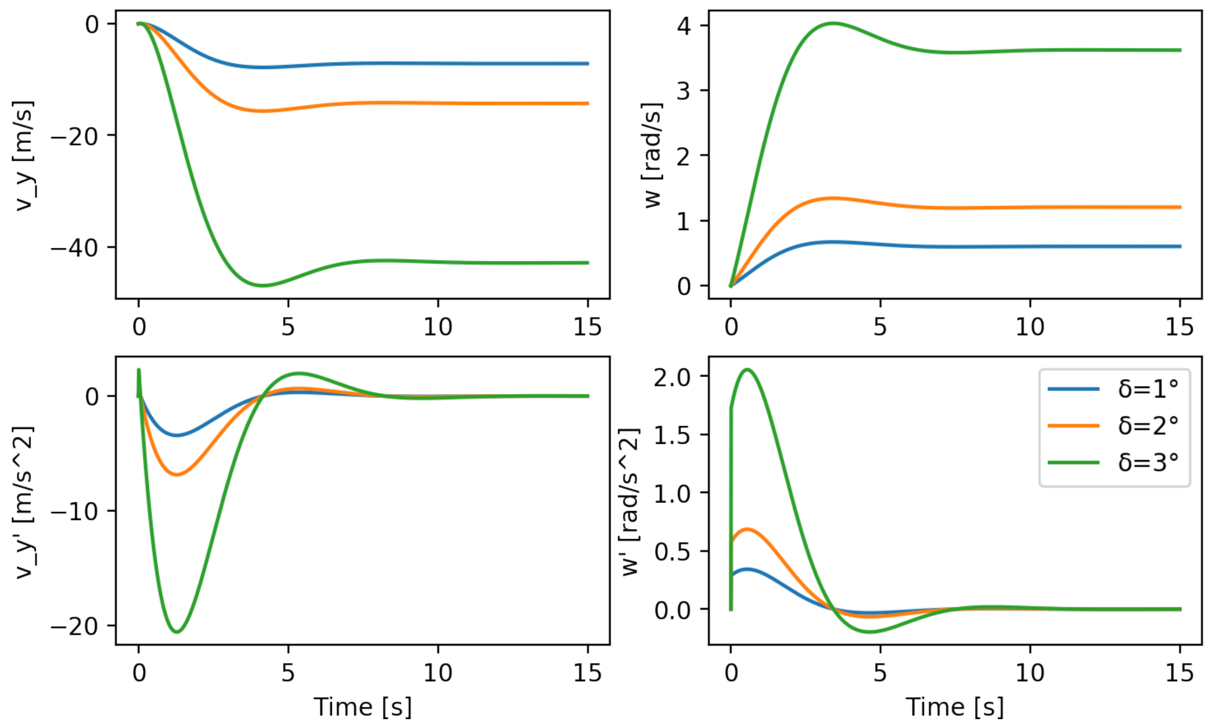


Figure 5.8: Step steer response of the single track lateral model. With its linear tire assumption, it provides some unlikely results even with a steering input of only 3 degrees

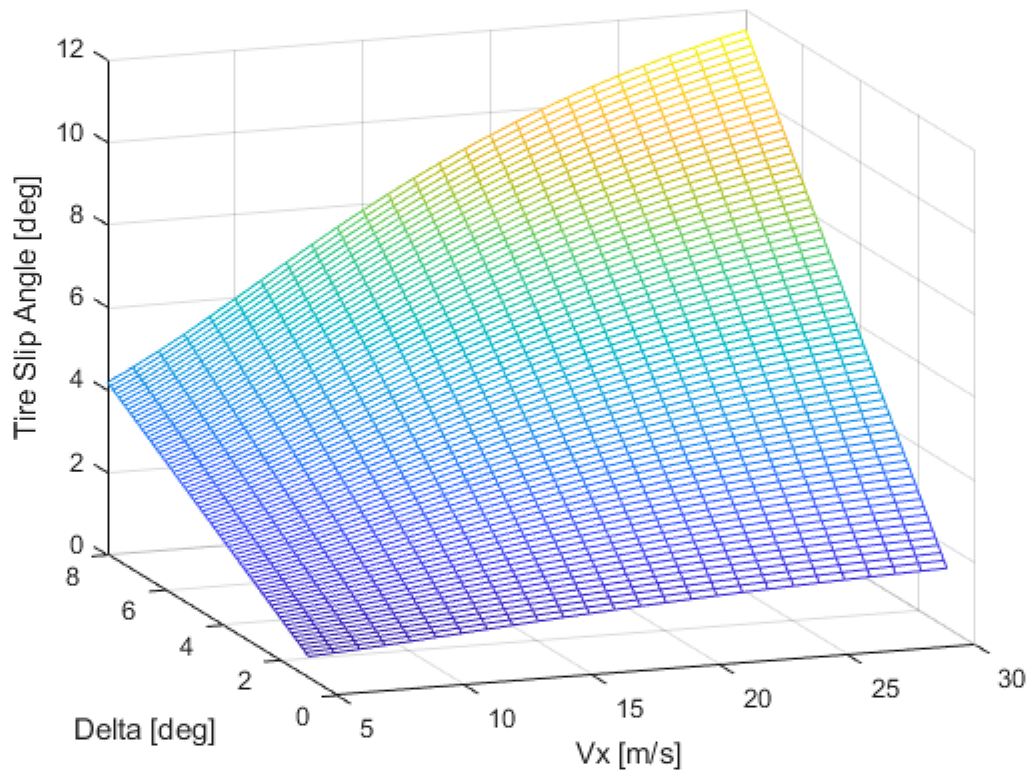


Figure 5.9: Longitudinal velocity and steering angle sweep of tire slip angle

It can be seen that under a certain steering input, tires never leave the linear region regardless of longitudinal speed. This steering angle, around 2.5 degrees in this case, might not seem like a lot, however assuming a typical 20:1 steering ratio, that would mean 50 degrees at the steering wheel. This amount of steering in day to day driving only happens during low speed turns where stability is not a concern. In addition, day to day driving on public roads do not push the tires past their linear region. Therefore, this model is suitable for normal road use.

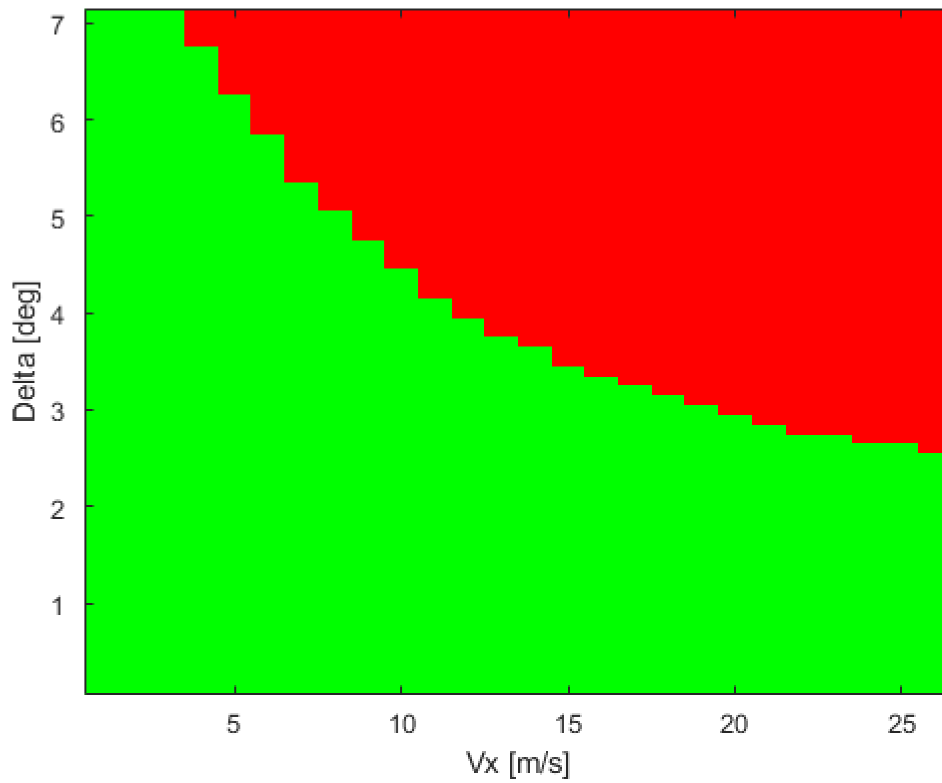


Figure 5.10: Passenger tires' lateral grip typically leave the linear region at a slip angle of 5 degrees. This figure shows conditions of when the tire slip angles are under (green) and above (red) 5 degrees

5.3 Results

The two controllers presented above were tested along with the standard Stanley controller. All three had similar performance in low speed, urban situations, but high speed driving was the differentiator. At higher speeds, all controllers begin to demonstrate oscillations, specifically after exiting a turn. For standard Stanley, this can be mitigated by adjusting the speed gain in the crosstrack error term. However, a gain that is suitable for high speeds would make the vehicle unresponsive at lower speeds. This highlights the need for gain scheduling, which is adjusting the gains of a controller based on current operating conditions. Considering gain scheduling leads to a rabbit hole of increasing development complexity when different vehicles with different sets of dynamics are involved, and the goal of direct perception is to develop an all encompassing control strategy with minimal vehicle specific development, gain scheduling is not pursued.

Both K Stanley and the dynamic controller use road curvature to generate a steering angle which is added to Stanley's output. This approach adds stability as the reliance on the oscillation prone Stanley is reduced. As shown in Table 5.1, both are able to maintain stability up to a higher speed than Stanley. Since K Stanley is a kinematic controller, the curvature component of the controller assumes no slip and thus underestimates the steering angle needed to match a given curvature, an effect that becomes more apparent with increasing speed. When it generates a steering angle of insufficient magnitude to match the desired curvature, heading and crosstrack error accumulate and Stanley steps in for correction. The dynamic controller does not need to fall back on Stanley as much, making it more stable at higher speeds. These results were generated by driving the vehicle with increasing longitudinal velocity around the highway in CARLA's Town04 map. Failure is defined as spinning out, leaving the lane, or any other unsafe vehicle states. The dynamic controller has the most stability by not showing oscillation at up to 101 KPH. While cars can certainly travel faster than that on the highway, they would slow down even for minor bends in the road to reduce its lateral acceleration, which is v_x/r^2, with r being the turning radius. The testing here was done with the vehicle being held at constant longitudinal velocities, without slowing down for turns, to explore the limits of the controller. When oscillation begins for the dynamic controller, the vehicle was under lateral accelerations of more than $6m/s^2$, which is far beyond the comfort threshold of $1.8m/s^2$ [57], which means in natural driving, the vehicle would never be subject to this combination of high speed and harsh lateral input. Figure 5.11 is a comparison of error regulation by all three controllers during and after a turn taken at just below the failure threshold speed of the least robust controller (Stanley). Aside from minimal oscillations, the dynamic controller also tracks the

lane better than Stanley and K Stanley. Figure 5.12 shows how the controller deals with when the vehicle needs to swerve out of the way of a slow moving or stopped obstacle in the same lane. Figure 5.13 shows a lane change scenario.

Table 5.1: Speeds above which oscillation and failure happen with each controller

Controller	**Oscillation Begins [KPH]**	**Failure [KPH]**
Stanley	66	79
K Stanley	72	82
Dynamic	101	114

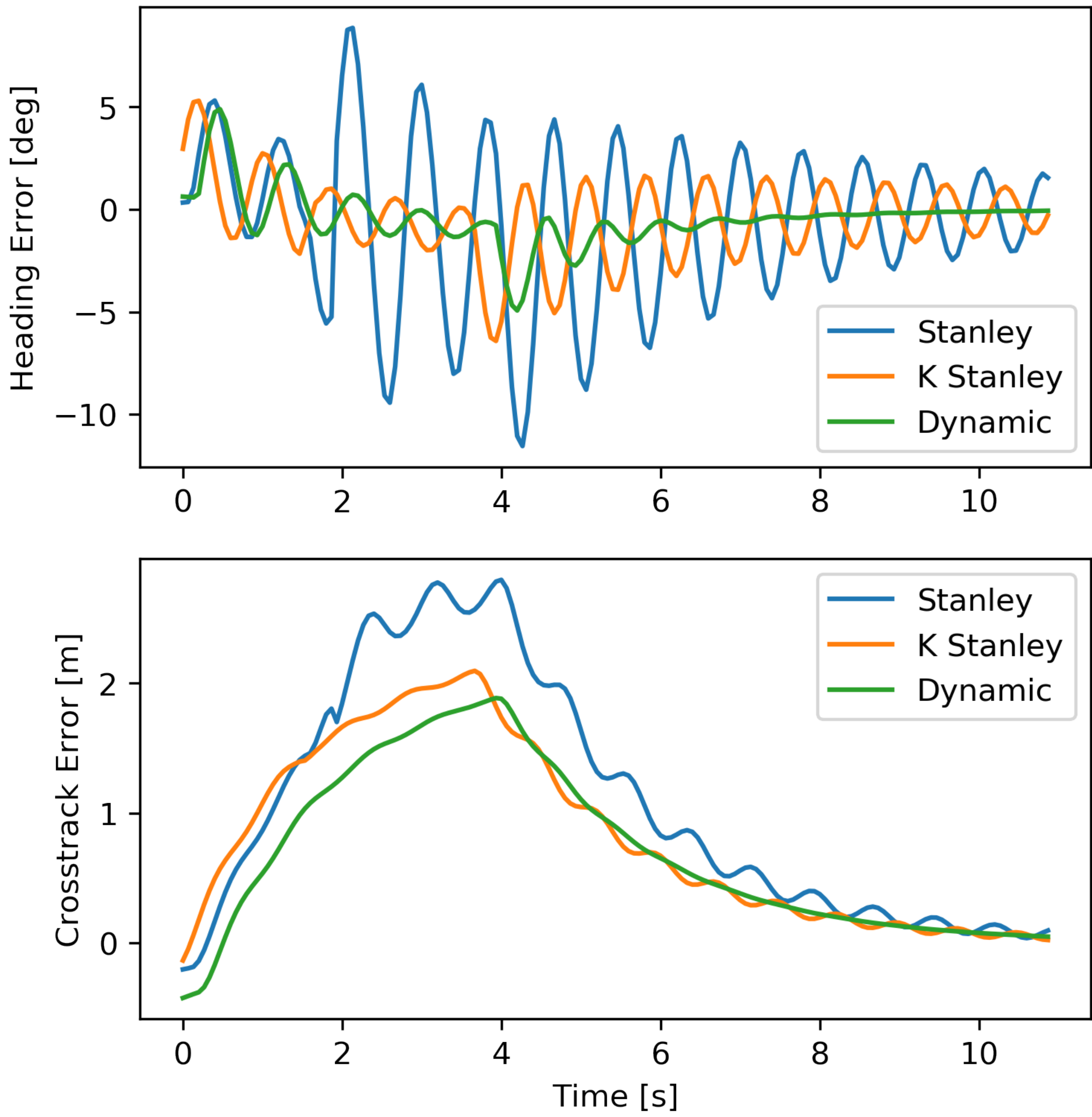


Figure 5.11: Lane deviation trace during and after a high speed turn using the three different controllers

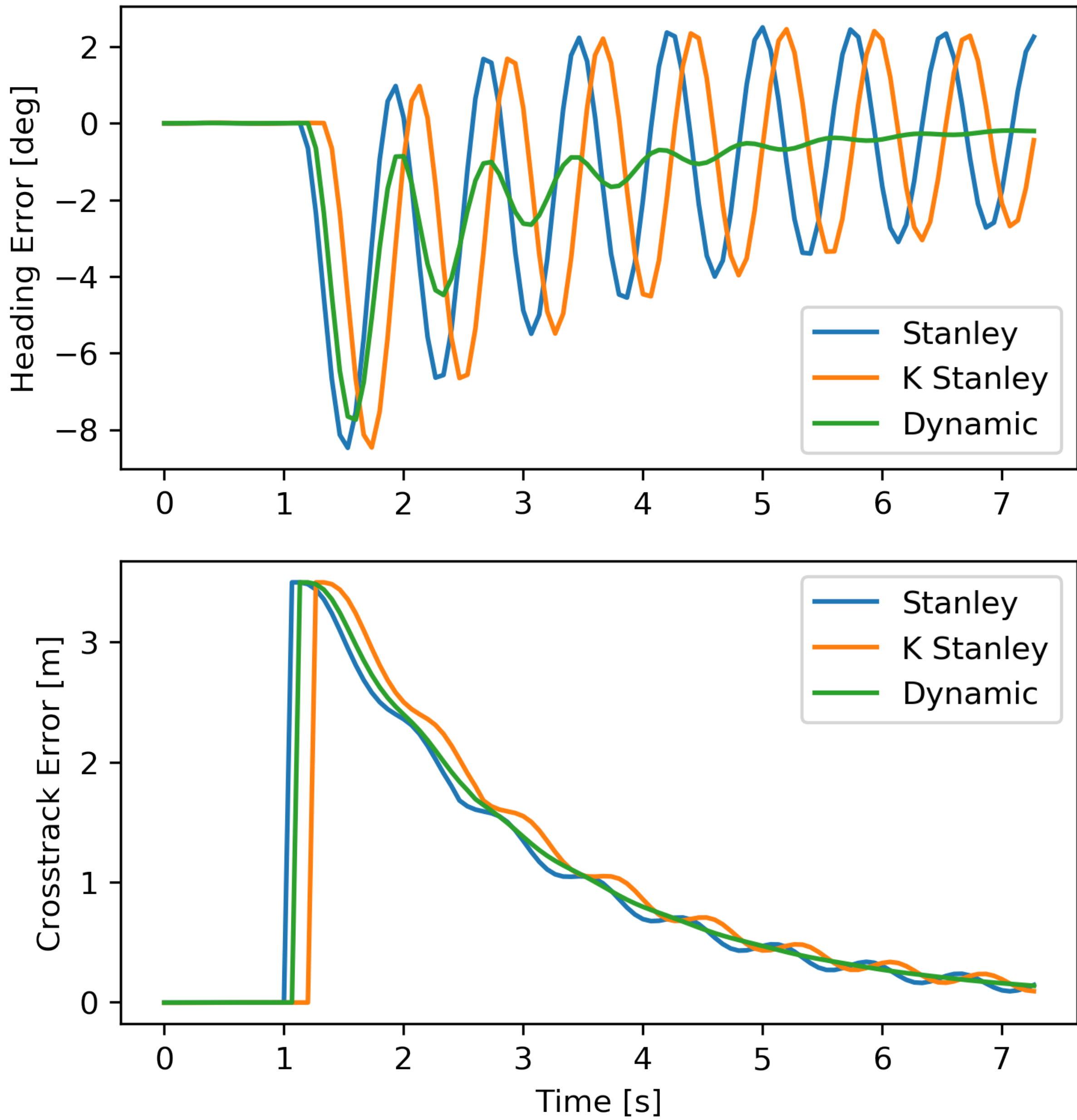


Figure 5.12: Lane deviation trace during and after a high speed obstacle avoidance maneuver using the three different controllers

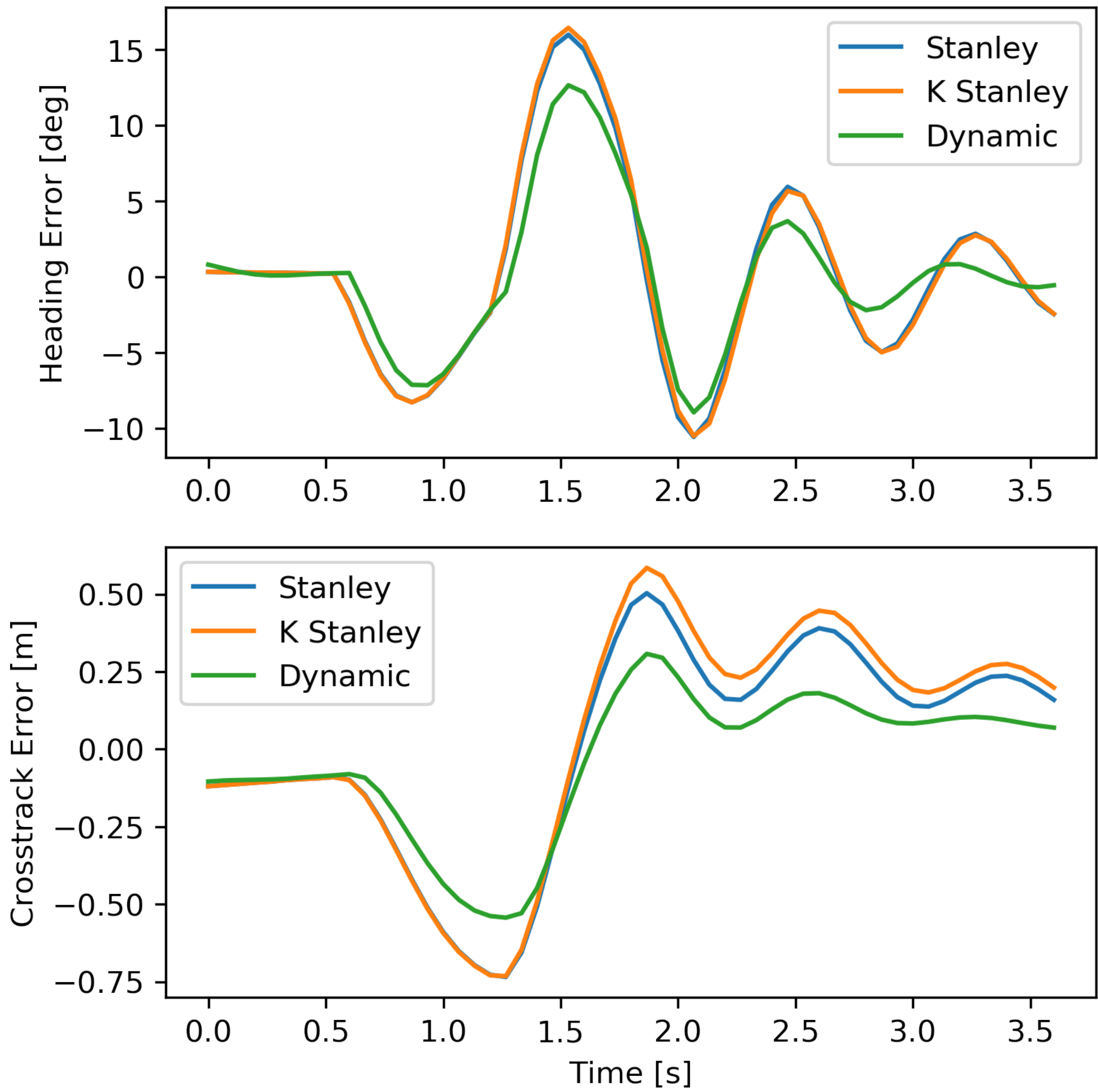


Figure 5.13: Lane deviation trace during and after a lane change using the three different controllers

Chapter 6

Conclusion

6.1 Summary

This thesis presents three main directions of research and their results:

- End-to-end Imitation Learning
- Direct Perception
- Curvature Based Dynamic Controller

The purpose of this thesis is to explore the two less common approaches in autonomous driving, imitation learning and direct perception (Figure 4.1). As a refresher, a qualitative comparison between them is shown in Figure 1.1. Imitation learning, or end-to-end control, is a control strategy that attempts to develop a policy that maps sensory stimuli to control actions. On a self driving car, the policy receives sensor readings and converts them to driving signals such as steering angles and longitudinal acceleration requests. Supervised learning is used to develop the policy, with training data coming from expert (human or algorithmic) demonstrators. Algorithmic experts are simulation agents that have access to more resources than the policy during inference, such as ground truth lane line and obstacle locations. Direct perception also starts with using a machine learning model to process sensor inputs and ends with control outputs, but it is not an end-to-end approach because the learning based component's sole task is to predict affordances, which are then used by a controller to generate driving signals for the vehicle. Affordances are variables

that hold key information about completion of the task. The difference between mediated perception, the most common paradigm, and direct perception is that direct perception does not reconstruct the surrounding environment to the extent that mediated perception does. The predefined affordance set should include everything the controller needs to operate, and is typically only several variables. To conduct closed loop simulation testing of the direct perception model, vehicle controllers were used to drive the vehicle based on these affordances.

6.1.1 Imitation Learning

Driving is a multimodal task in the sense that for the same sensory stimulus, there can be multiple correct actions to take (Figure 2.2). To build an end-to-end policy that accommodates this, it needs to be aware of the intentions of the controlling entity, be it a passenger or a GPS. Should the vehicle turn left or right at the next intersection? The first approach was to provide this intention to the network as an input. Discrete intention classes, or high level commands, are fed to the network in one-hot form along with each sensor snapshot. The training data contains these intentions (follow lane, left turn, right turn, straight), and the policy is trained to output appropriate actions given the current scenario and the current intention. The second approach is to use a network with branched outputs, with each branch representing an intention. The training process for this branched network uses all data to update the shared portion, or the trunk, of the network. The branches are only updated by data that have the appropriate intention by using the intention as a mask during loss calculation. Such a network simplifies the control task by having each branch only output actions possible under its own intention. The branched architecture performed better than the non-branched architecture, and the main advantage shown in testing was that the former was much less likely to have infractions on intention (i.e. turning left after being told to turn right).

6.1.2 Direct Perception

Direct perception uses an affordance predictor to predict affordances and a controller uses these affordances to control the system. In the case of controlling a car, the affordances need to be selected such that a controller can use them to safely drive the vehicle down the road. For this thesis, the affordances were crosstrack error, heading error, lane curvature, and distance to the closest car in front. The first three are used for lateral control, and the

last is used for longitudinal control. The potential advantage of direct perception is that it can be more explainable than imitation learning, and have less complexity than mediated perception. In addition, the system can easily be conditioned for different behaviors by altering the controller, something that is unfeasible with imitation learning.

To create the data needed to train an affordance predictor, a simulation based data recorder was built. Since a new dataset is being created, it was decided to add the left and right lane changes to the list of intention labels to make the final product much more complete of a driver. Human demonstrators drove the simulated ego vehicle while relaying their intentions to the data recorder. The demonstrators were instructed to drive in specific ways to avoid data imbalance in the labels and to break the correlation between action and intention.

6.1.3 Controller

Two curvature based controllers were proposed for lateral control. The first, K-Stanley, uses geometric relations between a bicycle model and path curvature to directly solve for the steering angle that would lead the vehicle to match a given curvature. The second, (name), uses a dynamic bicycle model to predict future path curvature and uses the gradient of the prediction calculation to optimize for the desired steering angle. Both use the Stanley controller to compensate for heading and crosstrack errors. Limitations for the dynamic controller due to its linear tire model was explored.

6.2 Future Work

The imitation learning and direct perception approaches to autonomous driving are still in their early stages of development and there is a long way to go before commercial viability. The upside is that there is potential for expansion in many different directions of research, and they all present interesting challenges that once a solution is found for, can be generalized to a wide range of applications outside of autonomous driving. Much was learned in the process of designing, coding, testing, debugging, and paper reading for the work presented in this thesis, and below are some ideas for continued research that would advance the field the most or patch its most glaring flaws.

Safety Guarantee

The machine learning models developed in this thesis work as intended in validation. However, there is no formal proof that they will work in production environments. Unlike rules based controllers where there is a clear relationship between input and output and stability can easily be proven, the possible set of inputs for these models are large enough that it is difficult to test everything. How can we be sure that a minor disturbance in the input sensor readings doesn't send the car driving towards the sidewalk? There has been some work on proving safety and stability of neural network policies [8, 16], but a general and definitive methodology has not been found.

Conditional Modeling

The driving task is conditional, meaning there can be multiple valid actions in the same scenario. This thesis handled the conditional nature of the driving task by either making navigation intention an input to the model, or using branched models to represent different intentions. The branching method was found to be an reliable way to get conditional outputs from the model. However, branches can only support a set number of discrete intentions. The direct perception network proposed in this thesis contains follow lane, left and right turn, and left and right lane change intentions. The addition of lane change made it much more capable than the imitation learning model, which itself was limited by the dataset used. However, it still cannot support obscure road networks and driving scenarios. For example, at a 5 way intersection, even with an intention to turn, there are multiple valid ways to go. Defining more intentions by adding branches is an inelegant solution because scenario data needed to train the lesser used branches are rare and a high number of branches is needed if we want to cover everything. More work can be done on finding an efficient way to provide intention in continuous space to the model, rather than discrete classes.

Affordance Selection

The type of controller that can be used in a direct perception pipeline is limited by the choice of affordances. Even though the affordances used in this thesis enabled safe operation of the vehicle, there has not been any work on systematically comparing the pros and cons of different affordance sets. In this thesis, heading and crosstrack errors were selected because together they fully define the current position error of the vehicle, and they can easily be

used by a controller to generate steering angles. However, they don't say much about the future error of the vehicle, so advanced controllers such as MPC or LQR cannot be used. Road curvature was added to the affordance set for this reason. Curvature defines a constant curvature path, or an arc, and this allows the use of the predictive controller derived in section 5.2. Low level path planners usually plan far enough ahead that the reference path cannot be accurately modeled by an arc. A useful expansion on this work is to train an affordance predictor to output a polynomial or a spline curve to better represent the road ahead.

In terms of obstacle perception, the affordance set in this thesis only contains a single distance to the vehicle in front, and this allows the ego vehicle to have adaptive cruise control like capabilities. Yet this is not sufficient for a self driving car. The vehicle is blind to obstacles beside or behind it, and perception of these areas is needed for safe lane changing or backing up. One way to solve this is to train a similar affordance predictor for the sides and the back, and a controller uses the outputs from all the predictors to drive the car. Alternatively, with sensors that provide complete coverage of the car's surroundings, sensor fusion can be used to paint a complete picture of surrounding obstacles. A single predictor can be trained to fuse data from all sensors to output a complete set of affordances. Instead of distance values in all four directions of the car, the affordance predictor can have a decoder head that outputs a top down semantic segmentation view that can be used by a prediction module and a path planner to obtain an optimal path down the road. Although this crosses into mediated perception territory, a good middle ground should be found between complexity and interpretability.

Path Planning

As they stand, all approaches developed for this thesis cannot yet guide vehicles in ways that a human driver can. For example, if there is a double parked vehicle, they will wait and not know to drive around. This is because the models were only trained to keep the vehicle in its lane, and there is no real time path planning to navigate around obstacles. With a dataset that includes these types of edge case behaviors, conditional imitation learning can be used to train a model that behaves the same way. To get a direct perception model up to this standard, a planning module needs to be more closely integrated. The direct perception pipeline in this thesis uses discrete intentions passed from an assumed preexisting path planner. In future work, making an actual path an input to the affordance predictor in rasterized or vector form could be tested.

Structure from Motion

A rather distant yet relevant method, structure from motion, can be compared or integrated with the approaches from this paper. Structure from motion is the process of estimating 3D geometry from a series of 2D images. The approach is wildly different, however the inputs and outputs are very similar. The direct perception method presented in this thesis also takes in image sequences. Instead of producing a 3D environment, it might be possible to adapt structure from motion techniques such as Direct Sparse Odometry [22] to output affordances, or to combine the two for ensemble learning.

References

[1] Turing-nlg: A 17-billion-parameter language model by microsoft, Feb 2020.

[2] Alexander Amini, Guy Rosman, Sertac Karaman, and Daniela Rus. Variational end-to-end navigation and localization. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8958–8964. IEEE, 2019.

[3] Khalid Ashmawy. Searchable ground truth: Querying uncommon scenarios in self-driving car development, Oct 2019.

[4] Aharon Azulay and Yair Weiss. Why do deep convolutional networks generalize so poorly to small image transformations? *Journal of Machine Learning Research*, 20(184):1–25, 2019.

[5] Aditya Babhulkar. Self-driving car using udacity's car simulator environment and trained by deep neural networks. 2019.

[6] Zhengwei Bai, Baigen Cai, Wei ShangGuan, and Linguo Chai. Deep learning based motion planning for autonomous vehicle using spatiotemporal lstm network. In *2018 Chinese Automation Congress (CAC)*, pages 1610–1614. IEEE, 2018.

[7] Mayank Bansal, Alex Krizhevsky, and Abhijit Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *arXiv preprint arXiv:1812.03079*, 2018.

[8] Felix Berkenkamp, Matteo Turchetta, Angela Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *Advances in neural information processing systems*, pages 908–918, 2017.

[9] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[10] Adith Boloor, Karthik Garimella, Xin He, Christopher Gill, Yevgeniy Vorobeychik, and Xuan Zhang. Attacking vision-based perception in end-to-end autonomous driving models. *Journal of Systems Architecture*, page 101766, 2020.

[11] Rogerio Bonatti, Ratnesh Madaan, Vibhav Vineet, Sebastian Scherer, and Ashish Kapoor. Learning controls using cross-modal representations: Bridging simulation and reality for drone racing. *arXiv preprint arXiv:1909.06993*, 2019.

[12] Valentino Braitenberg. *Vehicles, Experiments in Synthetic Psychology*. 1984.

[13] Paula Branco, Luís Torgo, and Rita P Ribeiro. Smogn: a pre-processing approach for imbalanced regression. In *First International Workshop on Learning with Imbalanced Domains: Theory and Applications*, pages 36–50, 2017.

[14] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[15] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. *arXiv preprint arXiv:1912.12294*, 2019.

[16] Jianyu Chen, Bodi Yuan, and Masayoshi Tomizuka. Model-free deep reinforcement learning for urban autonomous driving. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2765–2771. IEEE, 2019.

[17] Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9. IEEE, 2018.

[18] Felipe Codevilla, Eder Santana, Antonio M López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. *arXiv preprint arXiv:1904.08980*, 2019.

[19] R Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, 1992.

[20] Luca Cultrera, Lorenzo Seidenari, Federico Becattini, Pietro Pala, and Alberto Del Bimbo. Explaining autonomous driving by learning end-to-end visual attention. In *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[21] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[22] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry. *IEEE transactions on pattern analysis and machine intelligence*, 40(3):611–625, 2017.

[23] Maayan Frid-Adar, Idit Diamant, Eyal Klang, Michal Amitai, Jacob Goldberger, and Hayit Greenspan. Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification. *Neurocomputing*, 321:321–331, 2018.

[24] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[25] Hege Haavaldsen, Max Aasboe, and Frank Lindseth. Autonomous vehicle control: End-to-end learning in simulated urban environments. *arXiv preprint arXiv:1905.06712*, 2019.

[26] Hansson. Steering angle prediction by a deep neural network and its domain adaption ability. 2018.

[27] Bar Hilleli and Ran El-Yaniv. Deep learning of robotic tasks without a simulator using strong and weak human supervision. *arXiv preprint arXiv:1612.01086*, 2016.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] Christopher Innocenti, Henrik Lindén, Ghazaleh Panahandeh, Lennart Svensson, and Nasser Mohammadiha. Imitation learning for vision-based lane keeping assistance. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 425–430. IEEE, 2017.

[30] Mark Rosekind Jesse Levinson. Safety Innovation At Zoox. Technical report, Zoox, 2018.

[31] Chris Clover Jim Bernard. Tire modeling for low-speed and high-speed calculations. 104, 7 1995.

[32] Martin Kamran, Junyi Zhu, and Martin Lauer. Learning path tracking for real car-like mobile robots from simulation. 2019.

[33] Michael Kelly, Chelsea Sidrane, Katherine Driggs-Campbell, and Mykel J Kochenderfer. Hg-dagger: Interactive imitation learning with human experts. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8077–8083. IEEE, 2019.

[34] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[35] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[36] Bartosz Krawczyk. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence*, 5(4):221–232, 2016.

[37] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[38] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[39] Henrik Lindn, Ghazaleh Panahandeh, Lennart Svensson, et al. Imitation learning for vision-based lane keeping assistance. *arXiv preprint*, 2017.

[40] Rooz Mahdavian and Richard Diehl Martinez. Ignition: An end-to-end supervised model for training simulated self-driving vehicles. *arXiv preprint arXiv:1806.11349*, 2018.

[41] Rod McCall, Fintan McGee, Alexander Mirnig, Alexander Meschtscherjakov, Nicolas Louveton, Thomas Engel, and Manfred Tscheligi. A taxonomy of autonomous vehicle handover situations. *Transportation research part A: policy and practice*, 124:507–522, 2019.

[42] Marvin Minsky. *The Society of Mind*. 1986.

[43] Hans Moravec. *Mind Children*. 1990.

[44] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L Cun. Off-road obstacle avoidance through end-to-end learning. In *Advances in neural information processing systems*, pages 739–746, 2006.

[45] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, Jan Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018.

[46] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntak Lee, Xinyan Yan, Evangelos Theodorou, and Byron Boots. Agile autonomous driving using end-to-end deep imitation learning. In *Robotics: science and systems*, 2018.

[47] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[48] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[50] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments. *arXiv preprint arXiv:1806.06498*, 2018.

[51] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[52] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[53] Chen Sun, Jean M Uwabeza Vianney, and Dongpu Cao. Affordance learning in direct perception for autonomous driving. *arXiv preprint arXiv:1903.08746*, 2019.

[54] Remo MA van der Heiden, Shamsi T Iqbal, and Christian P Janssen. Priming drivers before handover in semi-autonomous cars. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 392–404, 2017.

[55] Jason Wang and Luis Perez. The effectiveness of data augmentation in image classification using deep learning.

[56] Kyle Wiggers. Waymo's autonomous cars have driven 20 million miles on public roads, Jan 2020.

[57] Jin Xu, Kui Yang, YiMing Shao, and GongYuan Lu. An experimental study on lateral acceleration of cars in different environments in sichuan, southwest china. *Discrete Dynamics in nature and Society*, 2015, 2015.

[58] Jiakai Zhang and Kyunghyun Cho. Query-efficient imitation learning for end-to-end autonomous driving. *arXiv preprint arXiv:1605.06450*, 2016.

APPENDICES

Appendix A

Key Code Snippets

A.1 Branch Training Code

Below is an example of a script used to train a branched network. It is called with command line arguments of paths to the dataset and the configuration file which dictates training parameters.

```
import matplotlib
matplotlib.use('Agg')

import os
import sys
import time
from shutil import copyfile

import matplotlib.pyplot as plt
import argparse
import numpy as np

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
```

```
from model import CDPNet, combined_loss
from dataset import DPDataset
from filehelp import date_string
from readconfig import readconfig

def save_train_history(train, val, figname, val_freq):
    train_line, = plt.plot(train, label='Training Loss')
    val_line, = plt.plot(val, label='Validation Loss')
    plt.xlabel(' x {} Batches'.format(val_freq))
    plt.ylabel('Loss')
    plt.legend()
    #plt.ylim((-2,6))
    plt.savefig(figname)

def main():
    '''get parameters'''
    func_dict, dims_dict, scales_dict, params_dict = readconfig(INIPATH)

    '''setup data'''
    train_set = DPDataset(
        DPATH,
        scales_dict,
        seq_len=int(params_dict['seq_len']),
        separation=int(params_dict['separation'])
    )
    train_loader = DataLoader(train_set,
    batch_size=int(params_dict['batch']),
    shuffle=True,
    drop_last=True)

    val_set = DPDataset(DPATH,
    scales_dict=scales_dict,
    seq_len=int(params_dict['seq_len']),
    val=True)
    val_loader = DataLoader(val_set,
    batch_size=int(params_dict['batch']),
```

```
shuffle=True,
drop_last=True)
iter_val = val_loader.__iter__()

'''setup model'''
model = CDPNet(funcs_dict=func_dict, dims_dict=dims_dict)
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
if torch.cuda.device_count() > 1:
    print('Using {} GPUs'.format(torch.cuda.device_count()))
    model = nn.DataParallel(model)
    #print(isinstance(model, nn.DataParallel))
model.to(DEVICE)
opt = torch.optim.RMSprop(model.parameters(), lr=params_dict['lr_init'])
lr_plateau = torch.optim.lr_scheduler.ReduceLROnPlateau(opt,
factor=0.5, patience=10, cooldown=5, min_lr=1e-6)

'''setup checkpoint directory'''
weights_path = './weights'
if not os.path.exists(weights_path):
    os.mkdir(weights_path)
save_path = os.path.join(weights_path,date_string())
os.mkdir(save_path)

config_name = os.path.split(INIPATH)[-1]
copyfile(INIPATH, os.path.join(save_path, config_name))
print('saving weights to', save_path)

'''training'''
train_loss_history = []
val_loss_history = []
lr_history = []
val_freq = 10
save_freq = 100
total_batches = len(train_loader)

start = time.time()
```

```
try:
    for e in range(params_dict['epoch']):
        #epoch loop
        for b, batch in enumerate(train_loader):
            #batch loop
            #unpack training data, move to GPU
            X = batch[0]
            y = batch[1]
            X = [x.float().to(DEVICE) for x in X]
            y = [wy.float().to(DEVICE) for wy in y]

            #forward pass
            pred, hidden_states = model(X)

            #loss calc, update weights
            total_loss, s_loss, p_loss = combined_loss(pred,
                y,
                X[2][:,-1,:]
            )
            total_loss.backward()
            opt.step()
            opt.zero_grad()

            #checkpoint
            if (b%save_freq==0):
                if b!=0:
                    if isinstance(model, nn.DataParallel):
                        sd = model.module.state_dict()
                    else:
                        sd = model.state_dict()

                    checkpoint_dir = os.path.join(save_path,
                        'e{}b{}.pt'.format(e,b)
                    )
                    torch.save(sd, checkpoint_dir)
```

```
#validation
if (b%val_freq==0) and b!=0:
    model.eval()
    try:
        X_val, y_val = next(iter_val)
    except StopIteration:
        #if we reach the end of validation data
        iter_val = val_loader.__iter__()
        X_val, y_val = next(iter_val)

    X_val = [x.float().to(DEVICE) for x in X_val]
    y_val = [wy.float().to(DEVICE) for wy in y_val]

    with torch.no_grad():
        pred_val, hidden_val = model(X_val)
        loss_val, s_loss_val, p_loss_val = combined_loss(
        pred_val,
        y_val,
        X_val[2][:,-1,:]
    )

    val_loss_history.append(loss_val.item())
    train_loss_history.append(total_loss.item())

    print('e {}, b {}/{}, \
    loss: {:.3f}, s_loss: {:.3f}, \
    p_loss: {:.3f}, \
    val loss: {:.3f}'.format(e,
    b, total_batches,
    total_loss, s_loss, p_loss, loss_val))

    model.train()


#if b>=1:
    #break
```

```
    finally:
        print('Training took {:.3f} seconds'.format(time.time()-start))
        if isinstance(model, nn.DataParallel):
            sd = model.module.state_dict()
        else:
            sd = model.state_dict()
        checkpoint_dir = os.path.join(save_path, 'final.pt')
        torch.save(model.state_dict(), checkpoint_dir)

        fig_path = os.path.join(save_path, 'loss.png')
        save_train_history(train_loss_history,
        val_loss_history, fig_path, val_freq)
        os.system('nvidia-smi')



if __name__=='__main__':
    #get data path and config path
    DPATH = sys.argv[1]
    INIPATH = sys.argv[2]
    print('Training with parameters from', INIPATH)
    main()
```