Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Predicting Ego-Vehicle Speed From Monocular Dash-Cam Video in Diverse Conditions

Samuel Sutherland-Dee

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 15th May, 2020

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author. This project did not require ethical review, as determined by Dr. Dima Damen, the supervisor of the Author. The only data used in this project was taken from a publicly available dataset.

Samuel Sutherland-Dee, Friday 15th May, 2020

Contents

List of Figures

List of Tables

List of Listings

Abstract

The notion of fully autonomous personal vehicles has shifted from a dream to an inevitability in recent years, bringing a host of technical and ethical challenges. Clearly, the technology in these vehicles is mission-critical: any failure could have disastrous consequences. The conditions in which autonomous vehicles might operate are incredibly diverse, including the weather, scene and time of day; the on-board systems must therefore be robust to many possible environments. The technology behind autonomous driving is a very active area of research, particularly the deep learning solutions to perception and planning problems. One of the most critical pieces of information in an autonomous vehicle is its current speed; any miscalculation can affect both trajectory calculation and over or under braking, potentially resulting in unnecessary accidents.

This project aims to discover whether a deep learning solution, specifically a Convolutional Neural Network, can be used to predict the ego-vehicle speed from monocular dash-cam footage, using no other sensor input. Such a standalone system is more robust to failure of individual components in other areas of the vehicle. To predict speed from video, some estimate of the absolute scale of the scene is necessary; traditional computer vision approaches for monocular localisation and ego-motion estimation are accurate only up to scale. The absolute scale cannot be easily estimated without prior knowledge of the scene and, even when an estimate is made, it can drift over time. As a starting point, a CNN architecture proposed by Nvidia for autonomous vehicles is reimplemented and optimised, using techniques that have proven popular and successful in deep learning research. Additionally, the dense optical flow computed between two consecutive frames, encoded into an RGB image, is chosen as the input to the CNN.

The results presented in Chapter 3 and 4 show that our proposed CNN greatly outperforms the Nvidia architecture in all conditions. Further analysis shows that our network is not perfect, specifically in relation to the prediction of high speeds; it is proposed that the cause of this issue is the optical flow estimation. This project offers a first step towards further study of ego-vehicle speed estimation using deep learning methods.

The main contributions of this project are as follows:

- A pipeline to extract speed data from Berkeley Deep Drive videos with the corresponding frames, calculate the optical flow, and upload to Blue Crystal 4 (BC4).
- A reimplementation of the Nvidia CNN architecture for autonomous driving.
- Optimisation of the Nvidia architecture, training over 80 models in the process (Chapter 3) and totalling over 400 hours of training.
- Comparison of the Nvidia architecture and our proposed improvements (Chapter 4).
- An array of scripts for analysing the results.

Supporting Technologies

The following technologies were used to support the development of this project:

- The Python 3 programming language is used for all development, specifically the following packages:
  - `PyTorch` (https://pytorch.org/) for constructing the Convolutional Neural Network.
  - `OpenCV` (https://opencv.org/) for calculating the optical flow between frames and image utilities.
  - `ffmpeg-python` (https://github.com/kkroening/ffmpeg-python) for extracting rotation metadata from video files.
  - `NumPy` (https://numpy.org/) for various mathematical operations.
  - `Matplotlib` (https://matplotlib.org/) for plotting and saving graphs.
  - `SciPy` (https://www.scipy.org/) for generating statistics.
  - `Pexpect` (https://pexpect.readthedocs.io/) for running scripts.
  - `PIL` (https://pillow.readthedocs.io/) for reading and writing images.
- The CNN models were trained on BC4.

Notation and Acronyms

Notation

$\mathbf{A}$	:	Matrix
$\mathbf{b}$	:	Vector
c	:	Scalar
$\mathbf{A}^\top, \mathbf{b}^\top$	:	Matrix/Vector Transpose
$\mathbf{A}^{-1}$	:	Matrix Inverse

Acronyms

BC4	:	Blue Crystal 4
CNN	:	Convolutional Neural Network
GPS	:	Global Positioning System
GPU	:	Graphics Processing Unit
HSV	:	Hue, Saturation, Value
IMU	:	Inertial Measurement Unit
LiDAR	:	Light Detection And Ranging
SGD	:	Stochastic Gradient Descent
SLAM	:	Simultaneous Localisation and Mapping

Acknowledgements

I would firstly like to thank my supervisor, Dr. Dima Damen, for her valuable advice and feedback during this project. In addition, I thank Michael for acting as a very sturdy sounding board throughout. Many thanks to my parents for keeping me fed and watered during the gestation of this dissertation. Finally, I would like to thank Christine for her unwavering support and encouragement.

Chapter 1

Contextual Background

1.1 Introduction

This project aims to discover whether Convolutional Neural Networks (CNNs) can be used to predict the speed of a vehicle from on-board dash-cam footage alone. If it is possible to predict the speed of the vehicle, to a sufficient accuracy, then such a system could be used as a fail-safe in case of Inertial Measurement Unit (IMU) failure or the degradation of Global Positioning System (GPS) accuracy in urban canyon environments. An urban canyon environment is described as when GPS signals are blocked by two high-rise buildings on either side of the street, such that there are too few satellite signals to accurately estimate positioning information [22]; this is a common problem in densely packed cities that are growing ever taller [3]. To counter urban canyon environments, it is necessary for the system to be accurate in cities and residential areas. To act as a fail-safe in case of IMU failure, the system must be robust to all kinds of weather, scene and time of day.

There are some assumptions that are made about the practical set-up of the system: the only input to the system during operation is monocular dash-cam footage, the system must be robust in an unknown environment, the system must not integrate other sensor recordings and the camera must not require calibration before every journey.

1.2 A Brief History of Autonomous Driving

The adoption of fully autonomous cars has been a dream for many futurists since the 1930's, imagining a car delivering passengers from their front door to their desired destination without human intervention. Alongside reducing accidents and congestion, such vehicles have the potential to transform cities entirely, eliminating the need for parking and allowing for a much more pedestrian friendly environment. For several decades after, it was not at all obvious how an autonomous car could be built. There was little to none of the digital sensing and computing equipment required to make it a reality.

Attention turned to road networks in the 1930's, with the construction of the Autobahn in Germany that provided a new road system to reduce accidents and increase speeds; the Autobahn famously has no federally mandated speed limit. The Autobahn is an environment void of certain obstacles that cause difficulty in autonomous navigation, such as pedestrians and blind corners. American futurist Norman Bel Geddes took the control systems present in railroads and applied them to the context of motorways, presenting his results at the 1939 World's Fair. He imagined lanes that would keep cars apart in their respective tracks on the motorway until they turn off, at which point the driver takes control [35]. Contemporary alternatives included magnets in the bottom of cars tracking steel wires in the roads and steel wheels on the inside of each tire engaging train-like rails laid in the road.

Progress towards a true autonomous car would prove to be incremental rather than revolutionary. It was clear that computing power would require significant improvement in order to process the vast amounts of information produced by sensors. The Stanford Artificial Intelligence Laboratory constructed a cart with a front facing camera, eventually enabling it to navigate new environments. Development progressed further in the 1980's, with Ernst Dickmanns equipping a Mercedes van to drive automatically at up to 90 km/h along the Autobahn [4], an incredible achievement at the time. Similar results were achieved

by the ARGO Project in Italy during the 1990's [16].

Figure 1.1: Stanford Racing with Victor Tango at an intersection during the 2007 DARPA Urban Challenge.

In 2004, the US Defense Advanced Research Projects Agency (DARPA) conducted the Grand Challenge; a competition for a ground vehicle to autonomously navigate 142 miles across a Nevada desert with a $1 million cash prize awaiting the winner [2]. The aim was to eventually develop a ground vehicle capable of eliminating the need for human drivers in potentially hazardous situations during war, such as supply convoys. Many universities and companies collaborated in the competition, however none finished the course; the highest scoring vehicle only travelled 7.5 miles. A second challenge in 2005 concluded with 5 of the 195 teams finishing and Stanford University winning the $2 million prize. The leader of the Stanford team eventually went on to found Google X and their autonomous car effort. The third and final challenge was conducted in 2007, in which the cars navigated an urban environment, negotiating other moving traffic and obstacles whilst obeying California traffic regulations. Only 6 of the 11 teams managed to complete the course and 'Team Tartan', led by Carnegie Mellon University, took the $2 million prize [79]. From this point on the proliferation of industrial research into autonomous cars turned these vehicles from a dream into an inevitability.

1.3 Speed Estimation

Humans are incredibly adept at estimating ego-motion and the 3D structure of a scene, even when using only one eye - the monocular case. It is likely that we develop a structural understanding of the world from past visual experiences, helping to infer 3D structure of new scenes from monocular views by recognising objects: a road is flat and buildings are straight. In terms of ego-motion, optical flow is a key cue [81]; we are able to determine our direction of travel - even when it differs from the direction of vision [85] - and our absolute speed [84] from optical flow. Of course, the human visual system is incredibly complex and much remains unknown, but it provides a good starting point for how to approach the task of speed estimation from monocular video. Optical flow is presented in detail in the next chapter, but in short: optical flow is the apparent movement of objects, surfaces and edges between two frames due to the relative motion between the camera and the scene.

It is reasonable to question why the calculation of optical flow and the training of a CNN is necessary for speed estimation. If the length of a lane marking is known, then the speed of the vehicle can easily be determined using the cross-ratio of two points along this line in two frames, forming four co-linear points [40]. To make a robust and useful system, however, we cannot assume we have any prior knowledge about the scene; in any case, lane markings vary between roads and can be occluded in traffic. This is a primitive method; perhaps instead, speed could be determined using a proven method such as Simultaneous Localisation and Mapping (SLAM). SLAM is the process by which a mobile robot can estimate the 3D structure of an unknown environment (mapping) whilst at the same time using this map to compute its own location (localisation); it is a well-studied task in robotics and computer vision [29]. In short, image features are matched between subsequent frames, using these correspondences to estimate the pose of the camera for each frame and the 3D position of the features. Other sensor data, such as LiDAR, can

also be incorporated into this system. Indeed, most SLAM implementations rely on other sensor data or stereovision approaches, both of which are not applicable for this project. MonoSLAM, proposed by Davison et al. [23], can recover the 3D trajectory of a monocular camera in real-time. The problem with this approach is that the reconstruction of the scene is only accurate up to scale; the absolute scale of the scene would be required to estimate the true speed of the vehicle. This requires some prior knowledge about the scene. To recover the true scale, MonoSLAM proposes calibrating the scale of the camera every time the system is turned on. This is not practical for our use case as it would be unreasonable to ask the owner of the vehicle to stand in front of the forward-facing camera with a checkerboard before every journey. During operation, this scale can drift over time and result in very poor speed estimations. Several techniques to remedy this include the detection of objects with known, or approximately known, dimensions to deduce the absolute scale [14, 34, 33], however the scale will still drift when no such objects are detected. In the case of speed estimation in diverse conditions, the number of objects that could be detected is prohibitively large; for example, there is enormous variation in signage and model of car. In addition, the objects used to estimate scale must be stationary. This may cause problems on highways where there are very few features to begin with and all other vehicles in the scene are moving, not to mention that objects may be occluded or not detected for a significant period of time.

With the problem of scale in mind, using a CNN is an interesting exercise to discover whether it is possible to overcome this inherent geometric problem. Using optical flow as the input to the CNN informs relative motion of the vehicle and the scene; it is up to the network to determine how this relates to absolute speed. It is shown in later chapters that - given enough training data - the CNN is able to predict speed surprisingly well.

1.4 Autonomous Driving Research

The technology that supports autonomous driving is an extremely active area of research, both in academia and industry. All of the major international computer vision conferences (CVPR, ICCV, ECCV) have held, or intend to hold, a workshop on autonomous driving [12, 9, 10, 11]. A significant number of major car manufacturers are sponsoring or directly pursuing autonomous driving research. Large technology companies are also heavily investing in autonomous vehicles, such as Google, Uber and Lyft - all taking a significant hit to their balance sheet as a result. Due to the intense competition between these companies, much of their technology is kept secret, although many actively engage with academia through the aforementioned workshops and the publication of some of their research.

The tasks involved in building a fully autonomous driving system can be broadly separated into three categories: perception, planning and control. Perception is how the vehicle understands its environment: where the vehicle is in the environment, what objects are nearby and where these objects are. Planning involves anticipating the movements of other objects in the scene and using this knowledge to plan the movement of the vehicle to avoid collisions, whilst navigating to the correct location. Control translates the instructions of the planning systems into steering, acceleration, braking and indication actuation. The task of speed estimation is classed as perception and so some other perception tasks are briefly described. These perception tasks can again be broadly split into single-frame tasks and multi-frame tasks; the majority being single-frame. The datasets that support autonomous driving research are then catalogued, noting the major trends and differences. For a comprehensive review of both the problems and state of the art in computer vision for autonomous vehicles, please see the work of Janai et al. [48].

1.4.1 Sensors

To understand how the tasks of perception and planning are completed in practice, it is important to be aware of a key concept in autonomous driving: sensor fusion. Sensor fusion is the combining of readings from different sensors to give more robust information, by exploiting their characteristics and reducing the reliance on a single sensor. This concept is not so applicable to speed estimation but especially important to object detection, localisation and planning. Autonomous vehicles are, in general, fitted with an extended suite of sensors, including cameras, Radar and Light Detection and Ranging (LiDAR). For example, Tesla's autopilot system is fitted with an array of cameras that provide 360° coverage, short range ultrasonics and a longer range Radar sensor at the front of the vehicle (Figure 1.2). This system is somewhat of an outlier in that it does not make use of the LiDAR readings that are heavily utilised in

other autonomous driving systems.

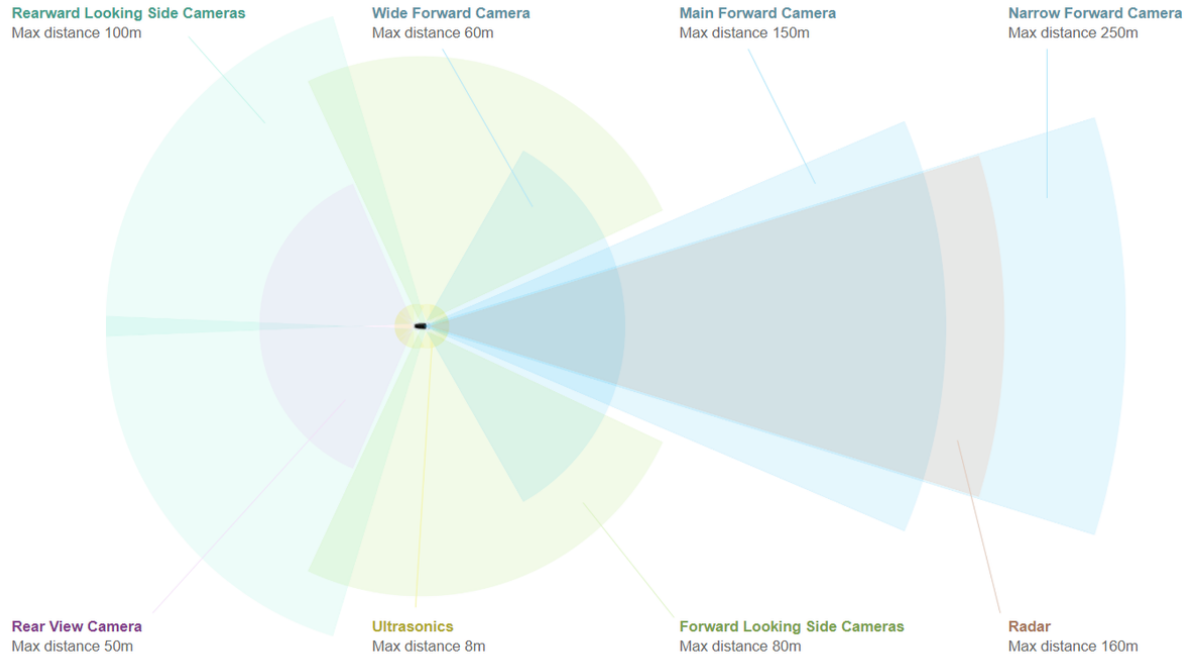


Figure 1.2: The sensor suite of Tesla Autopilot [7].

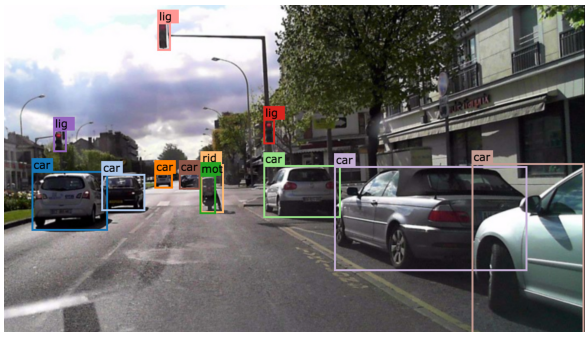
LiDAR measures the reflections of laser beams, emitted at a certain frequency, to give accurate depth estimation in the form of 3D points; it is more robust to different lighting and weather conditions than cameras, but the points become sparse at long distances and cannot capture detailed texture. Radars function in much the same way, emitting radio waves instead of laser light. Although Radar has a longer range than LiDAR, it suffers from much poorer resolution.

1.4.2 Tasks

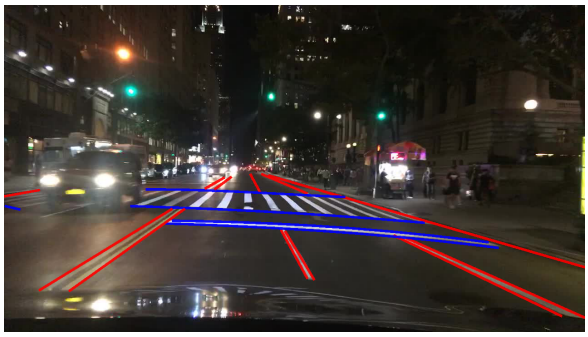
Object Detection and Tracking

Object detection is a fundamental computer vision problem involving two tasks: the classification of objects in an image (the what) and the localisation of these objects (the where). The localisation is usually in the form of an x-axis aligned bounding box. An example of object detection on a road scene can be seen in Figure 1.3a. While object detection is not new [66, 80], deep learning (specifically CNNs) has revolutionised the area, resulting in far more accurate and robust predictions [70]. However, object detection is not just limited to 2D images; for autonomous vehicles, 3D object detection is much more relevant, but a more ambitious problem since the depth of objects must also be estimated. Most approaches to 3D object detection employ LiDAR for accurate depth estimation which, when combined with images, prove to be very successful [19, 83]. These approaches tend to be very involved, however a general pipeline for fusing 2D object detections of vehicles with a 3D point cloud to produce 3D bounding boxes was proposed by Du et al. [27].

In object detection, each frame is usually processed independently with no associations over time. For autonomous vehicles to estimate the trajectories of other vehicles and therefore prevent collisions, objects must be tracked between frames. Object tracking, as an extension of object detection, proposes new challenges: the pose of objects may change over time, objects may come in and out of occlusion and objects may simply look very similar. The most obvious solution to object tracking is to apply an object detection algorithm to each frame individually, identifying the objects and turning the problem into bounding box association, so-called tracking-by-detection [73, 74]. This method has the drawbacks of relying heavily on accurate object detections and ignoring other features in the image during association. A newer trend has been to combine detection and tracking into the same framework [32]; one such method



(a) 2D Object Detection (BDD [86]).



(b) Lane Marking (BDD [86]).

Figure 1.3: 2D Object Detection and Lane Marking examples on different road scenes in BDD.

CenterTrack [87] places second on the KiTTi [36] leaderboard for multi-object tracking of cars.

Lane Marking

Lane marking detection and, more generally, drivable area detection are extremely important parts of an autonomous driving system, aiding in trajectory planning and vehicle localisation. Lane marking detection algorithms must be robust to all conditions, with wet roads providing the greatest challenge. While older solutions rely on handcrafted features [50, 24], deep learning has once again proven to be the most successful approach. VPGNet [57] additionally predicts the vanishing point, noting that lane markings usually converge on the horizon. However, the main contribution of VPGNet is its performance in low illumination and rainy conditions, achieving high accuracy in real time (20fps). An example of lane marking detection can be seen in Figure 1.3b.

Semantic and Instance Segmentation

Semantic segmentation is another fundamental problem in computer vision; the goal is to assign each pixel in the image a label from a predefined set of categories (e.g. road, pavement, car, pedestrian). The result of semantic segmentation can be seen in Figure 1.4a, each colour corresponding to a unique label; for example the pavements are pink and the sky is lighter blue. This segmentation allows the vehicle to better understand the scene, but is complicated by intricate object boundaries, small objects and sheer complexity of the scene. Semantic segmentation is a very complex problem and much beyond the scope of this introduction, but deep learning methods have, unsurprisingly, delivered the best results. Many of the best systems employ a residual network [43] in some form, usually for feature extraction; the best performing approach on the KiTTi dataset uses a residual network, tackling the boundary problem by allowing multiple classes to be predicted for the same pixel during training [89]. It is worth noting that there are many strategies for solving semantic segmentation, most of which are highly involved [48].

(a) Semantic Segmentation (Mapillary Vistas [5]).

(b) Instance Segmentation (BDD [86]).

Figure 1.4: Semantic and Instance Segmentation of different scenes.

As an alternative to semantic segmentation assigning each pixel a label, instance segmentation aims to assign each pixel to a unique object. This is illustrated in Figure 1.4b, noting that each car has been

assigned a different colour. The approaches to instance segmentation are broadly split into two categories: proposal-based [71] and proposal-free. Proposal-based methods consist of two steps, identifying instances of objects and then classifying them, whereas proposal-free methods predict pixel labels directly from the image. The top performing method for instance segmentation on the KiTTi dataset is Mask R-CNN [42], however this is outperformed by PolyTransform [59] on the Cityscapes [21] dataset. Instance segmentation and semantic segmentation can be combined into one task called panoptic segmentation [54], providing information about the position, semantics, shape and count of all objects. All of this information is much more useful to an autonomous vehicle than just semantic segmentation, but the task of extracting this information is far more involved.

1.4.3 Datasets

There are several datasets that have been built to support autonomous driving research and in particular the tasks described in this chapter. Some datasets focus on visual data, such as Berkeley Deep Drive (BDD) [86] and Cityscapes [21], whereas others focus on LiDAR and sensor fusion, such as Apolloscape [82] and nuScenes [17]. Many of these datasets provide labelled data for tasks that have not been described - mostly localisation and mapping - however, the datasets and the tasks are supported by them are listed in Table 1.1.

Dataset	Object Detection		Object Tracking		Lane Marking	Segmentation	
	2D	3D	2D	3D		Semantic	Instance
ApolloScape [82]		✓		✓	✓	✓	
Argoverse [18]		✓		✓	✓		
Astyx [62]		✓					
Audi [37]		✓				✓	
BDD [86]	✓		✓		✓	✓	✓
Cityscapes [21]						✓	✓
KiTTi [36]	✓	✓	✓	✓		✓	✓
Lyft [52]		✓			✓		
Mapillary Vistas [5]						✓	
nuScenes [17]		✓		✓			
Waymo [78]	✓	✓	✓	✓			

Table 1.1: Tasks with labelled data provided by each dataset.

We use BDD as the source of the data used to train and test our CNN for the following reasons: the video is recorded at a good frame rate (30fps), BDD is designed to be diverse in scene, weather and time of day, the data is labelled in a straightforward format and is pre-split into training, validation and testing.

The aim of this project is to predict the ego-vehicle speed from monocular dash-cam footage, using no other sensor data or prior information about the scene. In autonomous driving, this task would be classed as perception; the vehicle is attempting to calculate its state. What is hopefully apparent in the description of these other common perception problems is that all of the state of the art solutions are deep learning techniques. This is an indication that approaching the problem of speed estimation with deep learning is a useful exercise and one that should be explored. That is what this project attempts to do; discover whether a CNN can predict ego-vehicle speed, something that is non-trivial in classical computer vision when no other information about the scene is available.

Chapter 2

Technical Background

This chapter aims to inform the reader of the technical background required to understand the subsequent chapters on the execution and critical evaluation of this project. An optical flow image is used as the input to the CNN proposed in Chapter 3; it is therefore necessary to understand what optical flow is and what it means, as well as see the Farnebäck algorithm that is utilised to calculate the optical flow. CNNs are then described, noting the main operations that set them apart from fully connected neural networks. The loss function used to train the parameters of the CNN, Huber Loss, is then described, noting advantages over similar loss functions. Neural networks are susceptible to overfitting on training data, limiting the generalisation of the network; what this means and some of the regularisation techniques used to reduce this effect are described.

2.1 Optical Flow

Optical flow is the apparent movement of objects, surfaces and edges between two frames due to relative motion between the camera and the scene. Optical flow aims to estimate the 2D motion field arising from the projection of the actual 3D motion in the scene and works on the following assumptions:

1. The pixel intensities of an object do not change over a small window of time.
2. Neighbouring pixels have a similar motion.

To derive the optical flow equation, let's assume that we are attempting to estimate the optical flow between two frames recorded at times t and $t + dt$. These frames are close together in time such that dt is small. Consider a pixel with intensity $I(x, y, t)$ at position (x, y) in the first frame at time t. This pixel moves by a distance of (dx, dy) from the first frame to the second frame such that the pixel intensity now has the notation $I(x + dx, y + dy, t + dt)$ in the second frame. Since these pixels are the same and, from the first assumption, the pixel intensities of an object do not change over a small window of time, the following equation can be made:

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \tag{2.1}$$

Taking a truncated Taylor approximation of the right hand side of Equation 2.1 results in:

$$I(x + dx, y + dy, t + dt) \approx I(x, y, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt \tag{2.2}$$

From Equation 2.1 it follows that:

$$\frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt = 0 \tag{2.3}$$

Finally, by dividing by dt we end up with the optical flow equation:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{2.4}$$

where $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$ are the pixel velocities in the horizontal and vertical axis respectively. $\frac{\partial I}{\partial x}$, $\frac{\partial I}{\partial y}$ and $\frac{\partial I}{\partial t}$ correspond to the image gradients along the horizontal axis, vertical axis and time respectively, all

of which can be estimated from the frames. Therefore (u, v) are the unknown variables, yet they cannot be directly solved from the optical flow equation alone as there are two unknowns and only one equation.

There are some cases where the assumptions made to generate the optical flow equation result in a poor estimate of the 2D motion field. Imagine a rotating Lambertian sphere such that the apparent brightness is invariant to the angle of view. With a static light source, the pixel intensities are constant in the frame and therefore a static motion field is estimated, yet the sphere is rotating. In contrast, if the light source rotates around a stationary sphere, it would appear as though the sphere is still rotating by looking at the pixel intensities in isolation. Indeed, there are further problems with estimating the 2D motion field, especially when looking at one pixel alone or a small group of pixels that form an edge. When looking through an aperture at a moving line, such that the ends of the line are not visible, it is only possible to estimate the normal flow of the motion. This effect is illustrated in Figure 2.1 in which the lines could be moving in any of the directions indicated by the arrows such that the movement would appear the same by only looking through the aperture.

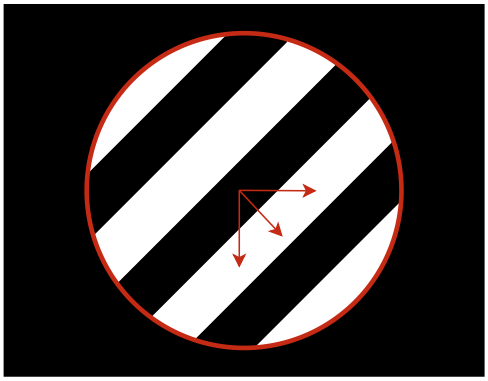


Figure 2.1: The lines in this aperture could be moving in any one of the candidate directions indicated by the arrows.

As previously mentioned, the optical flow equation is under-constrained when observing one pixel. Most techniques therefore rely on the second assumption at the beginning of this section: neighbouring pixels have similar motion. By looking at a region of pixels and assuming that the velocity of motion is constant over this region, one optical flow equation per pixel in the region can be formed. If the region is larger than two pixels then the optical flow equation is over-constrained and can be solved in a least squares fashion. The simplest solution is the constant velocity model, to find the velocity $\mathbf{v} = (v_x, v_y)$ which minimises:

$$\epsilon(v_x, v_y) = \sum_{\text{region}} (I_x v_x + I_y v_y + I_t)^2 \tag{2.5}$$

where $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$ and $I_t = \frac{\partial I}{\partial t}$. This equation can be solved by differentiating with respect to v_x and v_y and setting both equations to zero:

$$\frac{\partial \epsilon(v_x, v_y)}{\partial v_x} = \sum_{\text{region}} ({I_x}^2 v_x + I_x I_y v_y + I_x I_t) = 0 \tag{2.6}$$

$$\frac{\partial \epsilon(v_x, v_y)}{\partial v_y} = \sum_{\text{region}} (I_x I_y v_x + {I_y}^2 v_y + I_y I_t) = 0 \tag{2.7}$$

Now there are two linear equations for $\mathbf{v} = (v_x, v_y)$ which can be represented in matrix form as:

$$\mathbf{A}\mathbf{v} - \mathbf{b} = 0 \tag{2.8}$$

where

$$\mathbf{A} = \begin{pmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} -I_x I_t \\ -I_y I_t \end{pmatrix} \tag{2.9}$$

Solving for $\mathbf{v}$ yields:

$$\mathbf{v} = \mathbf{A}^{-1}\mathbf{b} \tag{2.10}$$

This method was first proposed by Lucas and Kanade in 1981 [60] and remains a popular method to calculate optical flow, particularly sparse optical flow. Sparse optical flow concerns the tracking of a sparse set of distinctive features, such as corners or edges, through frames. Dense optical flow, however, calculates a motion vector for up to every pixel in the frame. The trade-off between sparse and dense optical flow is between computational cost and coverage of the motion field. For this project, the natural reaction is to prefer dense optical flow, calculated offline before the CNN is trained, providing greater coverage from which features can be extracted. Therefore dense optical flow will be discussed further and sparse optical flow details omitted from this technical background.

2.1.1 Farnebäck Optical Flow

Farnebäck optical flow is a two-frame motion estimation algorithm based on polynomial expansion, introduced by Gunnar Farnebäck in 2003 [31]. This section will step through the algorithm itself and motivate its use in this project. The first step of the algorithm is to approximate each neighbourhood of pixels in both frames by quadratic polynomials, using the polynomial expansion transform. By assuming that the motion field is slowly varying, a method to estimate such motion fields from the polynomial expansion coefficients under translation is derived. Including a priori knowledge, iteration and multi-scale motion field estimation results in a robust algorithm.

To begin, each neighbourhood of pixels is approximated by a quadratic polynomial f:

$$f(\mathbf{x}) \sim \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c \tag{2.11}$$

where $\mathbf{A}$ is a symmetric matrix, $\mathbf{b}$ a vector and c a scalar. The coefficients of this polynomial are estimated from a weighted least squares fit to the pixels in the neighbourhood. Farnebäck gives the weighting two components called certainty and applicability, which are the same as in normalised convolution. In short, it is a good idea to set the certainty to zero outside of the image and use applicability to weight the centre point of the neighbourhood the most, then decrease the weights radially. Further details can be found in [30].

Now that each neighbourhood of pixels is approximated by a polynomial, we can examine the effects of a polynomial undergoing an ideal global translation. The goal is to relate two polynomials between frames under an ideal translation to estimate the actual motion. Consider the exact quadratic polynomial:

$$f_1(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}_1 \mathbf{x} + \mathbf{b}_1^\top \mathbf{x} + c_1 \tag{2.12}$$

Globally displacing f_1 by $\mathbf{d}$ results in a new signal f_2:

$$\begin{aligned} f_2(\mathbf{x}) = f_1(\mathbf{x} - \mathbf{d}) &= (\mathbf{x} - \mathbf{d})^\top \mathbf{A}_1 (\mathbf{x} - \mathbf{d}) + \mathbf{b}_1^\top (\mathbf{x} - \mathbf{d}) + c_1 \\ &= \mathbf{x}^\top \mathbf{A}_1 \mathbf{x} + (\mathbf{b}_1 - 2\mathbf{A}_1\mathbf{d})^\top \mathbf{x} + \mathbf{d}^\top \mathbf{A}_1 \mathbf{d} - \mathbf{b}_1^\top \mathbf{d} + c_1 \\ &= \mathbf{x}^\top \mathbf{A}_2 \mathbf{x} + \mathbf{b}_2^\top \mathbf{x} + c_2 \end{aligned} \tag{2.13}$$

Equating the coefficients yields:

$$\mathbf{A}_2 = \mathbf{A}_1 \tag{2.14}$$

$$\mathbf{b}_2 = \mathbf{b}_1 - 2\mathbf{A}_1\mathbf{d} \tag{2.15}$$

$$c_2 = \mathbf{d}^\top \mathbf{A}_1 \mathbf{d} - \mathbf{b}_1^\top \mathbf{d} + c_1 \tag{2.16}$$

Using Equation 2.15, we can solve for the displacement $\mathbf{d}$, if $\mathbf{A}_1$ is non-singular:

$$2\mathbf{A}_1\mathbf{d} = -(\mathbf{b}_2 - \mathbf{b}_1) \tag{2.17}$$

$$\mathbf{d} = -\frac{1}{2}\mathbf{A}_1^{-1}(\mathbf{b}_2 - \mathbf{b}_1) \tag{2.18}$$

So far we have assumed that the entire neighbourhood of pixels can actually be approximated by a single polynomial and that two neighbourhoods of pixels in two frames can be related by a global translation.

Clearly, these assumptions are not realistic for an actual sequence of frames, yet Equation 2.18 can still be used for real signals, although some errors are introduced. The aim of Farnebäck was to minimise these errors such that the algorithm remains useful. Firstly, the global polynomial in Equation 2.12 is replaced with local polynomial approximations. Taking a polynomial expansion of both images yields expansion coefficients of $\mathbf{A}_1(\mathbf{x})$, $\mathbf{b}_1(\mathbf{x})$ and $c_1(\mathbf{x})$ for the first image and $\mathbf{A}_2(\mathbf{x})$, $\mathbf{b}_2(\mathbf{x})$ and $c_2(\mathbf{x})$ for the second. Ideally, as in Equation 2.14, $\mathbf{A}_1 = \mathbf{A}_2$ but in practice it is necessary to make the approximation:

$$\mathbf{A} = \frac{\mathbf{A}_1(\mathbf{x}) + \mathbf{A}_2(\mathbf{x})}{2} \tag{2.19}$$

A similar approach is taken to introduce:

$$\Delta\mathbf{b}(\mathbf{x}) = -\frac{1}{2}(\mathbf{b}_2(\mathbf{x}) - \mathbf{b}_1(\mathbf{x})) \tag{2.20}$$

obtaining the primary constraint:

$$\mathbf{A}(\mathbf{x})\mathbf{d}(\mathbf{x}) = \Delta\mathbf{b}(\mathbf{x}) \tag{2.21}$$

with $\mathbf{d}(\mathbf{x})$ indicating that the motion field is now spatially varying. Equation 2.21 could be solved pointwise although results turn out to be too noisy. The assumption that the displacement field only varies slowly enables the integration of information over a neighbourhood of each pixel, increasing the robustness of the algorithm. Therefore, over a neighbourhood I around $\mathbf{x}$ the aim is to find the $\mathbf{d}(\mathbf{x})$ that minimises:

$$\sum_{\Delta\mathbf{x}\in I} w(\Delta\mathbf{x}) \left\| \mathbf{A}(\mathbf{x}+\Delta\mathbf{x})\mathbf{d}(\mathbf{x}) - \Delta\mathbf{b}(\mathbf{x}+\Delta\mathbf{x}) \right\|^2 \tag{2.22}$$

where $w(\Delta\mathbf{x})$ is a weight function for the points in the neighbourhood. Therefore the minimum is obtained for:

$$\mathbf{d}(\mathbf{x}) = \left(\sum_{\Delta\mathbf{x}\in I} w(\Delta\mathbf{x})\mathbf{A}(\mathbf{x}+\Delta\mathbf{x})^\top \mathbf{A}(\mathbf{x}+\Delta\mathbf{x}) \right)^{-1} \left(\sum_{\Delta\mathbf{x}\in I} w(\Delta\mathbf{x})\mathbf{A}(\mathbf{x}+\Delta\mathbf{x})^\top \Delta\mathbf{b}(\mathbf{x}+\Delta\mathbf{x}) \right) \tag{2.23}$$

where a unique solution exists if the whole neighbourhood is not exposed to the aperture problem.

The robustness of the algorithm can be further improved if the motion field can be parameterised according to some existing motion model. If the motion model is linear in its parameters, such as the affine model or the eight parameter model, then this is a simple procedure. The derivation for the eight parameter model in 2D is worked through in Farnebäck's paper [31]. Indeed, by again incorporating existing knowledge, the robustness can be improved, this time by a priori knowledge of the motion field. So far it has been assumed that the local polynomials at the same coordinates in the two frames are identical apart from a displacement. These polynomial expansions tend to vary spatially since they are local models and will therefore introduce errors. This is less of a problem for small displacements than it is for larger ones. By introducing a priori knowledge about the motion field, the polynomial at $\mathbf{x}$ in the first frame can be compared with the polynomial at $\mathbf{x} + \tilde{\mathbf{d}}(\mathbf{x})$, where $\tilde{\mathbf{d}}$ is the value of the a priori displacement field at coordinate $\mathbf{x}$. Then it is possible to only estimate the displacement between the a priori estimate and the real value in the second frame.

The inclusion of an a priori estimate into the algorithm allows for iteration; a better a priori estimate reduces the relative displacement and therefore improves the chances that the algorithm produces a good motion estimate. Two approaches are suggested: iterative and multi-scale. The estimate from one step is used as the a priori estimate for the next step in both approaches, with the a priori estimate usually set to zero for the first iteration. For the first approach the polynomial expansion coefficients are calculated only once, rendering iteration futile if the displacements are too large after the first iteration. Such large displacements can be dealt with by conducting the analysis at a coarser scale by using a larger applicability or region for the polynomial expansion. By handling larger displacements we are punished with reduced accuracy. The accuracy can be improved by the multi-scale approach; by iterating from a coarse to a more fine grained estimation, the displacements can be propagated up to the finer scales to obtain increasingly accurate estimates. This process can be computationally expensive since the polynomial expansion coefficients must be recalculated at each scale. This algorithm has an implementation

in OpenCV that allows for significant parameter customisation for the number of levels, window sizes, number of iterations and more.

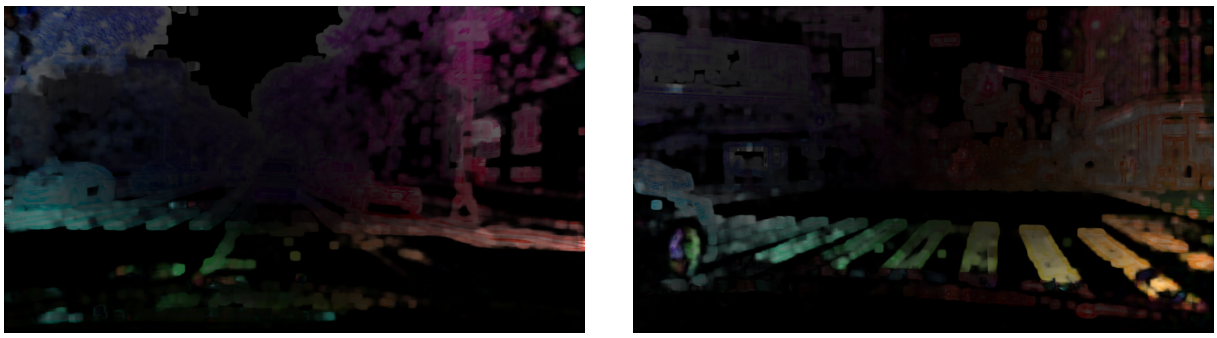


Figure 2.2: Example outputs of Farnbäck optical flow.

Examples of the output of the Farnebäck optical flow estimation algorithm, applied to two consecutive frames, are visualised in Figure 2.2. The optical flow is encoded into the Hue Saturation Value (HSV) colour space such that the **hue** represents the angle of the optical flow, the **saturation** is that of the second source image and the **value** represents the normalised magnitude of the optical flow. It is worth noting that the saturation is maintained from the source image so that intricate features are still present within the image for the CNN to potentially identify.

2.2 Convolutional Neural Networks

It is assumed that the reader has obtained knowledge, from another source [39], on the basics of machine learning including the properties and functionality of a feed-forward neural network with multiple layers. In addition, the gradient-based optimisation of a multi-layer neural network is assumed knowledge. This section will introduce CNNs and the operations commonly implemented that are specific to such networks, including the convolution operation and pooling.

CNNs are specialised for processing data that has a known grid-like topology. The most obvious example is image data as pixels are naturally arranged into a discrete 2D grid and will be the focus of this section. Another example is time series data, which can be thought of as a 1-D grid, taking samples at regular time intervals. CNNs were first proposed by the prominent Deep Learning researcher Yann LeCun in 1989 for optical character recognition on handwritten zip codes [56] and have since become extremely successful at a wide variety of tasks. The name Convolutional Neural Network indicates that the network utilises a mathematical operation called **convolution** which is now introduced.

2.2.1 The Convolution Operation

In mathematics, convolution is an operation on two functions of a real-valued argument, producing a third function expressing how the shape of one is modified by the other. How this definition relates to CNNs seems unintuitive at first, yet the convolution operation employed in CNNs is a specific form of mathematical convolution.

To begin, suppose we are measuring the speed of a vehicle using an onboard sensor that provides an output $x(t)$, the speed at time t. Now suppose that when constructing this car, we used a sensor that we received for free from a scrapyard. It is likely that such a sensor would produce noisy results and to combat this we would like to average several measurements over time. Clearly, more recent measurements should contribute more and thus we would like this to be a weighted average that gives more weight to more recent measurements. This can be achieved by applying a weighting function $w(a)$, where a is the age of the measurement at every instant. We end up with the function s, producing a smoothed estimate of the speed of our vehicle:

$$s(t) = \int x(a)w(t-a)da \tag{2.24}$$

This operation is convolution and is usually denoted with an asterisk:

$$s(t) = (x * w)(t) \tag{2.25}$$

In this example, the function x is typically referred to as the **input**, the function w as the **kernel** and the output as the **feature map**. When introducing CNNs, it was noted that they are specialised for grid-like data and more importantly, discrete grids. We would thus like to end up with a discrete definition of convolution from Equation 2.24. Suppose that our speed sensor actually takes a measurement every second, a more reasonable assumption than continuous reading. The time index t can then take on only integer values, one integer for each second. We can now define discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.26}$$

Finally, since this project involves applying a CNN to images, we would like to expand this discrete convolution to two dimensions. The previous example can only go so far, suppose we now have an image I as input and two dimensional kernel K. The convolution now becomes:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.27}$$

This convolution operation can be visualised in Figure 2.3 using a small image and kernel, restricting the output to only positions where the kernel lies completely within the image. To prevent cutting off the edges of the input, several methods are available with the most common being padding; PyTorch pads with zeroes by default.

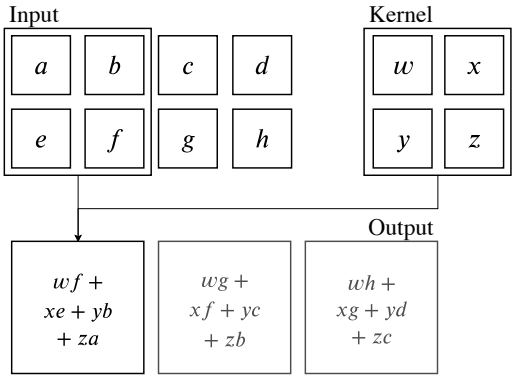


Figure 2.3: Convolution with a 2x2 kernel.

Many machine learning libraries implement a similar function called **cross-correlation**, which is the same as convolution without flipping the kernel:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.28}$$

This is not usually an issue when training a CNN; a flipped, yet equivalent, kernel will be learned relative to a CNN trained with convolution. Learning such a flipped kernel will have no effect on the output of the network.

Traditional neural networks have a separate parameter for the interaction between every input unit and every output unit. Convolution within a CNN, on the other hand, represents sparse interactions between the inputs and outputs by using kernels that are smaller than the input. This provides benefits both in terms of the smaller memory requirement and the lower computational cost of calculating the output. It is common to employ kernels that are orders of magnitude smaller than the input. Units in the deeper layers of a CNN will combine the sparse interactions of previous layers to build an efficient description of a larger part of the input to the network. The amount of input to the network that a unit can 'see' is referred to as the receptive field of the unit; deeper layers have a larger receptive field and operations like strided convolution and pooling can increase the receptive field of shallow units.

It is worth noting that for each convolutional layer in a CNN, it is highly desirable that many features are extracted at many locations; one kernel can learn only one feature. Thus convolution in a CNN refers to many convolutions in parallel. In addition, when RGB images are used as input to a CNN, convolution is applied separately to each of the three colour channels and subsequently the input and output of convolution are 3D tensors. CNNs are commonly trained in batch mode such that samples are grouped together into one 4-D tensor, with 4-D convolution implemented in many neural network libraries.

2.2.2 Pooling

Pooling is typically implemented as the third operation within a convolutional layer, after the convolutions and a non-linear activation. A pooling function adds some invariance to small translations in the output of the layer by replacing the output at a certain location with a summary statistic of the nearby outputs. This is useful for determining the presence of a feature instead of the exact location of such feature. For some networks it may be more desirable to activate if a feature is in a specific location and therefore pooling is not as useful. Pooling is commonly implemented with downsampling, as shown in Figure 2.4, helping to reduce the size of the network and improve computational efficiency.

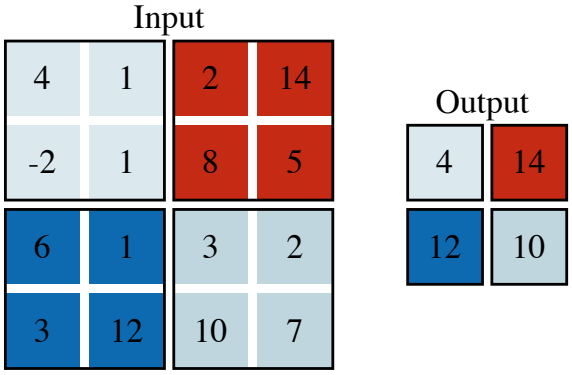


Figure 2.4: Max pooling with a kernel size of 2x2 and a stride of 2x2.

The most popular pooling function is max pooling [88], which outputs the maximum value present within a rectangular grid. An example of this operation can be seen in Figure 2.4. Other popular pooling functions include the average of a rectangular neighbourhood, the L^2 norm of a rectangular neighbourhood and a weighted average based on the distance from the central pixel. Even if max pooling is the most popular form of pooling, it may not be the most optimal for the given situation [15].

2.3 Loss Functions

The algorithm used to train neural networks, backpropagation, is an optimisation algorithm: the parameters of the model are trained to minimise an objective function, or loss function. The loss function acts to evaluate the performance of the entire model and boil it down to one scalar value. Therefore, the choice of loss function is dependent on the purpose and output of the model; the output for a classification task is a discrete list of class probabilities whereas a regression task (such as this project) outputs a continuous value. For a classification task, a loss function such as cross entropy loss [49] would be appropriate. The network proposed in this project, however, outputs a single value: the predicted speed of the vehicle. These predicted speeds must be appropriately evaluated against the ground truth speeds in each batch. One of the most common loss functions for regression tasks in deep learning is Mean Squared Error (MSE), simply defined as:

$$\text{MSE}\left(\mathbf{x}, \mathbf{y}\right) = \frac{1}{N} \sum_i \left(x_i - y_i\right)^2 \tag{2.29}$$

where N is the size of the batch, $\mathbf{x}$ are the predictions and $\mathbf{y}$ are the ground truths, respectively. MSE is simply the mean squared difference between the prediction of the model and the ground truth; the shape of this function is plotted as the red line in Figure 2.5. The problem with MSE is that it is extremely sensitive to outliers: it grows extremely quickly with error and as such, one large error in a batch can

skew the entire loss of a batch. This is a particular problem for this project, specifically the diversity of the training set and therefore the large number of outliers that it contains. These outliers end up skewing the learning of the model in such a way that it is worse when evaluated on the testing set. The nature of outliers are that they are rare occurrences and thus unlikely to be present in the testing set.

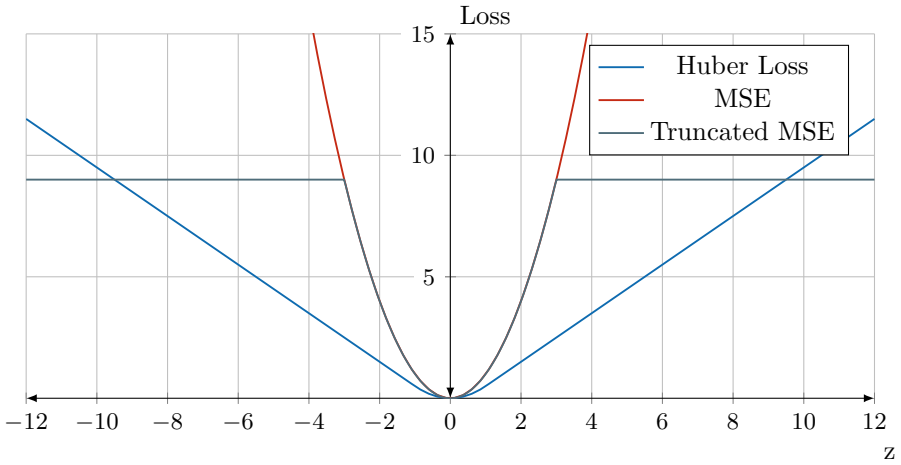


Figure 2.5: Huber Loss ($\delta = 1.0$), Mean Squared Error (MSE) and Truncated MSE ($\delta = 3.0$).

Truncated loss functions [67] offer one solution to a lack of robustness: by capping the maximum error for one sample at a predetermined constant, it limits the effect of outliers on the final loss. A truncated version of MSE loss can be written as:

$$\text{Truncated MSE}_\delta(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i \min\left(\delta, (x_i - y_i)^2\right) \tag{2.30}$$

where δ is the constant that places an upper bound on the squared error. Truncated MSE is clearly more robust than MSE, but comes with the complication of selecting the best value for the maximum squared error δ. This selection is non-trivial: a δ that is too small will result in a portion of data contributing almost nothing to the loss, but a δ too large will reduce the robustness to outliers. Another candidate for a robust loss function is L_1 loss, which replaces the squared error in MSE with absolute error. For large errors, L_1 loss clearly produces much lower values than MSE, so is more robust to outliers. The sticking point with L_1 loss occurs during gradient descent optimisation around 0; a constant gradient means that the optimisation does not scale with the distance to a minima and is therefore likely to repeatedly flip between positive and negative, never reaching the minima. To improve this behaviour around minima and maintain robustness to outliers, Huber Loss uses a squared term for very small errors and an absolute term otherwise. This is plotted in blue in Figure 2.5 and written as follows:

$$\begin{aligned} \text{Huber Loss}_\delta(\mathbf{x}, \mathbf{y}) &= \frac{1}{N} \sum_i z_i \\ z_i &= \begin{cases} \frac{1}{2}(x_i - y_i)^2 & \text{if } |x_i - y_i| < \delta \\ \delta |x_i - y_i| - \frac{1}{2}\delta & \text{otherwise} \end{cases} \end{aligned} \tag{2.31}$$

where δ defines the maximum error that is squared, commonly set to 0.5. Training with Huber loss is more stable than MSE and results in the lowest testing errors with our final network.

2.4 Regularisation of Neural Networks

Regularisation is a central problem in machine learning; that is, how to construct an algorithm that generalises well to data not present in the training set. This usually takes the form of a reduction in test loss, potentially at the expense of a higher training loss. Many techniques to achieve regularisation are available and these techniques are an active area of research. For the sake of brevity, only the techniques that will be discussed in Chapter 3 are described here, namely dropout, batch normalisation and data augmentation.

2.4.1 Dropout

One method used in machine learning to improve robustness during inference is to average the predictions of a number of models, aptly named model averaging. This works due to the fact that these models will usually not all make the same errors on the test set. Such an ensemble of models could all be different in terms of network architecture and loss function or could be trained on different datasets. Clearly, if all the errors are correlated, then model averaging does not help. If all the models make independent errors then model averaging will perform significantly better than any one of the members. On average, an ensemble of models will perform as well as any of the models alone. The drawback with this method lies in the fact that, to get sufficient performance from each of the models, it is likely that they are deep and have a large number of parameters. This in turn increases the computational cost during inference to an unwieldy level beyond five or ten models, not to mention the increased training time and memory cost.

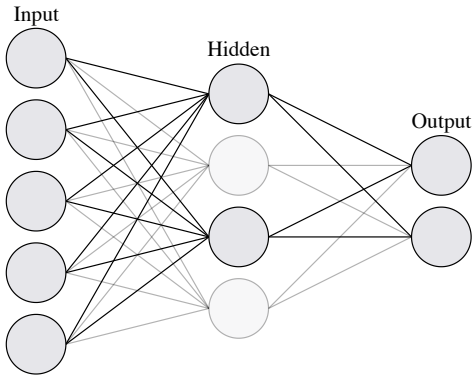


Figure 2.6: A forward pass during training of a neural network with dropout in the hidden layer ($p = 0.5$).

Dropout provides a less expensive approximation to training and evaluating an ensemble of exponentially many networks. Implemented on the level of units, a dropped unit refers to a unit that has been removed from the network, usually by multiplying the outputs of the unit by zero. By dropping some non-output units during training, dropout in effect trains the ensemble consisting of all subnetworks that can be formed from dropping such units. First proposed by Srivastava et al. [76], dropout has proven to be a powerful regularisation technique.

Specifically, the choice of which units to drop is a random one; each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or simply set at 0.5. This is implemented as a vector of independent Bernoulli random variables, each with a probability p of being 1, for each layer. This vector is then sampled and multiplied element-wise with the outputs of the layer that is being dropped out. A retention probability of 0.5 is close to optimal for a wide range of networks as tasks [76] although it is noted that for input units, the optimal probability is usually closer to 1. All units are retained at test time, however if a unit is retained with probability p during training, the outgoing weights are multiplied by p at test time. This ensures that the expected output, under the distribution used to drop units, is the same as the actual output.

2.4.2 Batch Normalisation

During the training of a deep neural net, the distribution of each layer's inputs is altered as the parameters of the previous layers change. This is known as internal covariate shift and has the effect of slowing down training; requiring lower learning rates and careful initialisation. It would therefore be desirable if these distributions did not change during training. A normalisation procedure is implemented for each mini-batch, giving name to the technique Batch Normalisation, introduced by Ioffe and Szegedy [47]. By fixing the distribution of inputs to have zero mean and unit variance, training speed improves as the network converges faster.

Consider the inputs to one unit x for a mini-batch $\mathcal{B} = \{x_1, x_2, ..., x_m\}$, the mean $\mu_\mathcal{B}$ and variance $\sigma^2_\mathcal{B}$ of the mini-batch are simply:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.32}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{2.33}$$

Therefore the normalisation procedure involves subtracting the mean and dividing by the root of the variance:

$$\hat{x_i} \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{2.34}$$

where $\hat{x_i}$ is the new normalised input to the layer and ϵ is a small constant for numerical stability. However, simply normalising each input of the layer may alter the representational power of the layer. To prevent this, a pair of parameters, γ and β are learned for each activation. These are then used to scale and shift the normalised value as so:

$$y_i \leftarrow \gamma\hat{x_i} + \beta \tag{2.35}$$

Note that the conventional bias in the layer is now superfluous since the effect is cancelled by the mean subtraction and replaced by the learned parameter β. Batch Normalisation has the effect of enabling higher learning rates and shortening the training time. In addition, Ioffe and Szegedy note that, from the results of their experiments, Batch Normalisation can help the generalisation ability of the network. This is due to the training network no longer producing deterministic values for a given training sample and that these samples are seen in conjunction with other examples in the mini-batch due to the normalisation procedure.

2.4.3 Data Augmentation

For obvious reasons, it is not possible to train our model on all the data that it could possibly encounter, such that the resultant test error is always 0. It remains, however, that the best way to improve the performance of a model is to supply it with more training data. Data augmentation is the process of applying various transforms to training data so that the model is more invariant to such transforms that might occur in unseen data. When working with images, the most common transforms include flipping, cropping, rotation, scaling, translation and brightness augmentation.

It is important for the transform to preserve the label of the data, for example in optical character recognition it would not be desirable to rotate a "6" by 180° such that it has the appearance of a "9", yet keeps the label of 6. These transformations are usually performed online, that is during training time when the samples are picked from the training set. It would also be possible to inflate the training set size by performing several transformations to the existing training data and saving the results into a new training set, so called offline augmentation.

The goal of the transformations performed in data augmentation is to increase the variance of the dataset. By training online, a sample might be transformed in a different way at each stage of training, therefore increasing the diversity of the dataset. This diversity can be increased further by applying transforms one after the other in a stochastic manner.

Chapter 3

Project Execution

This chapter aims to explain and justify the decisions made during the development of our final CNN used to predict ego-vehicle speed. The use of optical flow as the input to the CNN is briefly discussed, noting some drawbacks of optical flow for this task. The origin and the contents of the training and testing sets are then outlined; the dataset being arguably the most important part of any deep learning system. The final CNN architecture is then presented having been empirically optimised for the task at hand, breaking down the improvements to the original Nvidia architecture one by one. The training procedure for our CNN is then outlined: the loss function, gradient optimisation algorithm and hyperparameters.

3.1 Optical Flow

When deciding to use a CNN to complete the task of speed estimation, it might seem sensible to construct an end-to-end system in which two consecutive frames, taken straight from the video, are used as input to the CNN. This could be as simple as stacking the frames to form one input or passing each frame to separate feature extraction networks after which the features are merged and further processed. The latter could be thought of as two non-trivial modules within a single CNN, which, combined with a sparse learning signal such as a single speed value, has been shown to be inefficient [38]. With this in mind, along with the fact that there is a way to encode prior information relevant to speed into the input to the CNN, optical flow is selected as a sensible choice for this task of speed estimation.

One drawback of using optical flow for this task is the potential for motion blur at higher speeds, due to the relative motion between the camera and scene being large during the exposure period. This motion blur can impact the assumption that pixel intensities are invariant between frames. In the setting of dash-cam footage, the blur can be rather linear for highway driving yet massively spatially variant if the car is turning. It is worth acknowledging that using an algorithm that is more robust to motion blur, such as [69], may well yield better results in the task of speed estimation. Additionally, although large 2D motions at high speeds can be difficult for optical flow algorithms to estimate, the iterative Farnebäck algorithm can cope to an extent. More troublesome is the requirement of structure in the scene for a good motion field estimate; the road surface is often uniform in appearance when lane markings are not present or sparse. However, there is usually enough structure in the scene away from the road that is consistent across samples, for example the presence of tall buildings in city centres and trees in residential areas. Highways present a particular problem due to lack of features and structure, as is further discussed in Chapter 4.

The Farnebäck algorithm for calculating optical flow is used for several reasons: dense optical flow is more appropriate for the task than tracking sparse features, the algorithm is fairly robust to large movements and provides good coverage of the frame - this is especially important so that the CNN can extract features. Using a more modern deep learning method for calculating optical flow could perhaps produce better results (Section 4.3.1). Another major motivation for using Farnebäck optical flow is that there is an out of the box implementation provided by OpenCV. This implementation allows the customisation of the parameters: pyramid levels, number of iterations and window size to name a few. The exact numbers used for these parameters are described in Section 3.2.3 but generally there is a trade-off between robustness and the obfuscation of the motion field. Although it would be ideal for the optical flow to be robust and clear, the problems of optical flow at high speeds required the parameters to value robustness

over clarity.

Clearly, the output of the Farnebäck optical flow algorithm is not an image but a set of corresponding magnitudes and directions that make up the motion field vectors for each pixel. It would be possible to simply use this data as input to the network but if we could encode more information in the input to the CNN - without increasing overfitting - it would certainly be valuable. By encoding the magnitude and direction data into the HSV colour space, along with the original saturation of the frame, more information is provided to the CNN and the optical flow can be visually checked. Importantly, by inputting the saturation of the original frame, some features can be detected by the CNN that would be undetectable in pure optical flow. These features are often sign-posts, trees or buildings which could be a significant factor for determining location or scale and thus inferring some information about the current speed of the vehicle.

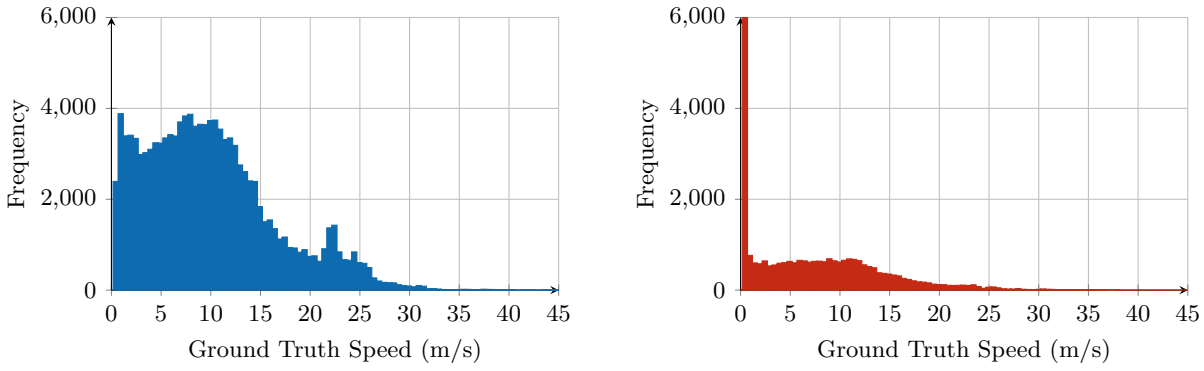
3.2 Dataset

3.2.1 Berkeley Deep Drive

Berkeley Deep Drive (BDD) [86] is UC Berkeley's contribution to large-scale annotated datasets for training autonomous vehicle models. As the number of conditions that such models might experience is vast, it is important that the data used to train these models is as diverse as possible. BDD provides geographic, environmental and weather diversity far beyond that of alternative datasets. The videos that comprise the dataset are collected in a crowd-sourced manner from tens of thousands of drivers, totalling more than 50 thousand rides. These videos were recorded at different times of day in diverse weather conditions in settings such as city streets, highways and residential areas. The distributions of these conditions can be viewed in Appendix A and some example frames extracted from BDD are shown in Figure 3.2.

The authors provide data that has been annotated for 10 tasks, including lane detection, semantic segmentation and 2D multi-object detection and tracking. The dataset consists of 100,000 videos, each 40 seconds long with annotation for the frames at every 10^{th} second. These videos are recorded at 30 fps in 720p with GPS/IMU data to preserve the driving trajectories - this is the most important part of the dataset for this project. Of the 100,000 videos, they have been split by the authors into training (70k), validation (10k) and testing (20k).

3.2.2 Custom Split of Berkeley Deep Drive



(a) Training set speeds.

(b) Testing set speeds.

Figure 3.1: Distribution of the speeds in the training and testing sets.

The 100,000 videos that comprise BDD total 1.8TB of data; which is quite an unwieldly amount for processing and training. The authors of BDD provide an online portal where all the data can be downloaded; conveniently, the videos are grouped into files containing 1,000 videos each, labelled as either training, validation or testing. Our custom split of BDD consists of 5,000 training videos and 1,000

testing videos for further processing. For each video there is a JSON file that contains the timestamped GPS data and the crucial speed recording measured **every second**. When attempting to parse these JSON files, it was found that a significant proportion were malformed and thus useless for this project. This left 3638 training videos and 678 testing videos remaining from which speed data could be extracted.

The CNN proposed in this chapter relies on optical flow encoded as an image and a speed recording as one sample to perform supervised learning. As will further be discussed, the optical flow is calculated from two consecutive frames that were extracted from the video according to the timestamp associated with each speed recording. Since each video is 40 seconds long and a speed recording was made every second; each video yielded 40 samples to be used for training or testing. In total, the training set consists of 120,000 optical flow image and speed pairs, with the testing set containing 26,945 such pairs. The distributions of speeds in the training and testing sets are plotted in Figure 3.1.

(a) Rainy, Residential, Day

(b) Rainy, City Street, Night

(c) Clear, City Street, Day

(d) Clear, Residential, Day

(e) Overcast, City Street, Dawn/Dusk

(f) Partly Cloudy, Highway, Dawn/Dusk

Figure 3.2: Example frames extracted from BDD videos, noting the weather, scene and time of day.

3.2.3 Pre-processing Pipeline

Before any neural network can be trained, the dataset must be prepared; this is usually a non-trivial process. The steps taken to pre-process and prepare the datasets for training and testing are outlined below. Care was taken to automate the process as much as possible and reduce the amount of intervention required, especially when errors are encountered. A python script utilising `pexpect` can run these separate processes overnight as the runtime can exceed 20 hours.

Downloading of Videos

The videos were downloaded from the portal [1] provided by the Berkeley Deep Drive authors. Videos are in the `.mov` format and are available for download in quantities of 1000 at a time: the first 5000 training videos and the first 1000 testing videos were downloaded. Each video title is a unique key of two 8 digit hexadecimal numbers that is used to identify the video for the remainder of processing. Additionally, the JSON data that contains the GPS/IMU readings for each video was downloaded.

Extraction of Frames and Speeds

Python is used for the script that extracts frames and speeds from the video due to its simple interface and libraries that make interacting with the operating system straightforward. To begin, the video keys and JSON keys are cached for efficient lookup. The video keys are then iterated over and the corresponding JSON file loaded. If the JSON file cannot be parsed or it does contain GPS data or a starting timestamp, then the video is discarded since no useful information could be extracted. Each speed recording has an associated absolute timestamp in milliseconds and therefore to calculate the relative timestamp of the speed recording, an absolute timestamp for the start of the video is required. If no such starting timestamp exists then the video is discarded. The relative timestamp of the recording could then be associated to a frame in the video. Conveniently, OpenCV's `VideoCapture` object allows for scrubbing through the video via milliseconds or frame number. Once a relative offset in milliseconds is established between the first frame and the speed recording, the video is scrubbed to the frame of the recording, saving it along with the preceding frame.

As the videos that comprise the BDD dataset are crowd-sourced, a significant proportion of them have been captured on smartphones. Some smartphones capture video in landscape but store them in a different orientation with the correct rotation encoded in the metadata of the video. OpenCVs `VideoCapture` does not read this metadata and correct the orientation accordingly, so `ffmpeg` is used to correct the orientation before saving the frames. A table is also saved containing the filename of each frame and the speed recording of each entry. It was decided that saving the images at this stage and calculating the optical flow later is more future-proof in case the method of calculating optical flow or any parameters were changed at a later date.

Calculation of Optical Flow

The pairs of frames are iterated over and read into another Python script that calculates the optical flow, encodes it into an image and saves it to file. The frames are first converted to grayscale before being passed into OpenCVs function for calculating Farnebäck optical flow. This function additionally takes some parameters that are outlined below:

- `levels = 1` - the number of pyramid layers including the initial image.
- `winsize = 15` - the averaging window size; larger values increase robustness but yield a more blurred motion field.
- `iterations = 2` - the number of iterations the algorithm does at each pyramid level.
- `poly_n = 5` - the size of the pixel neighbourhood used to find the polynomial expansion in each pixel; larger values increase robustness but yield a more blurred motion field.
- `poly_sigma = 1.3` - the standard deviation of the Gaussian that is used to smooth derivatives used as a basis for the polynomial expansion.

The default values are used where appropriate, unless some experimentation produced observationally better optical flow images. The output of this function is a list of vectors for representing the optical flow for each pixel. This list of vectors is converted to a list of magnitudes and a list of angles for each pixel and is then encoded into the HSV colour space as follows:

```
magnitude, angle = cv.cartToPolar(flow[..., 0], flow[..., 1])

hsv_flow[..., 0] = angle * (180 / np.pi / 2)

hsv_flow[..., 2] = cv.normalize(magnitude, None, 0, 255, cv.NORM_MINMAX)
```

Code Listing 3.1: Encoding optical flow data into HSV.

In other words: the **hue** corresponds to the angle of the optical flow (a value scaled to fit in the range [0,180]), the **saturation** is maintained from the source image and the **value** corresponds to the normalised magnitude of the optical flow vectors.

Preparation of the Dataset

Once the optical flow images are saved, it is time to decide which of those samples constitute the training set. By plotting the speed distribution of every training sample, it was observed that there were significantly more zero speeds than desired - the distribution looked similar to that of the testing set in Figure 3.1b. This is likely due to the amount of frames extracted when the cars were stationary at stop signs, traffic lights and slow moving traffic. A Python script is used to separate the speeds into two lists: one with all samples with speeds below 0.5 m/s and one with all other samples. These lists are randomly shuffled and a command line parameter for the total size of the training set determining how many of each list are added to the actual training set:

```
zero_speeds = list(filter(lambda x: x[1] <= 0.5, data_list))
non_zero_speeds = list(filter(lambda x: x[1] > 0.5, data_list))

random.shuffle(zero_speeds)
random.shuffle(non_zero_speeds)

zero_speeds = zero_speeds[:int(total_set_size)//50]
non_zero_speeds = non_zero_speeds[:(int(total_set_size) * 49)//50]

data_list = zero_speeds + non_zero_speeds

random.shuffle(data_list)
```

Code Listing 3.2: Selecting which samples constitute the training set.

These shuffled samples are then given an index so that they can easily be randomly sampled when loaded into a PyTorch dataset. For the training set `total_set_size` was set to 120,000.

Upload to BC4

A Python script then uploads all the training and testing samples to the Blue Crystal 4 (BC4) scratch storage space using `scp`.

3.3 Network Architecture

The network architecture began as a reimplementation of Nvidia's CNN architecture, proposed in their paper titled 'End to End Learning for Self-Driving Cars' [13] - the network diagram of which can be viewed in Appendix B. No code is provided by the authors so the architecture was constructed in PyTorch from scratch, along with an environment for loading the dataset and training the model. To begin, this network architecture was trained for 15 epochs on the custom split of BDD previously described. Stochastic Gradient Descent (SGD) with no momentum was used as the optimiser and Huber Loss as the loss function for a fairer comparison later on. The final results of this network, along with the results collected after the improvements, are discussed in Section 3.5. In short, the Nvidia architecture suffers from extremely poor generalisation capability and finishes with unacceptably high testing errors for the task of speed estimation. With this in mind, the improvements made to this architecture are now described below, before the final network architecture is presented.

3.3.1 Improvements

In order to test the effectiveness of the improvements, an ablation study was conducted. For each improvement, a model is trained without it - in the case of the optimiser, SGD is used instead of Adam [53] and ELU [20] in place of ReLU [65] for the activation function. The results of this study are presented in Table 3.1, showing that the final network outperforms all others. In other words, each improvement contributes positively to the final network architecture.

Method	Huber Loss	MSE	Mean L1 Error
Ours	**1.163**	**5.770**	**1.571** ms^{-1}
Ours - No Data Augmentation	1.385	7.760	1.771 ms^{-1}
Ours - SGD	1.644	9.179	2.073 ms^{-1}
Ours - No Batch Normalisation	1.355	7.068	1.792 ms^{-1}
Ours - No Dropout	1.183	5.792	1.593 ms^{-1}
Ours - No Max Pool	1.281	6.719	1.681 ms^{-1}
Ours - ELU	1.352	7.002	1.786 ms^{-1}
Ours - No Output Limiting	1.318	6.431	1.757 ms^{-1}

Table 3.1: The results of an ablation study performed on the final network architecture.

Data Augmentation

The original system proposed by Nvidia adds to the training set by augmenting the data with artificial shifts and random rotations; the magnitude of these perturbations is chosen randomly from a normal distribution with a mean of 0 and a standard deviation of twice that measured from the data supplied by human drivers. However, this standard deviation value is not published by the authors and PyTorch does not currently support transformations with a normal distribution, so the Nvidia architecure was initially trained without any data augmentation. The augmentations by Nvidia were applied offline and added to the training set before the model was trained. From observing the overfitting of the Nvidia model on the custom split of BDD, it was clear that a more aggressive augmentation scheme was required.

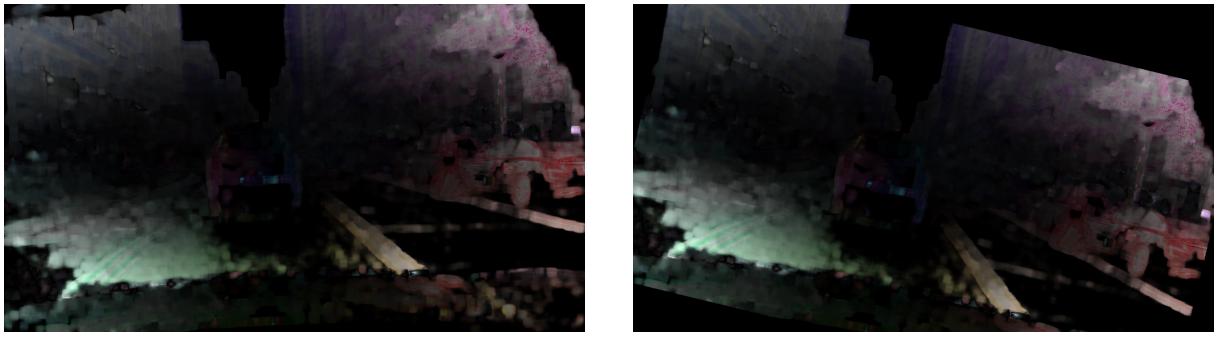


Figure 3.3: An example data augmentation transform applied during training.

During the training of our model, the samples are augmented online - that is, a random transformation is applied for every sample during training. This allows for different transformations to be applied to the same sample in different epochs of training, increasing the diversity of the training set. No transformations were applied during testing. A uniform random rotation between -15 and 15 degrees about the centre is applied along with a random uniform translation in the y direction of between -20% and 20% of the image height (between -72 and 72 pixels). It is possible that larger rotations and translations could obstruct features and reduce the accuracy of the optical flow image, nevertheless the current augmentations reduced overfitting. Other ranges for the rotation and translation were experimented with, 15 degrees rotation and 20% translation proving to be the most effective. Additionally, applying perspective transforms during training was tried but yielded poor results - this is likely due to the relaxation of parallel lines. The result of applying online data augmentation during training is a 16% and 26% reduction in Huber Loss and Mean Squared Error respectively (Table 3.1).

Optimiser

Stochastic gradient-based optimisation is not new, yet it has proven incredibly effective for training the model parameters of deep networks. The stochastic nature of SGD is important; by estimating the actual gradient of the cost space - calculated on the entire training dataset - with the gradient of a randomly selected subset of the training dataset, SGD provides a much more computationally efficient solution at the expense of slightly slower convergence times.

The ultimate aim of an optimiser is to find the global minima of the cost space, avoiding saddle points and local minima along the way. Usually, deep learning cost spaces are highly non-linear and high-dimensional such that they are littered with these obstacles. One popular addition to vanilla SGD is momentum [68] which is as simple as it sounds; by applying some momentum to the update at each step, the optimiser has a better chance of effectively rolling along saddle points and up the hills of local minima to find a new downwards gradient. A more recent and extremely successful advance involves the computation of individual adaptive learning rates for different parameters from the first and second moments of the gradients. This method is named Adam [53] and has proven effective for large datasets and high-dimensional parameter spaces and is able to deal with sparse gradients. Therefore, Adam is well suited to be used as the optimiser for speed estimation with a dataset of reasonable size and a large model. Indeed, when Adam is employed, a 29% improvement in Huber Loss and a 37% improvement in Mean Squared Error over vanilla SGD are achieved (Table 3.1).

Batch Normalisation

Batch Normalisation (BN) is described in detail in Section 2.4.2 but, as a reminder, BN aims to reduce internal covariate shift by normalising layer inputs [47]. This should allow for higher learning rates, better invariance to initialisation and potentially a regularisation effect. Through experimentation, it was found that BN applied to the convolutional layers - after the convolution and before the activation - achieved the best results. BN is not applied to the fully connected layers after dropout due to the results from Li et al. [58]. It was found that dropout shifts the variance of specific units when the network is transferred from train to test, but BN maintains its statistical variance that is learned throughout training. The authors found that this inconsistency can lead to unstable numerical behaviour and subsequent poor predictions.

Table 3.1 demonstrates the effect of BN on the final testing errors, giving an improvement of 15% and 18% in Huber Loss and Mean Squared Error respectively. This is likely due to the faster convergence and regularisation effects of BN; the task of speed estimation on such a diverse dataset is extremely prone to overfitting.

Dropout

The regularisation technique dropout is described in detail in Section 2.4.1. The inclusion of dropout in the final network architecture is motivated by the lack of generalisation ability present in the Nvidia architecture. Dropout is applied to the input to the first fully connected layer after it has been flattened following the final convolutional layer. The original authors of dropout propose applying it to every fully connected layer although this did not improve the final testing results of our network. Instead, one application of dropout was enough to reduce the Huber Loss by a modest 2%.

Regularisation of deep networks is a fine balance: reducing overfitting at the expense of a higher training loss is usually good, although in general the training loss is a lower bound on the testing loss. Too much regularisation can reduce the representational capacity of the network and ultimately push the training and testing losses higher than optimal as a result. It is possible that batch normalisation and data augmentation have reduced the effect of dropout and this is supported in the final results (Table 3.1), although it is noted that the inclusion of dropout does not worsen the errors.

Max Pooling

The structure of the convolutional layers in the original Nvidia architecture is clearly split into two parts: three strided convolutional layers with 5×5 kernels and two non-strided convolutional layers with smaller 3×3 kernels. It is therefore an obvious choice to apply a strided max-pool between these two convolutional parts. As mentioned in Section 2.2.2, max pooling can add some spatial invariance to the presence of

features and this is especially important when applied to the task of speed estimation. It is likely that, through the different camera positions on cars and data augmentation, important features spatially vary between samples. Additionally, cars can experience significant vibration and poor road conditions that can result in camera shake and may offset features between frames.

Through testing it was determined that one max pool between the third and fourth convolutional layer resulted in the best performance and was thus adopted into the final network architecture. The results of the network with and without max pooling is shown in Table 3.1; max pooling providing a 9% and 14% improvement in Huber Loss and Mean Squared Error respectively.

Activation Function

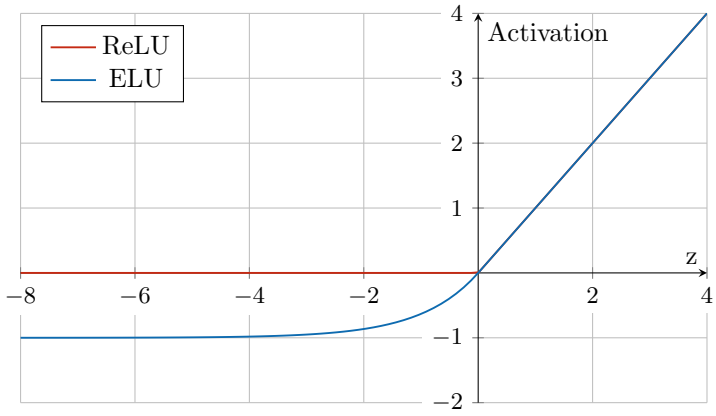


Figure 3.4: The Recitifed Linear Unit (ReLU) and Exponential Linear Unit (ELU, $\alpha = 1.0$).

The original Nvidia architecture employs the Exponential Linear Unit (ELU) as the activation function for each layer. ELUs were first proposed by Clevert et al. in 2016 [20] as an improvement to the popular activation function Rectified Linear Unit (ReLU) [65]. The equations for ELU and ReLU are very similar: ELUs saturating to a negative term for inputs below zero. This is visualised in Figure 3.4 and formally as:

$$\mathrm{ELU}(x) = \max(0, x) + \min(0, \alpha \cdot (\exp(x) - 1)) \tag{3.1}$$

$$\mathrm{ReLU}(x) = \max(0, x) \tag{3.2}$$

where the negative limit of ELU is controlled by the value of α. The theory behind ELUs is that the negative values push the mean of the activations closer to zero which acts in the same way as batch normalisation, as described in Section 2.4.2, but with lower computational complexity. This would allow ELU without batch normalisation to be competitive with ReLU with batch normalisation. However, from testing the final architecture with ELU and ReLU activations on all layers, it can be seen in Table 3.1 that the final loss is 14% percent lower when ReLUs are employed. This is possibly due to the fact that the output of the network, a speed prediction, can only be a positive real number and perhaps the presence of negative activations could hinder the prediction. More experimentation on this topic is required.

Limiting Output Function

During initial optimisation of the network, it was observed that outlier samples in the testing set were contributing large errors to the final statistics. Even though Huber Loss is less sensitive to such outliers than Mean Squared Error, it was still a problem that needed addressing. The solution is to pass the output of the final neuron through a tanh-like function that positively saturates towards some predetermined speed limit. From observation of the training and testing data, a speed limit of 45 m/s ($\approx$ 100 mph) is chosen. Therefore the network cannot predict speeds above this value; in theory reducing the severity of outliers. Fully autonomous vehicles will not be able to willingly pass above the speed limit of the road, so this remains a practical measure if this system were to be deployed.

Negative velocity indicates reversing, however negative speed is a null concept and as such, the function is also clamped to zero for negative outputs of the network. The output of the final neuron is divided

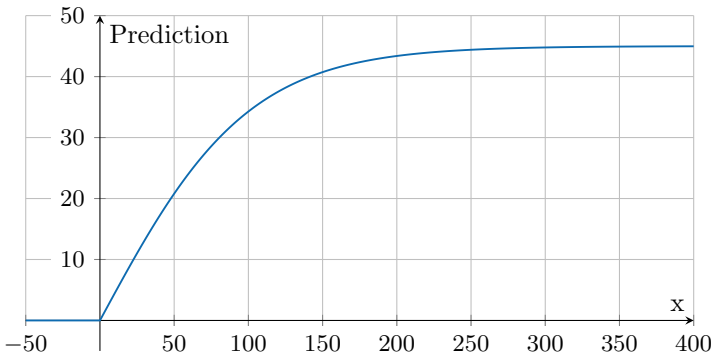


Figure 3.5: The positively saturating tanh function applied to the output of the final neuron as the final step of prediction.

by 100 before input into tanh for numerical stability at the beginning of training and to significantly reduce the gradient of the tanh curve. This affords the prediction to be less sensitive to small errors in the output of the final neuron. This function is graphed in Figure 3.5 and written as follows:

$$f(x) = \max(0, \tanh\left(\frac{x}{100}\right) \cdot 45) \tag{3.3}$$

The effect of applying this function is a 12% and 10% decreases in Huber Loss and Mean Squared Error respectively (Table 3.1).

3.3.2 Final Network Architecture

Following the improvements that have been outlined in this Chapter, the final network architecture can now be presented. A diagram of the architecture can be seen in Figure 3.6 and a more detailed layer-by-layer description is contained in Table 3.2. For some example feature maps produced by the convolutional layers, please see Appendix C.

Name	**Type**	**Output Size**	**Kernel Size**	**Notes**
Input		$3\times360\times640$		
Conv1	Convolution	$24\times180\times320$	5×5	2×2 stride, Batch Norm, ReLU
Conv2	Convolution	$36\times90\times160$	5×5	2×2 stride, Batch Norm, ReLU
Conv3	Convolution	$48\times45\times80$	5×5	2×2 stride, Batch Norm, ReLU
Pool1	Max Pooling	$48\times23\times41$	2×2	2×2 stride
Conv4	Convolution	$64\times23\times41$	3×3	Batch Norm, ReLU
Conv5	Convolution	$64\times23\times41$	3×3	Batch Norm, ReLU
		Flatten, Dropout ($p = 0.5$)		
FC1	Fully connected	1164		ReLU
FC2	Fully connected	100		ReLU
FC3	Fully connected	50		ReLU
FC4	Fully connected	10		ReLU
Output	Fully connected	1		$\max(0, \tanh\left(\frac{x}{100}\right) \cdot 45)$

Table 3.2: Network Architecture Details (Output Size = Channels × Height × Width).

The network consists of 9 layers in total: 5 convolutional layers followed by 4 fully connected layers. The inputs to the network are RGB optical flow images, resized to 640×360 pixels and scaled such that the values in each channel are transformed from $[0, 255]$ to $[0.0, 1.0]$. This forms a 3D input image of dimensions 3×360×640 (PyTorch accepts Tensors of the form Channels × Height × Width). As previously mentioned, the images are augmented during training - after resizing and before scaling. The optical flow images first pass through a series of three convolutional layers with strides of 2×2 and increasing

numbers of kernels (24, 36, and 48). These layers have kernels of size 5×5, applying batch normalisation after convolution and ReLU as the activation function. This is followed by a max pooling layer with a stride and kernel of size 2×2. The input is then passed through two convolutional layers consisting of 64 3×3 kernels each, again applying batch normalisation before the ReLU activation. These convolutional layers are designed to perform feature extraction with some spatial invariance induced by the max pool.

The output of the convolutional layers is then flattened and these units are retained with a probability of 0.5 during training. Four fully connected layers follow, each with decreasing numbers of units but all using ReLU as the activation function. The network outputs one number that is then fed through the tanh limiting function - described in Section 3.3.1 - to give a final speed prediction. The loss for each batch is computed using Huber Loss (Section 2.3) and the error is backpropagated through the network in order to update the parameters.

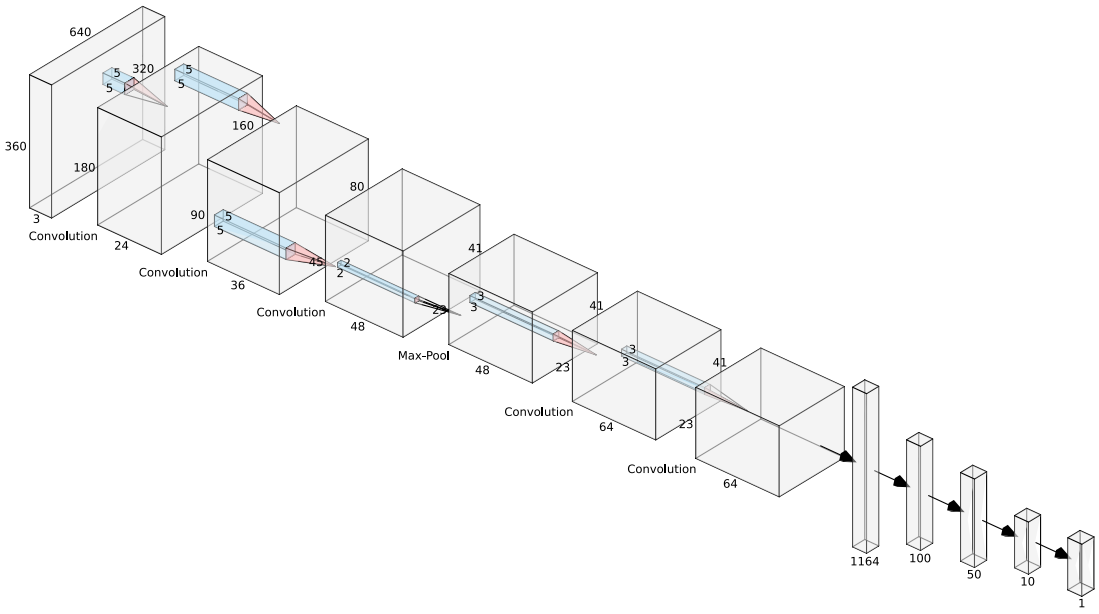


Figure 3.6: Network Architecture.

3.4 Training

The network is built and trained using the PyTorch Python API, as previously mentioned. This is effectively a wrapper for more performant C/C++ code and, more importantly, CUDA code that can be executed on the GPU in a massively parallel manner. Alternatives to PyTorch include Google's offering, Tensorflow, as well as Keras and MXNet to name a few. PyTorch is chosen due to the simple and elegant interface provided. In order to accelerate training, a GPU is essential and for computer vision tasks that have large models and images as input to the network in significant batch sizes, a GPU with a large amount of VRAM is necessary. This is due to the fact that the model itself and a single batch for training must fit onto the VRAM at one time. Fortunately, the university possesses a supercomputer in the form of BC4: equipped with 64 Nvidia Pascal P100 GPUs, each with 16GB of VRAM. These cards are much more powerful and expensive than even high-end desktop GPUs and thus provide the perfect infrastructure on which to train deep learning models. Although it is common for demand to greatly outstrip the supply of time on these GPUs, the delay is not enough to justify a different route.
When training the model - that is to say, optimisation of the model parameters via gradient descent - there are several decisions that must be made. Firstly, which optimisation algorithm should be used; the comparison of two such optimisers, SGD and Adam, has been previously discussed in Section 3.3.1. There are other alternatives of course, Adagrad [28] and RMSProp [44] for example, although a brief examination of the literature indicates that Adam is the widely preferred optimiser [72]. Secondly, what values should the hyperparameters take, specifically the learning rate, batch size and number of epochs. The batch size is simply the number of samples in each batch, with the gradients used to update the

Batch Size	Huber Loss	MSE	Mean L1 Error
32	1.271	6.585	1.700 ms^{-1}
64	**1.163**	**5.770**	**1.571** ms^{-1}
128	1.233	5.974	1.656 ms^{-1}

Table 3.3: Final testing errors of runs with different batch sizes (15 epochs, learning rate = 0.001).

model averaged over the samples in the batch. A larger batch size should provide a better estimate of the gradient with respect to the total training set but provide fewer updates to the models parameters. However, a larger batch size also works to reduce the runtime of training; the opposite being true for smaller batch sizes. The results of training with batch sizes of 32, 64 and 128 are shown in Table 3.3, a batch size of 64 providing the best results.

Learning Rate	Huber Loss	MSE	Mean L1 Error
0.001	**1.163**	**5.770**	**1.571** ms^{-1}
0.0005	1.214	5.969	1.633 ms^{-1}
0.0001	1.329	6.887	1.733 ms^{-1}

Table 3.4: Final testing errors of runs with different learning rates (15 epochs, batch size = 64).

The process of choosing the best learning rate is the same as that of the batch size: several learning rates were tested one by one, expanding out from 0.001 positively and negatively. Learning rates higher than 0.001 caused training instability and those lower than 0.001 provided poorer performance as shown in Table 3.4. The number of epochs is set to 15 as training for longer produced no benefit. No early-stopping is implemented although the potential benefit of this is discussed in the next chapter. Nonetheless, the final set of hyperparameters are summarised in Table 3.5.

Parameter	Value
Batch Size	64
Learning Rate	0.001
Epochs	15

Table 3.5: Hyperparameters used for training the final model.

The original Nvidia architecture is trained by calculating the Mean Squared Error (MSE) between the prediction of the network and the ground truth. Through experimentation with using MSE as the loss function for our improved network, it was noticed that the error was heavily influenced by outliers. The effect that these outliers have on the loss function can skew the learning of the network: the parameters are updated to fit these outliers at the expense of fitting the rest of the training data. This generally leads to poorer overall performance on both the training and testing sets. To improve performance, the outliers can be identified and removed from the training set, or a more robust loss function can be employed. If the network is to function in diverse conditions, removing any data will not help this cause; it would be ideal if the network still trains on all the available data, but the amount the network learns from these outliers is reduced. Therefore, Huber Loss is employed as the loss function (Section 2.3).

3.5 Results

Now that our network architecture and training procedure have been presented, the final comparison of results can be made between the Nvidia architecture and our proposed improvements. The full list of relevant errors can be seen in Table 3.6 and demonstrates a significant improvement in all metrics. The large difference between the Mean Squared Error and Mean L_1 Error results for our improvements points to the presence of severe outliers in the testing set as these mean averages tend to be sensitive to such

outliers. This is further observed in the gulf between the mean and median results for both Square Error and L_1 Error.

Method	Huber Loss	MSE	RMSE	Mean L1 Error	Median L1 Error
Nvidia	2.525	17.418	4.173 ms^{-1}	2.979 ms^{-1}	2.081 ms^{-1}
Ours	**1.163**	**5.770**	**2.402** ms^{-1}	**1.571** ms^{-1}	**0.892** ms^{-1}

Table 3.6: Comparison of the final testing errors between the Nvidia architecture and our proposed improvements.

The results are further analysed in the following Chapter (Section 4.1) in which they are broken down by ground truth speed and the various conditions in which the testing samples were recorded. The loss curves for both the original Nvidia architecture and our proposed improvements are displayed in Figure 3.7 and Figure 3.8 respectively. The key thing to note from the loss curves is the disparity between the training and testing losses for the Nvidia architecture. This is a classic example of overfitting or lack of generalisation: the network learns the detail and the noise in the training data to such an extent that any unseen data presents a significant challenge. Overffitting is usually due to a model that has too many parameters and can be solved by simplifying the model in the first instance. However, our proposed improvements do not reduce the capacity of the model yet produce testing errors that are almost exactly in line with the training errors. The training loss is not relevant to a model once it is trained, the useful feature of deep neural networks is their ability to generalise to unseen data.

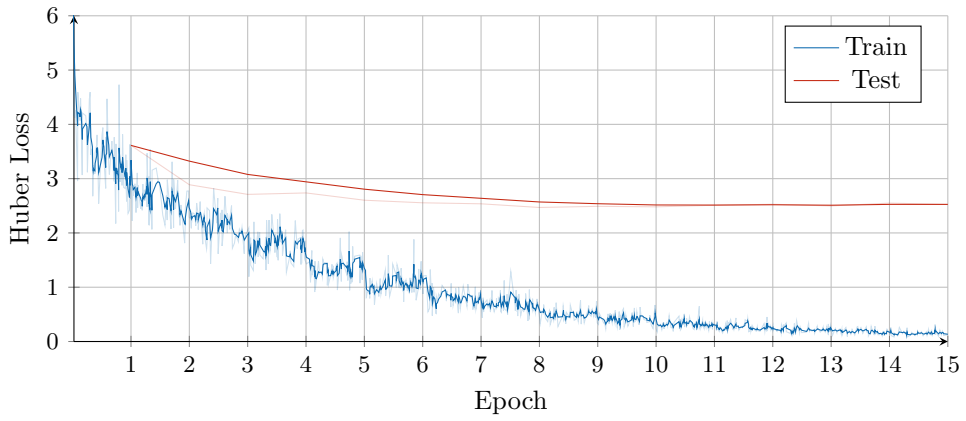


Figure 3.7: Loss curves of the original Nvidia architecture.

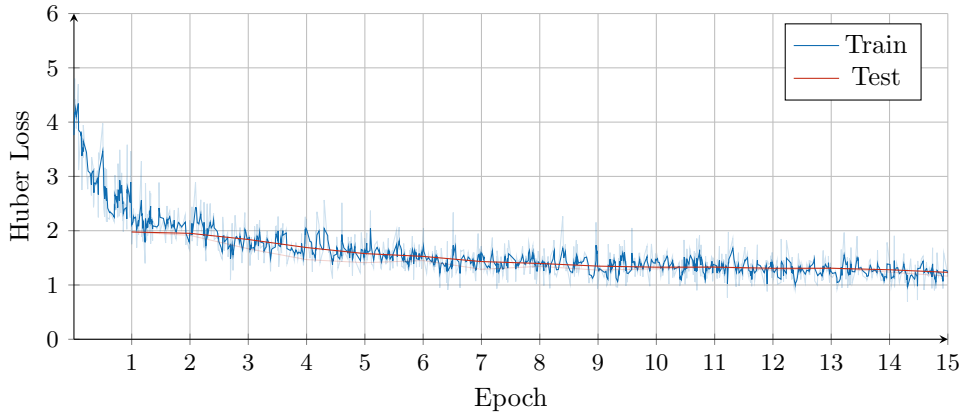


Figure 3.8: Loss curves of the proposed improvements.

Chapter 4

Critical Evaluation

The final benchmark results presented in the previous chapter are promising, but require contextualising to accurately evaluate the performance of the network. This context is provided by firstly examining the distributions of the ground truths in the testing set against the predictions of the network. In addition, the performance of the network at different speeds is analysed, along with the performance in the diverse conditions afforded by BDD. This analysis will show that the network and the training procedure are, of course, imperfect. The limitations of this project are then discussed, including the problems with optical flow in the context of autonomous cars and practical considerations such as runtime. To finish, ways in which this project could be extended and potentially improved are suggested.

4.1 Performance Analysis

Evaluating the performance of the network by analysing a single number can only go so far: metrics such as MSE and Mean L_1 Error, computed on the entire testing set, tell us very little about what the network is actually doing and if the results produced are of any significance. For example, it might be the case that the network is just learning to model the distribution of the training set, predicting the mean of the training set for every sample during testing. Therefore, the distributions of the speeds in the testing set and the predictions of the network are plotted in Figure 4.1. This is essentially a sanity check that the network is making sensible predictions overall and not exhibiting undesirable behaviour. The distribution of the predictions certainly passes this test; the shape is very similar to that of the testing set. One noticeable difference is that the network predicts around half the number of ground truth speeds in the first bin ($[0, 0.5)$ m/s), although this appears to be compensated by an increased number of predictions in the second bin ($[0.5, 1)$ m/s).

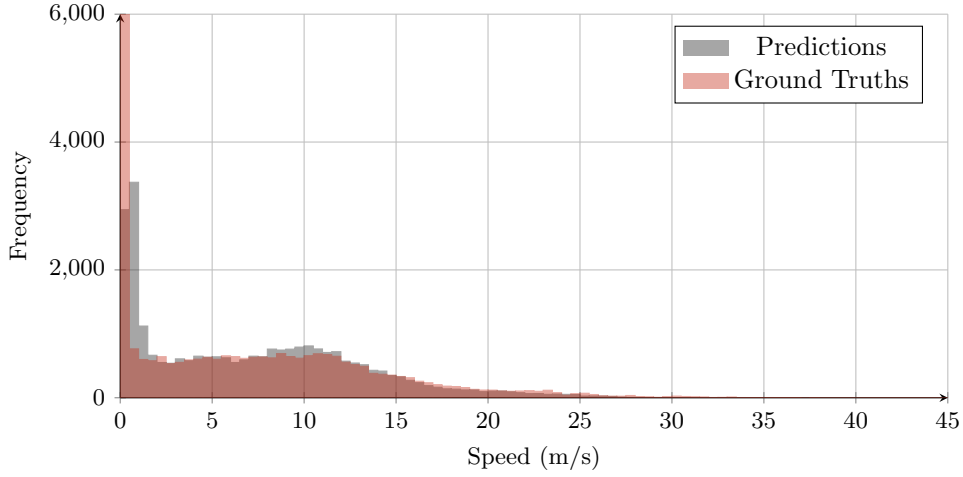


Figure 4.1: Distribution of the speeds in the testing set and the predictions made by the network.

4.1.1 Performance by Speed

A useful and indicative analysis of the network can be made by observing how the error varies with the ground truth speed. This can pinpoint certain weaknesses of the network or the training process, allowing for more meaningful alterations than simply observing the loss curve. Indeed, it was this analysis that motivated the change from MSE to Huber Loss as the loss function, due to significant outliers at high speeds. These outliers were confirmed by the disparity between the mean and median results for both L_1 Error and Squared Error; a mean average is much more sensitive to such outliers.

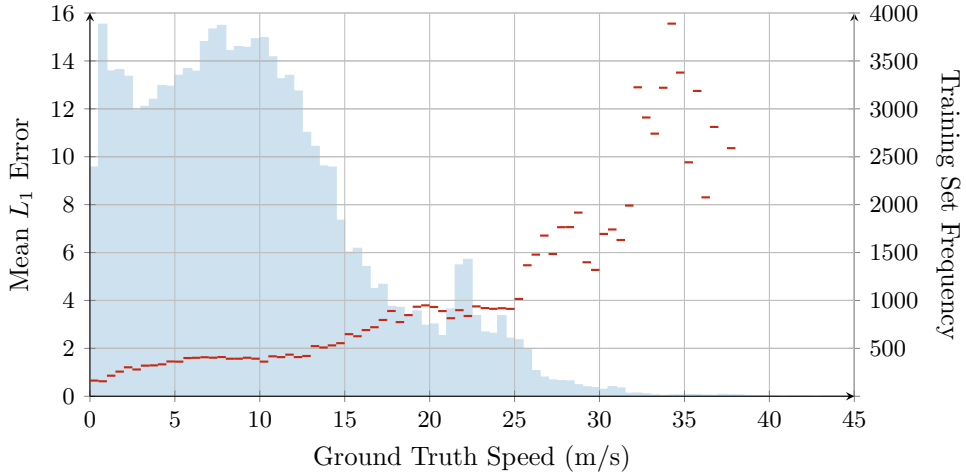


Figure 4.2: Mean of the L_1 Error on the testing set, binned by ground truth speed (red) and a histogram of the speeds in the training set (blue).

The Mean L_1 Error at different speeds is plotted in Figure 4.2 as the red lines. These lines are superimposed on, and aligned with, the distribution of speeds in the training set (semi-transparent blue). The major trend to observe is the almost exponential growth in error as the ground truth speed increases. It is not clear why this is the case; it could be owing to the inaccuracy of optical flow at high speeds due to large motion or simply the lack of training data for high speeds. Indeed, the distribution of training data shows that the network performs well for speeds that have good coverage in the training data. It is hard for any neural network to generalise to data dissimilar to that which it has been trained on - when generation of new data is not the purpose of the network. Nonetheless, the network makes consistent predictions for speeds lower than 13 m/s, errors beginning to grow exponentially for speeds higher than this. This is a promising sign, errors are consistent for which there is sufficient training data coverage, as indicated by the frequency of these speeds in the training set.

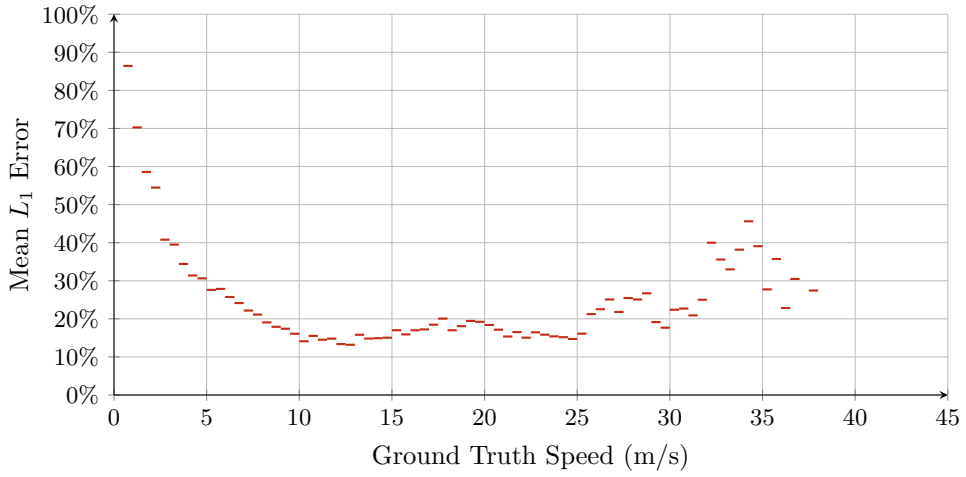


Figure 4.3: Mean L_1 Error as a percentage of the ground truth speed.

A different way of looking at the performance of the network at different speeds is to rescale the Mean L_1

Error to a percentage of the ground truth speed and this is plotted in Figure 4.3. This metric indicates that the network performs best for speeds between 10 and 25 m/s, which is not immediately obvious from looking at Figure 4.2. The minimum error as a fraction is 13% between 12.5 and 13 m/s.

4.1.2 Performance by Condition

How the network performs at varying speeds is important, but so too is the performance in diverse conditions. It would be a failure for the network to be completely useless in any single weather condition or scene: to consider the network a success, it must perform well across all conditions. Fortunately, the weather condition, time of day and scene type are provided as part of BDD for the majority of videos; the distributions of these conditions for the training and testing sets can be viewed in Appendix A.

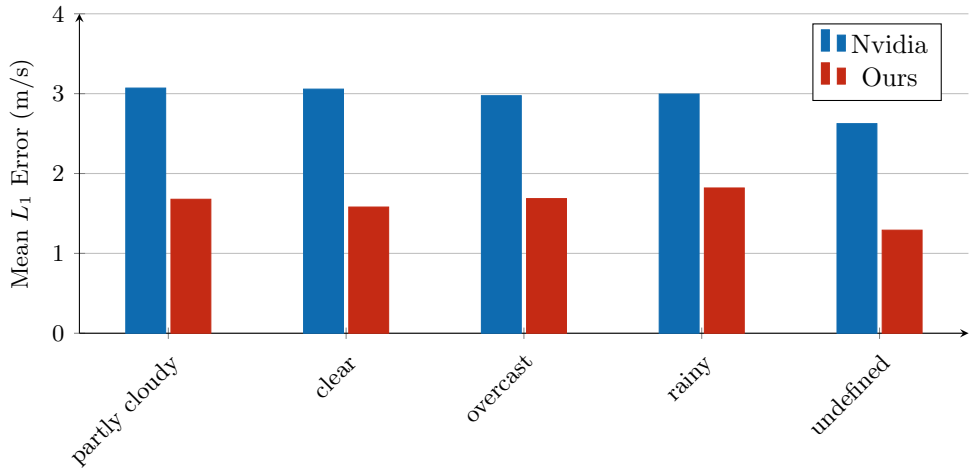


Figure 4.4: The performance of Ours and Nvidia's network on the testing set by weather condition.

To begin, in Figure 4.4, the Mean L_1 Error are plotted for the original Nvidia architecture and our improvements, grouped by weather condition. Both networks perform fairly consistently in all of these different weather conditions, despite these conditions not being equally represented in the training set (Figure A.2a). Our improvements outperform the Nvidia architecture in every weather. The most surprising of these results is the fact that our network performs well in rainy conditions - even if slightly worse than other weather. The rain has an effect on the optical flow estimation: droplets of rain hitting all surfaces will obscure features, the windscreen wipers will provide erroneous optical flow measurements and wet surfaces will reflect light, further confusing the optical flow. The effect of reflections on the road surface is particularly pronounced in Figure 4.5 to the left of the white car.

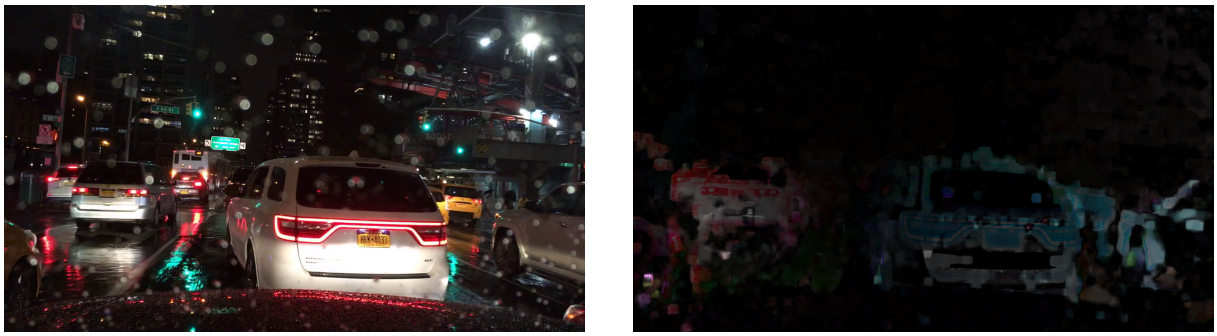


Figure 4.5: An optical flow image in rainy conditions.

In addition to the weather condition, each video is labelled with the scene in which it was recorded. The errors for each of these scenes are plotted in Figure 4.6; our improvements outperform the Nvidia architecture in every scene. The errors for samples recorded on highways is significantly higher than the rest of the scenes (excluding 'other'), in concurrence the with the performance by speed analysis that the network produces higher errors at higher speeds. The distributions of these scenes in the training and

testing sets can be viewed in Figure A.1, showing that the cause of high errors for the 'other' scenes is likely due to a lack of training data. Likewise with the weather performance, it is a good sign that the network produces very similar errors for residential and city street scenes even though these scenes are not represented equally in the training set. Finally, the performance for each time of day is plotted, again showing consistent error across all times of day (Figure 4.7), but our network again outperforming the Nvidia network in every category.

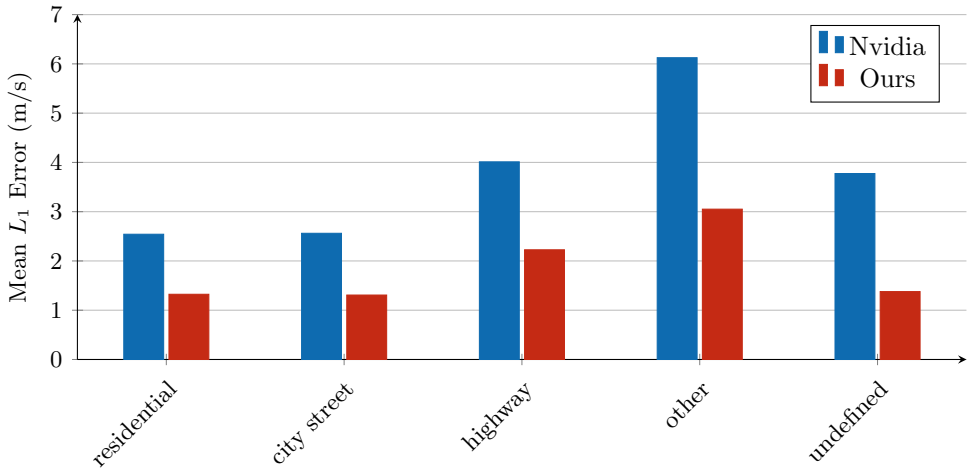


Figure 4.6: The performance of Ours and Nvidia's network on the testing set by scene (other = {gas station, parking lot, tunnel}).

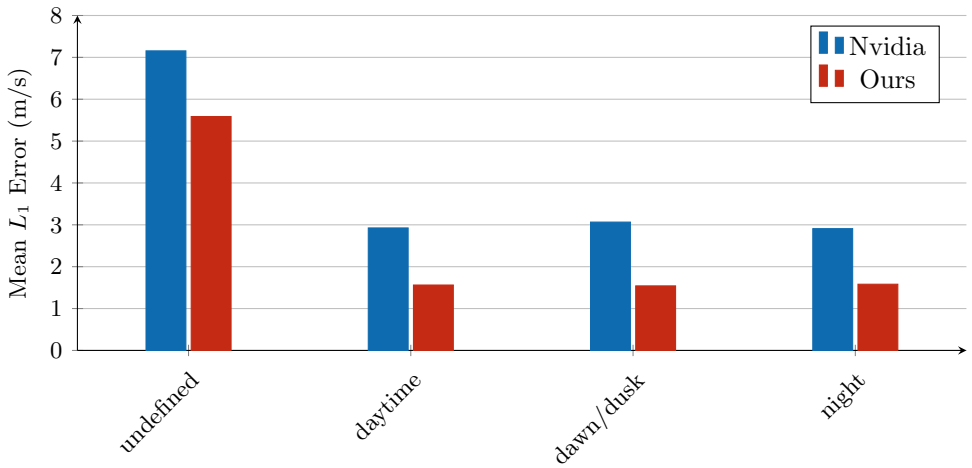
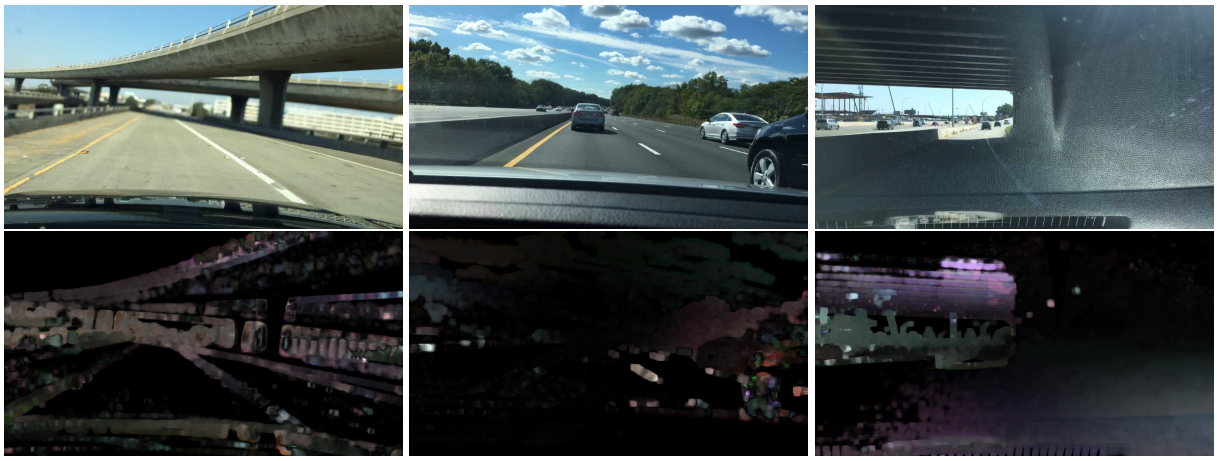


Figure 4.7: The performance of Ours and Nvidia's network on the testing set by time of day.

4.1.3 Failure Cases

An analysis of the failure cases is helpful in identifying weaknesses of the network, fundamental issues with employing optical flow in the context of dash-cam footage or simply complications of poor data collection and filtering. Three such failure cases are visualised in Figure 4.8; the top row contains the frame at the speed recording and the bottom row contains the corresponding optical flow images - computed from the frame in the top row and the preceding frame in the video.

In the first failure case (Figure 4.8a), a horizontal band in the middle of the frame is out of focus, this is most obvious by looking at the white building in the right of the frame. In addition, the previous frame has a poor focus as well. When the optical flow is calculated on these two out-of-focus frames, the estimate is very poor and blurry. Since the underlying saturation values are out of focus, the network will struggle to extract any useful features and it is unlikely to find similarities in the training data. It



(a) Pred: 13.3 m/s, Label: 30.5 m/s. (b) Pred: 14.8 m/s, Label: 34.1 m/s. (c) Pred: 2.7 m/s, Label: 18.1 m/s.

Figure 4.8: A set of failure cases. Top row: Frames at speed recording. Bottom row: Corresponding optical flow images used as input to our CNN.

is worth noting that most autonomous cars have several cameras with varying focal lengths embedded into the front bumper. All of these frames could be incorporated into the network to reduce the effect of poor focusing.

The second failure case (Figure 4.8b), another example of highway footage, contains two problems. Firstly, the optical flow image has poor coverage of the frame; the road surface and median barrier are both very uniform. It is a common fault of optical flow that it is effectively impossible to estimate the motion field within a uniform area. This lack of features in the optical flow estimation ultimately leads to a lack of features for the CNN to extract and process. Secondly, the rotation of the left wheel of the black car in the adjacent lane is out of sync with the frame rate of the camera, causing the wagon-wheel effect. This acts to confuse the optical flow estimation and results in an erroneous estimate around this wheel. Considering the lack of coverage of the frame already, this error further obfuscates the optical flow image.

The third failure case (Figure 4.8c) simply demonstrates the inherent problem with dash-cam footage: it is possible for the image to be easily obscured. This takes the form of droplets and windscreen wipers in the rain, the actual dashboard and bonnet and reflections from the dashboard onto the windscreen - as is the case with this example. The reflection blocks out half of the frame from an optical flow estimate; an effect that no doubt presents a problem for our network.

4.2 Limitations

4.2.1 Optical Flow

Reversing Videos

The labelling of the BDD videos does not indicate whether the vehicle is reversing or travelling forwards; there is no numerical way of determining the direction. Clearly the optical flow image in these two cases will look completely different, since the hue in the HSV colour space corresponds to angle of the optical flow. By removing the reversing video - or indicating the direction of travel to the CNN - the network would not have to learn whether the car is reversing or not, thus eliminating a point of failure and improving the capacity of the network.

Relative Motion of Vehicles

To be able to predict the absolute speed of the ego-vehicle, it would be ideal for the optical flow to only be calculated using stationary objects in the scene. This is, however, almost always not the case in the real world: other vehicles travel at different speeds on the road and pedestrians walk alongside and cross the street in city centres. The relative motion between the car and these moving objects is always different to the absolute motion of the car. This effect is especially pronounced when the car is stationary

(Figure 4.9) but equally important when on the highway. The solution to this is a matter of future work, although one solution involves another preprocessing step; semantically segmenting the scene to remove other road vehicles and pedestrians from the optical flow image.

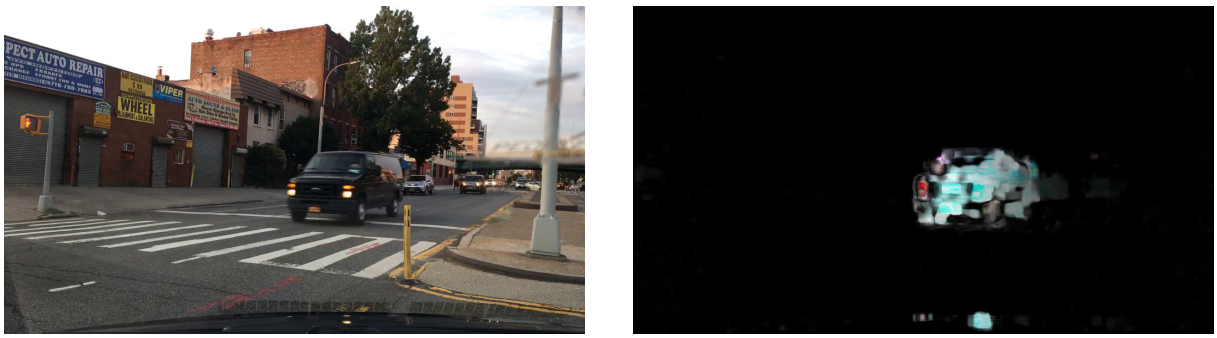


Figure 4.9: An example frame and corresponding optical flow image showing only the relative motion of another road vehicle.

The Cityscapes dataset [21] offers labelled data for semantic segmentation and publishes leaderboards for pixel-level and instance-level semantic labelling. Pixel-level semantic segmentation simply aims to assign every pixel in the image a class (e.g. vehicle, tree, road etc.) whereas instance-level semantic segmentation goes further by segmenting objects in the scene and then assigning a semantic mask to each object. The simplest solution would be to identify all pixels that represent other road vehicles and either remove the corresponding pixels in the optical flow image, potentially interpolating over them. This, however, fails to account for road vehicles that are stationary and therefore providing a useful optical flow estimate. The identification of separate vehicles using instance-level segmentation allows for stationary vehicles - whose optical flow is consistent with surrounding objects - to be included in the optical flow estimate.

Accuracy at High Speeds

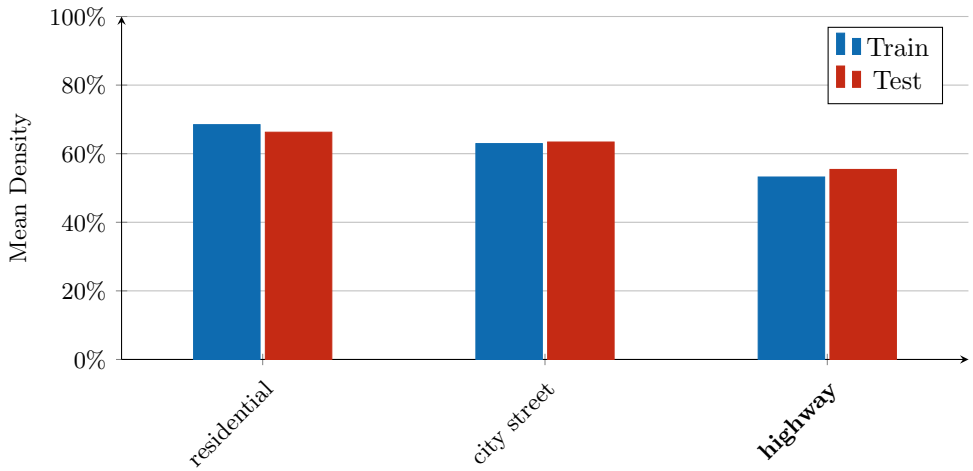
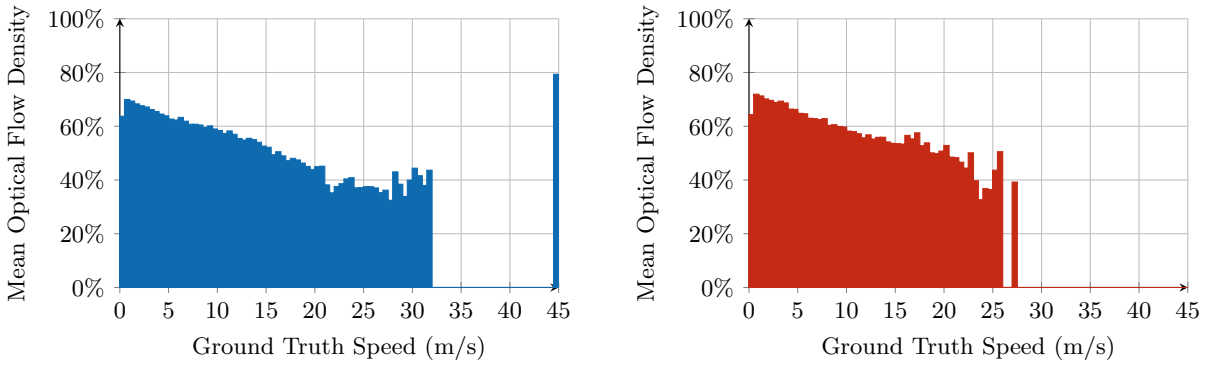


Figure 4.10: Distribution of the optical flow density of each image in the training and testing sets by scene.

Having previously established that the network performs poorly at high speeds and therefore also on highway scenes, we can now analyse why this might be the case. The mean optical flow density for each type of scene is plotted in Figure 4.10 - density being the percentage of non-zero pixels in the optical flow image. These graphs show that on average, a highway scene has a lower optical flow density than residential and city street scenes. This is due to the lack of structure in highway scenes and the problems with optical flow estimation at high speeds. A lower optical flow density means that there is less information in the image and fewer features that can be possibly extracted by the CNN. This relationship, however,

is not so straightforward; stationary optical flow images tend to be very low density - apart from moving objects in the scene (Figure 4.9).



(a) Training set optical flow density.

(b) Testing set optical flow density.

Figure 4.11: Distribution of the optical flow density of each image in the training and testing sets binned by speed (bin width = 0.5 m/s, min # of samples = 30).

The relationship between higher speeds and lower optical flow density is supported by the graphs in Figure 4.11 - results are omitted (set to 0) for bins with fewer than 30 samples. These results are in line with those of Figure 4.10: highway scenes have lower optical flow density on average. Indeed, the optical flow density also directly correlates with the error of the prediction (Figure 4.12). The performance of the network at higher speeds thus depends on the optical flow density of the samples. To improve the accuracy at higher speeds, it would be necessary to train the model on much more data at these higher speeds.

4.2.2 Runtime Performance

For a component of a autonomous vehicle system to be practically feasible, the runtime of the system must be fast enough to operate at the required frequency. In an ideal world, the system would be able to complete a forward pass at the same frequency that the frames of the video are captured - up to 60 times a second (30 in the case of BDD). In practice, however, the amount of computational power required to achieve this frequency is limited by power consumption; the more power the systems that drive the car consume, the shorter the driving range and the higher the frequency of charging. In addition, there are many other systems running concurrently on autonomous cars: object detection and tracking, lane marking detection and navigation, all pulling enormous amounts of data from a variety of sensors, not just cameras.

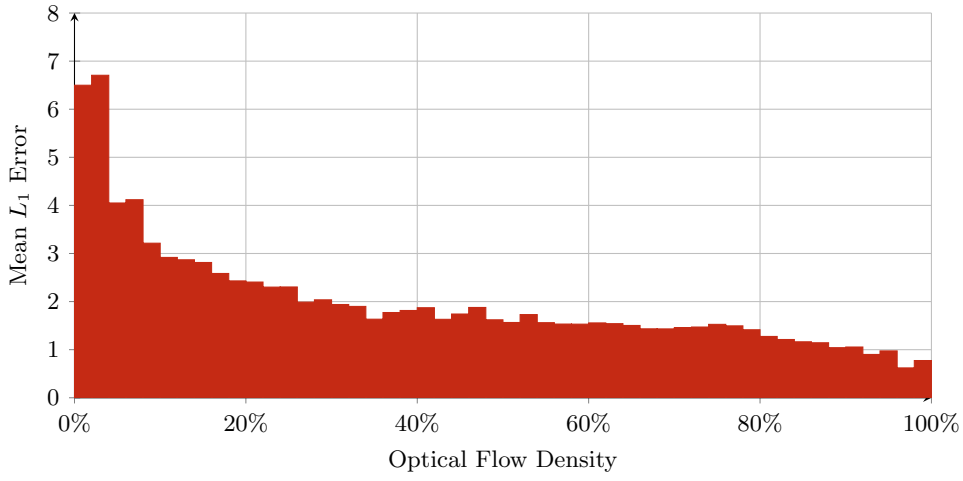


Figure 4.12: Mean L_1 Error on the testing set, binned by optical flow density.

Process	Runtime (ms)	Runs per second
Farnebäck	157	6.4
FlowNet2	276	3.6
Ours	19	52.6

Table 4.1: Processing times for calculating Farnebäck optical flow and a forward pass of FlowNet2 and our proposed CNN (averaged over 100 runs).

Nvidia is releasing a line up of system-on-chip products with deep learning and computer vision hardware acceleration [6] while minimising power consumption. Tesla are creating their own hardware from scratch, although with the same goals as Nvidia in mind [8]. An overlay on a video on Tesla's website indicates that their computer vision systems run at rates of 12 to 18 times a second. The results of calculating the a forward pass of our proposed CNN (Table 4.1) show that, when run on BC4, the CNN can make over 50 forward passes a second. For the calculation of Farnebäck optical flow, however, it is a different story, only managing to run at about 6 times a second, albeit on a laptop processor (Intel i5-7267U). FlowNet2 (Section 4.3.1), when run on BC4, is even slower. It is worth noting that, in all these cases, reducing the dimensions of the input image will speed up the runtime, although at the loss of information.

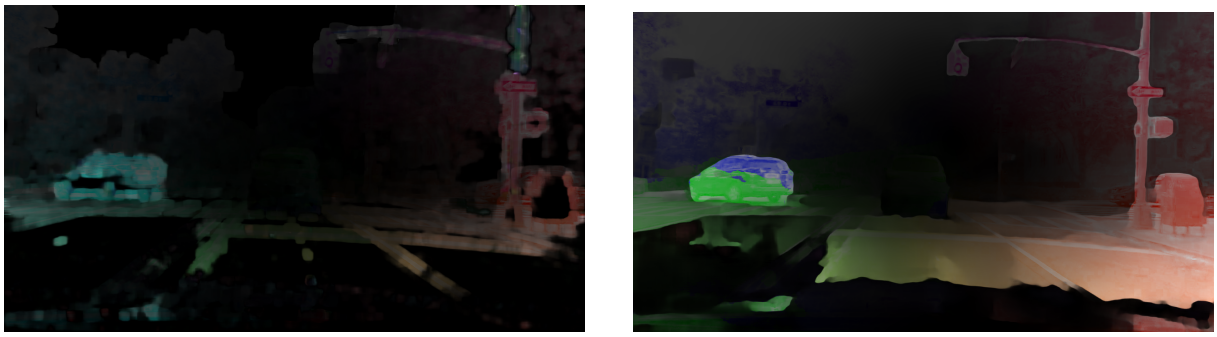
4.3 Future Work

4.3.1 CNNs for Optical Flow Estimation

The advances in the field of computer vision since the advent of CNNs have not been limited to such tasks as object detection or semantic segmentation only. Indeed, CNNs have shown promise at many tasks but most notably, from the point of view of this project, optical flow estimation. Several labelled datasets have been released to support the training of CNNs to calculate optical flow: Flying Chairs [25] and KiTTi [36] proving to be the most popular. Flying chairs is an artificial dataset in which images of chairs are superimposed onto background photos pulled from Flickr. The chairs and the background undergo independent planar motions which are used to generate the ground truth optical flow. The KiTTi dataset was touted as a contender to provide the speed data for this project but it also provides labelled data for optical flow estimation in the context of autonomous cars. Therefore, the results of CNN architectures on KiTTi is of particular interest if such an architecture was to be used to extend this project.

In 2015, Fischer et al. [26] proposed two CNN architectures to estimate optical flow: FlowNetSimple stacks the two input frames into one 6 channel tensor, whereas FlowNetCorr convolves the two input frames independently before combining the feature maps using a 'correlation layer' that performs multiplicative patch comparisons. The same authors noted that traditional state-of-the-art optical flow estimation algorithms are iterative - this includes the Farnebäck algorithm (see Section 2.1.1). FlowNet2 [46] was then proposed, stacking multiple FlowNetSimple and FlowNetCorr netwoks together to improve the estimation of large displacements. The authors of FlowNet2 provide a model trained on the Flying Chairs dataset that is available for download. Two frames from our custom split of BDD were fed as input to FlowNet2, the subsequent optical flow estimation is visualised in Figure 4.13. The code was edited such that the optical flow estimation was encoded in the same way as described in Section 3.2.3. It can be seen that FlowNet2 provides much greater coverage of the scene than the Farnebäck algorithm and appears to be much less noisy. It is certainly possible that if FlowNet2 was trained on the KiTTi dataset that it would produce significantly better results for this example scene and indeed, if this optical flow is used as the input to the network proposed by this project, it would perhaps result in more accurate speed estimations.

An inspection of the leaderboard for optical flow evaluation on the KiTTi dataset reveals a clear winner: Deep Rigid Scene Flow (DRISF), proposed by Ma et al. in 2019 [61]. This work focuses on the 3D scene flow in autonomous driving scenarios. For comparison, FlowNet2 places 50$^{\text{th}}$ on the leaderboard, recording an outlier percentage of 10.41%, over double the 4.73% produced by DRISF. Indeed, DRISF beats second place by 0.94% - an impressive achievement. However, this method is not an end to end system like that of FlowNet2; DRISF consists of three networks estimating optical flow, stereo information and the extraction of visual clues. The structure of the scene is the inferred and optimised to produce a



(a) Farnebäck Optical Flow.

(b) FlowNet2.

Figure 4.13: Visualisation of the optical flow produced by the Farnebäck algorithm and FlowNet2 for the same pair of source frames.

3D motion. Despite the performance of DRISF, it requires consecutive frames from two stereo cameras, which is not provided as part of BDD. The network that DRISF employs to calculate optical flow is PWC-Net, proposed by Sun et al. [77] from Nvidia and performs better than FlowNet2 on the KiTTi benchmark (46^{th} vs. 50^{th}). In summary, future work on this project involves the training of FlowNet2 or PWC-Net on the KiTTi dataset after which the optical flow can then be calculated for the frames extracted from our custom split of BDD. These optical flow estimations can then be used to train the CNN to predict the speed of the vehicle to a potential greater accuracy. The deep learning methods that have been mentioned provide greater coverage of the frame and compute more accurate optical flow estimations than the Farnebäck algorithm.

4.3.2 Dataset

It has been made clear throughout this report that the dataset used to train and test our CNN contains very few samples from high speeds and a rather non-uniform distribution of other speeds. Even with the problems that high speeds bring to optical flow accuracy and density, more data is a good thing. The more data available to start with, the more data will be left after filtering - that is, removing bad samples and fixing the distribution of the training data. For the task of speed estimation, we'd ideally like a uniform distribution of speeds in the training set, ensuring that the model is not inherently biased towards a small set of speeds. In the case of BDD, there are much fewer samples with high speed than low speed. One solution for this is to download many more BDD videos and hope that there are enough samples from higher speeds. Alternatively, data from a second dataset could be incorporated into the training set. This does not entirely solve the problem however, as by using the minimum amount of high speed video to produce the required number of samples for the training set, the diversity of these samples is still much worse than that of lower speeds, from which samples are extracted from a more diverse set of videos. Nonetheless, more samples from videos recorded at higher speeds would be required to boost the performance of the network.

4.3.3 Training and Validation Methods

In this project, the model has been trained and tested using single-fold cross validation; the model was trained on data in the training set alone and after training was evaluated on the testing set - the training and testing sets not having any data in common. The data was pre-split into the training and testing sets as part of BDD. This has the advantages of simplicity and training speed, but may be restricting the optimal performance of the network; there may be a split of the training and testing data that provides better performance. One solution is to employ k-fold cross validation, proposed by Mosteller and Tukey [64]. To improve robustness, k-fold cross validation splits all the data randomly into k equally sized parts. The model is then trained k times, each time selecting a different part to act as the testing set, training on the rest of the parts. Each of the models will therefore be trained on a unique training set and tested on a unique testing set, obtaining the final loss by averaging out the results of all k models. It is recommended and most common for k to be set to 10 for most applications. This method has the obvious drawback of training time increasing by a factor of k.

Another improvement that could be made to the training method is the employment of early stopping - first proposed by Morgan and Bourlard [63]. For early stopping, a third set of data is used, the validation set which is separate from the training and testing sets. The model is evaluated on this validation set every epoch (or every few epochs), recording the results. If the model starts to perform worse on the validation set or does not improve enough, training is stopped and the model evaluated on the testing set for a final result. For further robustness, this approach can be easily combined with k-fold cross validation by simply randomly selecting another part of the data to form the validation set, a different part for each of the k models. Early stopping acts as a reguliser, preventing overfitting on the training data as soon as the generalisation error increases.

A simple yet effective method of further increasing testing (and validation) accuracy is to smooth the speed predictions over several frames, as the previous predictions are usually a good indication of the current speed. The issue with using BDD data is that this is not possible: there is only a speed recording every second and smoothing predictions over seconds will fail to capture fast acceleration and deceleration. Capturing fast acceleration and deceleration whilst still smoothing results can be achieved by weighting more recent predictions higher (as is described in Section 2.2.1).

4.3.4 CNN-LSTMs

Frame by frame speed predictions are inherently temporal, each prediction depends heavily on the previous prediction - and even older predictions to an exponentially lesser extent. In the current model, each prediction is completely independent of every other prediction. One possible piece of future work is the addition of Long Short Term Memory (LSTM) modules into the network architecture. LSTMs, introduced by Hochreiter and Schmidhuber [45] are a kind of Recurrent Neural Network (RNN) that can capture long term dependencies across data series by 'storing' information from one prediction to the next. LSTMs are comprised of so called memory cells that take an input from the current time step, an input from their state at the previous time step and output their updated state. The information that is stored in an LSTM memory cell is regulated by several gates: a 'forget' gate decides how much of the current information is thrown away, an input gate determines how much of the new information fed into the cell should be kept and an output gate to determine what information is released to the rest of the network. LSTMs have proven extremely successful for a variety of tasks [51].

The LSTM memory cells as previously described only accept 1-D input. It might be desirable to replace the multiplicative gating mechanism with convolution instead, accepting a 3D input and producing a 3D output - also known as a ConvLSTM. Spatial features can now be remembered for future predictions, perhaps the previous movement of objects to help indicate the rate of acceleration or deceleration. These ConvLSTM layers can be incorporated directly into the current network architecture. The drawback for this project, and again for BDD video, is that the network would have to be trained using sequences of consecutive frames to be able to learn what data to keep in the ConvLSTM units, not possible with the data extracted from BDD.

4.3.5 Model Pruning

If our CNN for speed prediction were to be deployed into a real vehicle, it would be one of many such systems, all working together in perception and planning problems. These networks would have limited computational power and must run within strict power efficiency limitations. With this in mind, it is prudent that the network is as small as possible such that it occupies minimal space in memory and achieves rapid inference. One method of accomplishing this is called model pruning, first introduced by Han et al. [41], that learns only the important connections in the network. It is an iterative process by which the network is trained from scratch to learn which connections are important, the unimportant connections are removed and the network retrained to fine tune the weights of the remaining connections. By repeatedly applying this process, the authors were able to shrink two state of the art image classification networks: AlexNet [55] by 9× and VGG-16 [75] by 13× whilst maintaining the same accuracy.

Chapter 5

Conclusion

In this project, we have proposed a system for ego-vehicle speed estimation from monocular dash-cam video, with no prior knowledge of the scene in which the vehicle is situated. The Farnebäck optical flow is calculated between two consecutive frames, encoded as an image in the HSV colour space and input to our CNN with five convolutional layers and four fully connected layers which subsequently outputs the speed prediction. As a starting point, the CNN architecture proposed by Nvidia [13], designed for a similar task, has been reimplemented. This architecture has been optimised for better performance, using deep learning techniques such as batch normalisation and max pooling (Section 3.3.1).

The contextual background in the form of other works related to the task of speed estimation and state of the art solutions to autonomous driving perception problems has been outlined in Chapter 1. In addition, the technical background needed to understand Farnebäck optical flow and the improvements made to the Nvidia architecture has been described in Chapter 2.

Through the results outlined in Chapter 3, we have shown that our proposal is a promising start and performs much better than the Nvidia architecture, achieving a Mean L_1 Error of 1.571 ms^{-1} on the testing set. Perhaps surprisingly, the network performs as well in all the weather conditions and times of day, however it performs markedly worse in highway scenes. Further analysis in Chapter 4 identifies clear areas of weakness: Farnebäck optical flow struggles with large displacements at high speeds, the relative motion of vehicles can give misleading information to the network and there is no way to separate reversing video. In addition, runtime performance and memory requirements must also be considered for deployment onto the limited computing power present in autonomous vehicles. Perhaps the most limiting factor is the availability of high speeds in the training set, something that should be addressed to achieve lower errors. We have proposed further work that could potentially improve the performance of our system, including using a CNN to calculate optical flow, different training and validation techniques and incorporating some recurrent structure to the network in the form of a CNN-LSTM, to take advantage of the temporal nature of speed.

5.1 Contributions

Speed and Frame Extraction from BDD. We have proposed a novel pipeline for the extraction of speed recordings and the corresponding frames for videos in the BDD dataset (Section 3.2.3). These frames are then further processed to calculate the optical flow between the two consecutive frames extracted for each speed recording, before being uploaded to BC4.

Reimplementation of the Nvidia Architecture. We have reimplemented the Nvidia CNN architecture [13] from scratch, training and evaluating it on our custom split of BDD. We found that in general the network performs very poorly in all conditions.

Improved CNN Architecture. We have proposed several improvements (Section 3.3.1) to the Nvidia architecture; finding through an ablation study that each of the improvements positively impacts the performance of the network. Our improved network greatly outperforms Nvidia's in all conditions and is a good first step towards a robust and accurate standalone system for ego-vehicle speed estimation using a single network.

Bibliography

[1] Berkeley deep drive website. https://bdd-data.berkeley.edu/index.html. Accessed: 17-04-2020.

[2] The darpa grand challenge: Ten years later. https://www.darpa.mil/news-events/2014-03-13. Accessed: 21-02-2020.

[3] Gps accuracy. https://www.gps.gov/systems/gps/performance/accuracy/. Accessed: 18-02-2020.

[4] The man who invented the self-driving car (in 1986). https://www.politico.eu/article/delf-driving-car-born-1986-ernst-dickmanns-mercedes/. Accessed: 20-02-2020.

[5] Mapillary vistas dataset. https://www.mapillary.com/dataset/vistas. Accessed: 07-05-2020.

[6] Nvidia xavier. https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-agx-xavier/. Accessed: 03-05-2020.

[7] Tesla autopilot. https://www.tesla.com/en_GB/autopilot. Accessed: 18-02-2020.

[8] Tesla autopilot ai. https://www.tesla.com/en_GB/autopilotAI. Accessed: 03-05-2020.

[9] Workshop on autonomous driving, cvpr 2019. https://sites.google.com/view/wad2019/challenge. Accessed: 08-05-2020.

[10] Workshop on autonomous driving, cvpr 2020. http://cvpr2020.wad.vision/. Accessed: 08-05-2020.

[11] Workshop on autonomous driving, eccv 2020. https://sites.google.com/view/pad2020. Accessed: 08-05-2020.

[12] Workshop on autonomous driving, iccv 2019. http://www.wad.ai. Accessed: 08-05-2020.

[13] 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.

[14] Tom Botterill, Richard Green, and Steven Mills. A bag-of-words speedometer for single camera slam. In *2009 24th International Conference Image and Vision Computing New Zealand*, pages 91–96. IEEE, 2009.

[15] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 111–118, 2010.

[16] Alberto Broggi. *Automatic vehicle guidance: the experience of the ARGO autonomous vehicle*. World Scientific, 1999.

[17] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. *arXiv preprint arXiv:1903.11027*, 2019.

[18] Ming-Fang Chang, John W Lambert, Patsorn Sangkloy, Jagjeet Singh, Slawomir Bak, Andrew Hartnett, De Wang, Peter Carr, Simon Lucey, Deva Ramanan, and James Hays. Argoverse: 3d tracking and forecasting with rich maps. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[19] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1907–1915, 2017.

[20] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[21] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[22] Youjing Cui and Shuzhi Sam Ge. Autonomous vehicle positioning with gps in urban canyon environments. *IEEE transactions on robotics and automation*, 19(1):15–25, 2003.

[23] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1052–1067, 2007.

[24] Hendrik Deusch, Jürgen Wiest, Stephan Reuter, Magdalena Szczot, Marcus Konrad, and Klaus Dietmayer. A random finite set approach to multiple lane detection. In *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pages 270–275. IEEE, 2012.

[25] A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazırbaş, V. Golkov, P. v.d. Smagt, D. Cremers, and T. Brox. Flownet: Learning optical flow with convolutional networks. In *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[26] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015.

[27] Xinxin Du, Marcelo H Ang, Sertac Karaman, and Daniela Rus. A general pipeline for 3d detection of vehicles. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3194–3200. IEEE, 2018.

[28] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[29] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[30] Gunnar Farnebäck. *Polynomial expansion for orientation and motion estimation.* PhD thesis, Linköping University Electronic Press, 2002.

[31] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Scandinavian conference on Image analysis*, pages 363–370. Springer, 2003.

[32] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Detect to track and track to detect. *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[33] Duncan P Frost, Olaf Kähler, and David W Murray. Object-aware bundle adjustment for correcting monocular scale drift. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4770–4776. IEEE, 2016.

[34] Dorian Gálvez-López, Marta Salas, Juan D Tardós, and JMM Montiel. Real-time monocular object slam. *Robotics and Autonomous Systems*, 75:435–449, 2016.

[35] Norman Bel Geddes. *Magic motorways.* Read Books Ltd, 2011.

[36] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[37] Jakob Geyer, Yohannes Kassahun, Mentar Mahmudi, Xavier Ricou, Rupesh Durgesh, Andrew S. Chung, Lorenz Hauswald, Viet Hoang Pham, Maximilian Mühlegg, Sebastian Dorn, Tiffany Fernandez, Martin Jänicke, Sudesh Mirashi, Chiragkumar Savani, Martin Sturm, Oleksandr Vorobiov, Martin Oelker, Sebastian Garreis, and Peter Schuberth. A2D2: Audi Autonomous Driving Dataset. 2020.

[38] Tobias Glasmachers. Limits of end-to-end learning. *arXiv preprint arXiv:1704.08305*, 2017.

[39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[40] Inhwan Han. Car speed estimation based on cross-ratio using video data of car-mounted camera (black box). *Forensic science international*, 269:89–96, 2016.

[41] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[42] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[44] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), 2012.

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

[46] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2462–2470, 2017.

[47] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[48] Joel Janai, Fatma Güney, Aseem Behl, and Andreas Geiger. Computer vision for autonomous vehicles: Problems, datasets and state-of-the-art. *arXiv preprint arXiv:1704.05519*, 2017.

[49] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[50] Heechul Jung, Junggon Min, and Junmo Kim. An efficient lane detection algorithm for lane departure detection. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 976–981. IEEE, 2013.

[51] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 21:23, 2015.

[52] R. Kesten, M. Usman, J. Houston, T. Pandya, K. Nadhamuni, A. Ferreira, M. Yuan, B. Low, A. Jain, P. Ondruska, S. Omari, S. Shah, A. Kulkarni, A. Kazakova, C. Tao, L. Platinsky, W. Jiang, and V. Shet. Lyft level 5 av dataset 2019. urlhttps://level5.lyft.com/dataset/, 2019.

[53] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[54] Alexander Kirillov, Kaiming He, Ross Girshick, Carsten Rother, and Piotr Dollár. Panoptic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9404–9413, 2019.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[56] 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.

[57] Seokju Lee, Junsik Kim, Jae Shin Yoon, Seunghak Shin, Oleksandr Bailo, Namil Kim, Tae-Hee Lee, Hyun Seok Hong, Seung-Hoon Han, and In So Kweon. Vpgnet: Vanishing point guided network for lane and road marking detection and recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 1947–1955, 2017.

[58] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the disharmony between dropout and batch normalization by variance shift. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2682–2690, 2019.

[59] Justin Liang, Namdar Homayounfar, Wei-Chiu Ma, Yuwen Xiong, Rui Hu, and Raquel Urtasun. Polytransform: Deep polygon transformer for instance segmentation. *arXiv preprint arXiv:1912.02801*, 2019.

[60] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. 1981.

[61] Wei-Chiu Ma, Shenlong Wang, Rui Hu, Yuwen Xiong, and Raquel Urtasun. Deep rigid instance scene flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3614–3622, 2019.

[62] Michael Meyer and Georg Kuschk. Automotive radar dataset for deep learning based 3d object detection. In *2019 16th European Radar Conference (EuRAD)*, pages 129–132. IEEE, 2019.

[63] Nelson Morgan and Hervé Bourlard. Generalization and parameter estimation in feedforward nets: Some experiments. In *Advances in neural information processing systems*, pages 630–637, 1990.

[64] Frederick Mosteller and John W Tukey. Data analysis, including statistics. *Handbook of social psychology*, 2:80–203, 1968.

[65] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[66] Constantine Papageorgiou and Tomaso Poggio. A trainable system for object detection. *International journal of computer vision*, 38(1):15–33, 2000.

[67] Seo Young Park and Yufeng Liu. Robust penalized logistic regression with truncated loss functions. *Canadian Journal of Statistics*, 39(2):300–323, 2011.

[68] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[69] Travis Portz, Li Zhang, and Hongrui Jiang. Optical flow in the presence of spatially-varying motion blur. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1752–1759. IEEE, 2012.

[70] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[71] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[72] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[73] Samuel Schulter, Paul Vernaza, Wongun Choi, and Manmohan Chandraker. Deep network flow for multi-object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6951–6960, 2017.

[74] Sarthak Sharma, Junaid Ahmed Ansari, J Krishna Murthy, and K Madhava Krishna. Beyond pixels: Leveraging geometry and shape cues for online multi-object tracking. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3508–3515. IEEE, 2018.

[75] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[76] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[77] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan Kautz. PWC-Net: CNNs for optical flow using pyramid, warping, and cost volume. 2018.

[78] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. Scalability in perception for autonomous driving: Waymo open dataset, 2019.

[79] Chris Urmson, J Andrew Bagnell, Christopher Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. Tartan racing: A multi-modal approach to the darpa urban challenge. 2007.

[80] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[81] Matthew B Wall and Andrew T Smith. The representation of egomotion in the human brain. *Current biology*, 18(3):191–194, 2008.

[82] Peng Wang, Xinyu Huang, Xinjing Cheng, Dingfu Zhou, Qichuan Geng, and Ruigang Yang. The apolloscape open dataset for autonomous driving and its application. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[83] Zining Wang, Wei Zhan, and Masayoshi Tomizuka. Fusing bird's eye view lidar point cloud and front view camera image for 3d object detection. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–6. IEEE, 2018.

[84] R Warren, DH Owen, and LJ Hettinger. Separation of the contributions of optical flow rate and edge rate on the perception of egospeed acceleration. *Optical flow and texture variables useful in simulation self motion.(II).(Interim Tech. Rep. for Grant No. AFOSR-81-0078, pp. 23-76). Columbus, OH: The Ohio State University, Department of Psychology*, 1982.

[85] William H Warren and Daniel J Hannon. Direction of self-motion is perceived from optical flow. *Nature*, 336(6195):162–163, 1988.

[86] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. Bdd100k: A diverse driving dataset for heterogeneous multitask learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[87] Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl. Tracking objects as points, 2020.

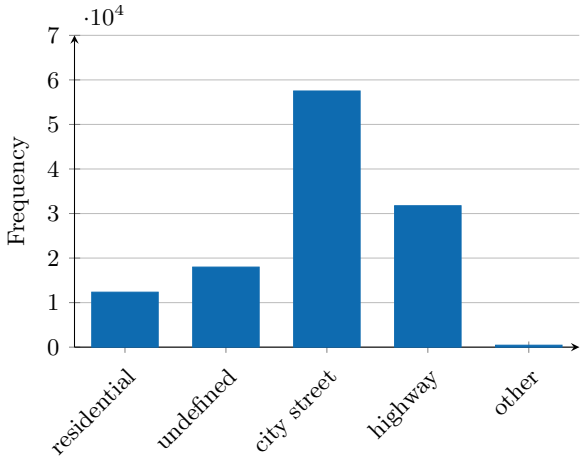
[88] Yi-Tong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *IEEE International Conference on Neural Networks*, volume 1998, pages 71–78, 1988.

[89] Yi Zhu, Karan Sapra, Fitsum A Reda, Kevin J Shih, Shawn Newsam, Andrew Tao, and Bryan Catanzaro. Improving semantic segmentation via video propagation and label relaxation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8856–8865, 2019.

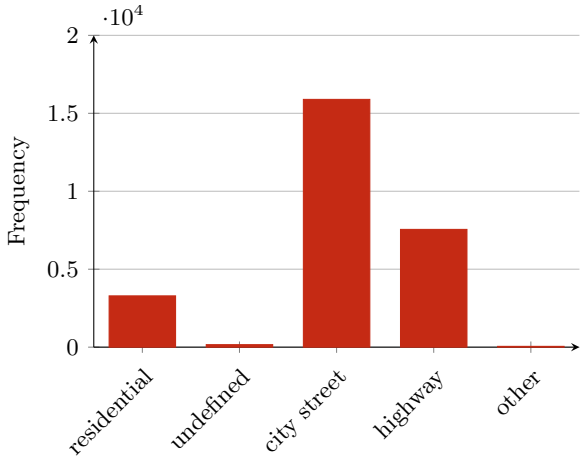
Appendix A

Dataset Conditions

The following graphs plot the number of samples in the training and testing sets by the scene type, weather condition and time of day.

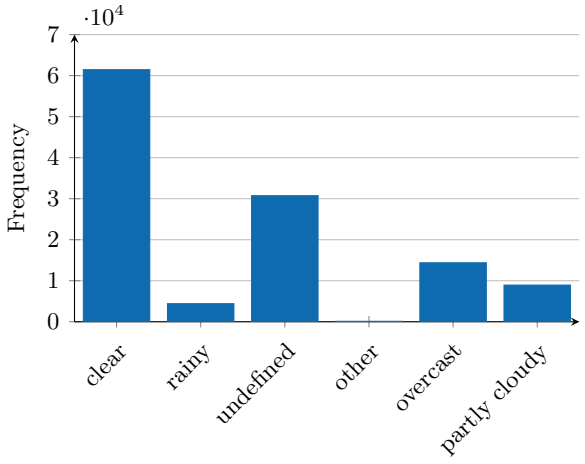


(a) Training set scenes.

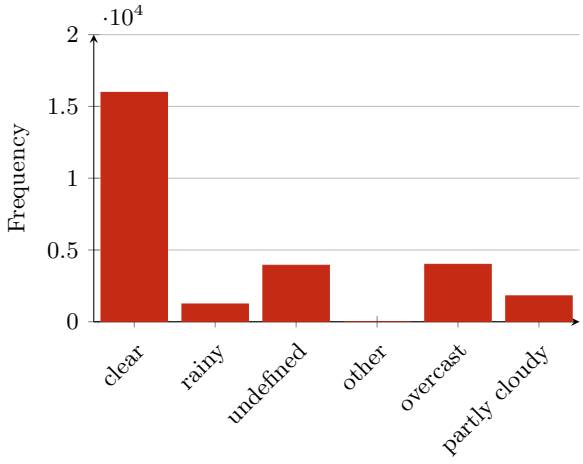


(b) Testing set scenes.

Figure A.1: Distribution of the scenes in the training and testing sets (other = {gas station, parking lot, tunnel}).

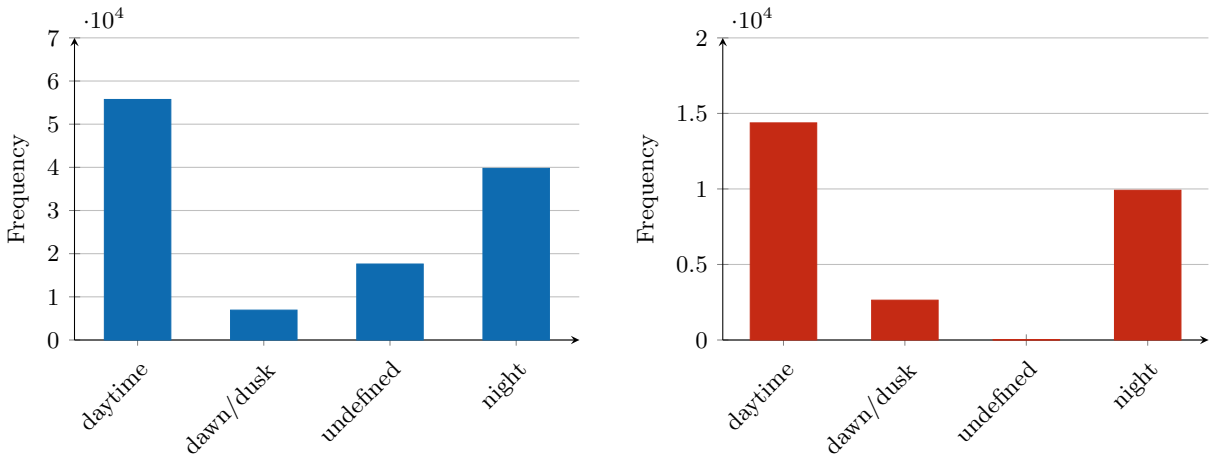


(a) Training set weather conditions.



(b) Testing set weather conditions.

Figure A.2: Distribution of the weather conditions in the training and testing sets (other = {foggy, snowy}).



(a) Training set times.

(b) Testing set times.

Figure A.3: Distribution of the time conditions in the training and testing sets.

Appendix B

Nvidia's CNN Architecture

The following image is the network architecture diagram presented in Nvidia's paper titled "End to End Learning for Self-Driving Cars" [13] that formed the basis for the CNN presented in Section 3.3.

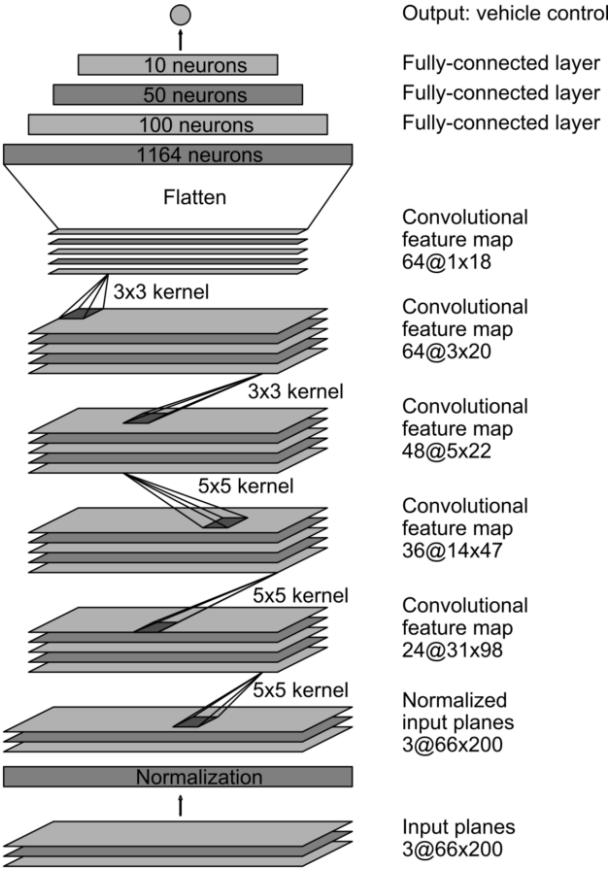


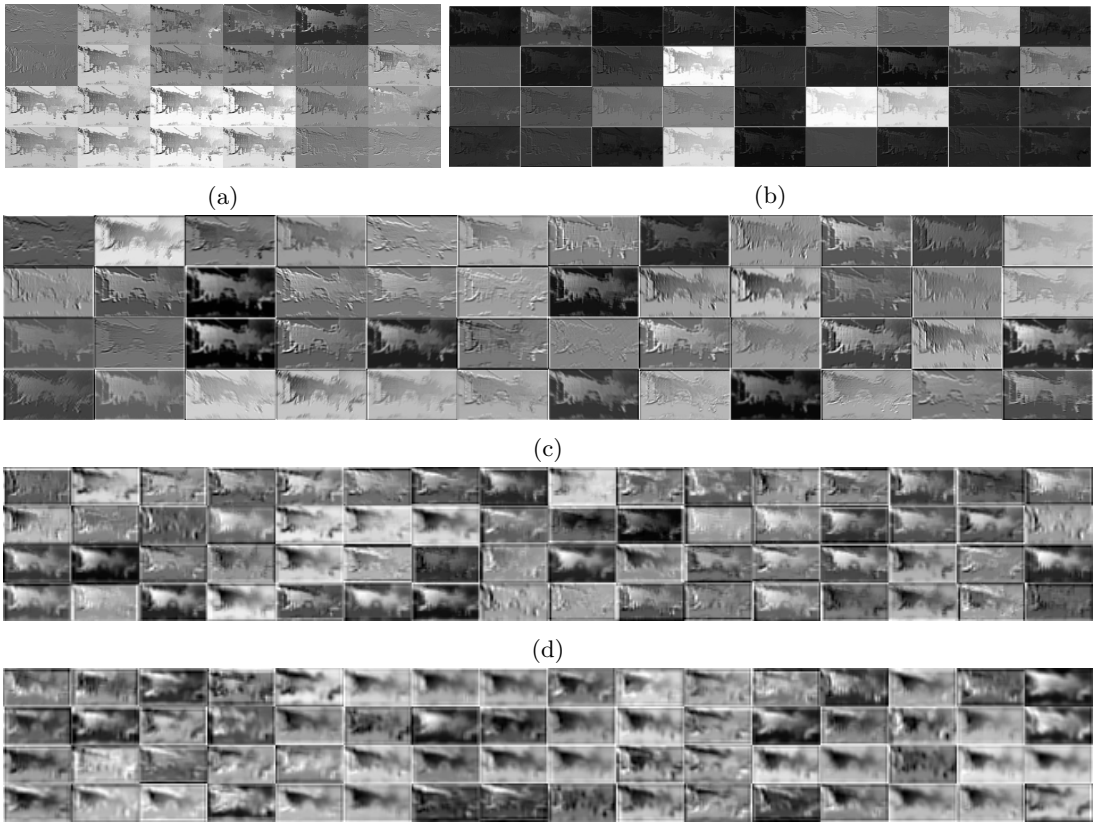
Figure B.1: Nvidia's CNN architecture.

Appendix C

Example Feature Maps

(a) (b)

Figure C.1: A sample frame and corresponding optical flow image from the training set.



(a) (b)

(c)

(d)

(e)

Figure C.2: The feature maps produced by a forward pass of Figure C.1b. (a)-(e) Conv1-5.