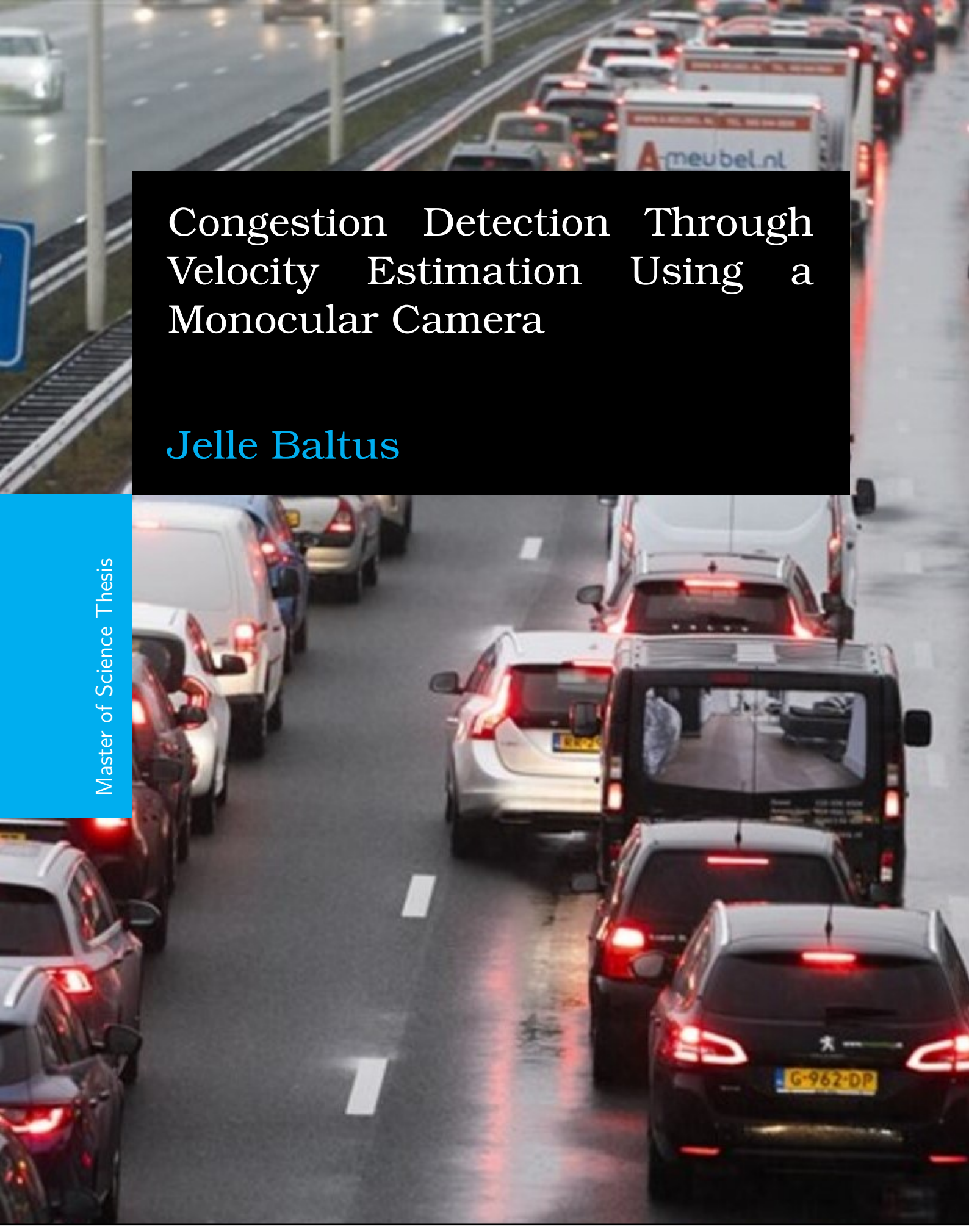

Congestion Detection Through Velocity Estimation Using a Monocular Camera

Jelle Baltus

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Congestion Detection Through Velocity Estimation Using a Monocular Camera

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Jelle Baltus

March 18, 2024

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

This thesis report aims to answer the following research question: “Is it possible to estimate relative velocities of vehicles surrounding the ego vehicle using a monocular camera with such an accuracy that meaningful conclusions can be made about the current traffic state?” To answer this question, a velocity estimation algorithm is developed in three major parts: object detection, object tracking with detections and velocity estimation using tracked 2D objects. For the detection part, a version of the YOLOv3 (You Only Look Once version 3) single shot detection neural network is used. For object tracking with detections, the Simple Online and Realtime Tracking (SORT) algorithm is used. The last part, velocity estimation using tracked 2D objects, a state-of-the-art method using a neural network is compared to a novel proposed method, using a 2D to 3D map in combination with a kalman filter using a constant velocity model. The results of the detection and tracking parts were good enough to reason that they are used as a base of the velocity estimation algorithm. When comparing the-state-of-the-art velocity estimation algorithm and the novel approach, the errors of the novel approach were significantly higher, and the results of the state-of-the-art methods could not be replicated. This means that the research question of this thesis can be answered with yes, it is possible to estimate relative velocities of surrounding vehicles, however the resulting estimation errors are too high to make meaningful conclusions about the current traffic state.

Table of Contents

List of Figures

List of Tables

Preface

This document is the final report of my Master of Science graduation thesis for the study of Systems and Control at the Delft University of Technology. The main reason for this topic, velocity estimation through camera images, is gaining knowledge on the inner workings of neural networks, as they are being used to solve more and more complex tasks by the day. For example, when I started with this thesis, no one had ever heard of chatGPT and now its used worldwide by millions of people. To understand even a small bit of the algorithms behind these neural networks is very valuable knowledge in my opinion. However, neural networks alone is not necessarily a Systems and Control topic, so I decided to combine it with a problem that was: traffic control, more specifically congestion detection. Because if congestions can be detected faster and with more accuracy, a country with a complex highway infrastructure, such as the Netherlands could benefit greatly from it.

So hopefully, some of the solutions presented will be implemented in some way or form, so that everyone will have to endure less traffic jams in the future.

Acknowledgements

I would like to thank my supervisor Dr.ing. Sergio Grammatico for his assistance during the writing of this thesis and my family for supporting me along the way.

Delft, University of Technology
March 18, 2024

Jelle Baltus

Chapter 1

Introduction

Google Maps is a massively popular map service: In 2019, Google itself [27] stated that more than a billion people worldwide use Google Maps every month and more than 5 million apps and websites have a form of google maps integrated into them. One of the main features of Maps is giving users the fastest route to their chosen destination given the current traffic state. This is done by making a prediction of the current traffic state, combining historical traffic data of possibles routes and the live positions and derived speed of mobile phones on that route that consented to share this information with Google [28].

However, with the massive amounts of historical knowledge on traffic conditions and live user data collected, it is still not 100% accurate. A humorous example of this is when a German artist called Simon Weckert used a hand cart with 99 smart phones sharing their position data to trick Google Maps [29]. The roads where the cart was driven on, appeared congested on Maps, when in truth there was little to no traffic at the time. This caused Google Maps to mistakenly redirect their users.

The amount of false positives, such as the one created by Mr. Weckert, could be mitigated with some techniques already used in autonomous driving: an autonomous vehicle uses sensors such as depth cameras, Lidar sensors and Radar sensors to locate objects around them and calculate their relative speed. This information is then used to navigate the autonomous vehicle safely through traffic. These relative speeds can give Maps a lot more traffic information than just the speed of the user: one vehicle can drive slow because of multiple reasons, such as unwell driver, breakdown, or the road ahead is congested. But if multiple vehicles are driving slow, then the cause is almost certain to be congested roads.

The main problem with this live traffic data solution is that the sensors which can perceive depth are quite expensive, even though the prices are going down significantly the past few years. It can be argued that some car manufacturers, such as Tesla, already have sensors installed into their latest models for some form of autonomous driving. However this is still a small percentage of the vehicles on the road today: a report from the European Automobile Manufacturers' Association (ACEA) [30] states that in 2021 the average age of passenger cars in the European Union is 12 years, when autonomous driving technology was not yet widely available in consumer vehicles.

Another solution that can be proposed is to use static cameras alongside the roads with computer vision algorithms to estimate the traffic state. Normal cameras are significantly cheaper than LIDARs and depth cameras and examples of this can be found alongside highways in the Netherlands, where cameras are used by the government to monitor the traffic state. A problem that rises with this is that to cover the whole road network will be a very expensive endeavour, not even mentioning the legal and ethical implications that a company as Google has to go through to place cameras along the roads or use footage from existing cameras.

To have the best of both worlds, the two solutions above can be combined into one: mount a camera onboard a vehicle and use computer vision algorithms to estimate the velocities of surrounding vehicles. This way existing cars do not have to be severely adapted to accommodate the needed technology. For example, a dash cam that is already becoming more and more popular for insurance purposes, would suffice.

This thesis proposes an algorithm using only dash camera footage to estimate the relative velocity of vehicles captured by the camera in real-time and compares them to similar solutions found while exploring research literature. This can be stated more formally into the following research question which this thesis aims to answer:

"Is it possible to estimate relative velocities of vehicles surrounding the ego vehicle using a monocular camera with such an accuracy that meaningful conclusions can be made about the current traffic state?"

1-1 Main Topics of the Thesis

The proposed velocity estimation algorithm can be broken down into three major parts. The first part is a neural network using the video footage of an onboard camera to extract the captured objects into bounding boxes and classify them, abbreviated in this thesis as object detection. The second part is using these extracted objects and attempt to track them over time as accurate as possible, abbreviated in this thesis as object tracking. The last part is using these object tracks to estimate their velocity as accurate as possible, abbreviated in this thesis as velocity estimation.

1-2 Structure of the Thesis

After this introduction, this thesis will continue with a summary of the background research done for the literature survey which can be found in Chapter 2. After this summary, the implementations of the three main parts of the velocity estimation algorithm (described in Section 1-1) is showcased in Chapter 3. The results of these implementations are discussed and compared in Chapter 4 and the thesis concludes with a formal conclusion in Chapter 5.

Chapter 2

Background Research

As already stated in Section 1-2, this chapter will showcase the relevant background research done for this thesis by taking excerpts from a literature survey done at the beginning [31]. Most of the literature research was done in the fields of convolutional neural networks, as the techniques developed in these types of neural networks is used in the current state-of-the-art object detectors, and autonomous driving datasets, as creating the data needed for correctly train object detection networks is out of the scope of this thesis project. Section 2-1 will go more into detail about how convolutional neural networks are used and highlight some important examples of these networks. Section 2-2 showcases the KITTI dataset and section 2-3 the Berkely Deep Drive 100K dataset, both been part of the background research. The KITTI dataset will be used to validate the object tracking algorithm, while the Berkely Deep Drive dataset will be used to train the object detection part of the velocity estimation algorithm.

2-1 Object Detection using Convolutional Neural Networks

This section will explain what convolutional neural networks in essence are, how they work and give some examples of popular convolutional neural networks. This is done by first giving some basic neural network principles, such as the structure, and some basic principles on neural network training. After this, the differences between a standard neural network and a convolutional neural network are shown. The section ends with a few examples of popular convolutional neural networks, also shown in the literature survey [31].

2-1-1 Introduction to Artificial Neural Networks

An artificial neural network is in essence a function that maps an input $\mathbf{x}$ to an output $\mathbf{y}$, similar to conventional mathematical functions. This is where the similarities end however, as artificial neural networks have unique structures that mimic biological neural networks, where individual neurons are connected to each other, forming a complex network. Such a

biological network can use different senses as input and perform actions based on these inputs, mostly moving specific muscles in a body.

In contrary to biological neural networks, artificial neural networks can have multitude of tasks that they can perform. A few examples are:

- Classifying data points into different categories. This data can be images, sensor data or user-inputted numbers and the output is a specific class from a list of known classes.
- Approximating an unknown mathematical function, using only the given inputs and percieved outputs
- A controller that executes an optimal policy based on an output of a system. The input is the output of the controlled system (in the form of sensor data) and the output is the input of the controlled system.

Most of these examples follow a basic structure that can be seen in Figure 2-1:

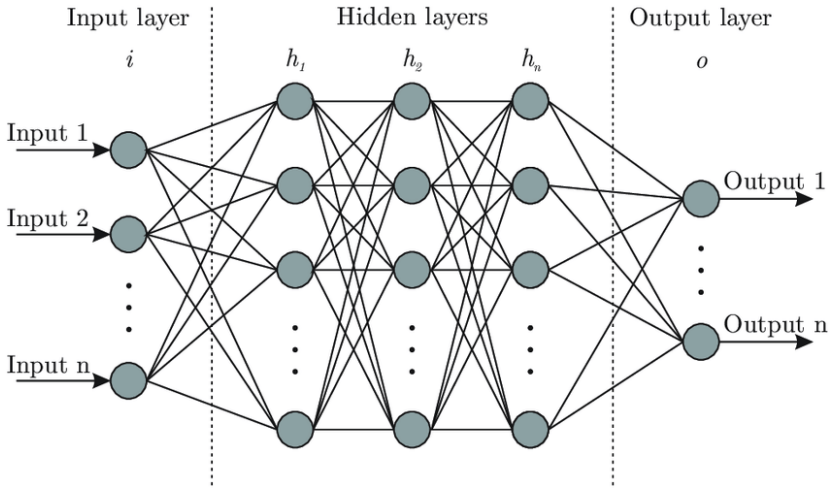

Figure 2-1: An example of an artificial neural network [1]

The network above is divided in layers, which is one of the cornerstones of artificial neural networks: all networks consist of layers of artificial neurons stacked on top of each other. In this example, the network consists of three parts: an input layer, hidden layers and an output layer. The input layer simply contains all the input variables of the network. The middle (also called hidden) layers accept inputs from the nodes of the previous layer and compute an output, which is used by the next layer of nodes. These layers are called hidden, because they can only be observed when looking inside the network: they are effectively "hidden" from the outside world. The amount of hidden layers present in a network determines its depth, since a network always requires and input and output layer. When a network has a significant amount of these hidden layers, they will be classified has "deep neural networks". The output layer accepts the outputs from the last hidden layer (or input layer when no hidden layers are present) and computes the output of the network. This output is almost always a vector with real numbers, that can be interpreted in a specific way or be an input for an external algorithm.

As already stated, each neural network layer consists of artificial neurons. These neurons can process their inputs into an output given their internal parameters. Figure 2-2 contains a schematic of such an individual node.

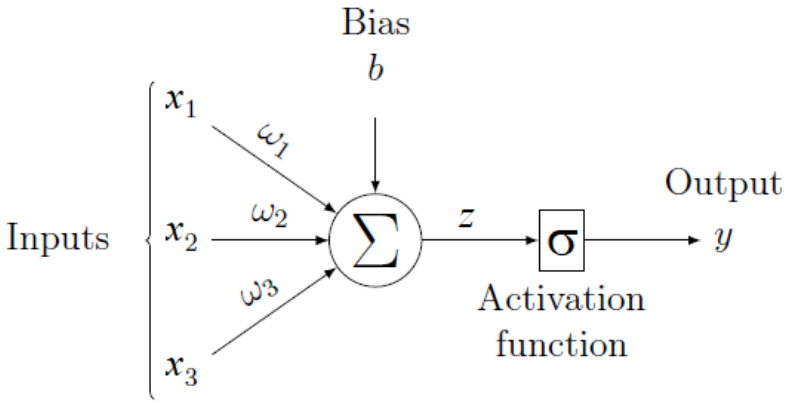

Figure 2-2: Schematic of a neural network node [2]

$$z = \mathbf{w}^T \mathbf{x} + b \tag{2-1}$$

The inputs of all the neuron, indicated with $\mathbf{x} = [x_1\, x_2\, \ldots\, x_n]$ are first weighted with a weight vector $\mathbf{w} = [w_1\, w_2\, \ldots\, w_n]$. These weights represent how important certain inputs are to the neuron and are one of the two sources of neuron parameters, the other being the bias term b. These parameters will be altered during the training phase of the network.

The weighted inputs are summed up together with the bias term (see equation 2-1) to form the input of the activation function. This activation function determines the output of the neuron and is the only source of non-linearity in the neuron and therefore in the whole network. Without this activation function, neural networks would just represent a complex string of vector and matrix equations. The activation function that is used in most modern neural networks is the Rectified Linear Unit function, often abbreviated to the ReLU function, which can be seen in equation 2-2.

$$f(z) = \begin{cases} 0, & z \leq 0 \\ z, & z > 0 \end{cases} = \max\{0, z\} \tag{2-2}$$

This function outputs zero when a negative input is given and behaves like a linear function when a positive input is given, making the function non-linear(precisely piece-wise linear), but still behaving like a linear function most of the time. This last property will help in the training stage, as can be seen later in this section.

However, not all neurons in a network use the ReLU function as activation function: the output layer of classification networks. This layer is also known as the classification layer and gives an output in the form of a vector, where the index of a vector entry represents a class in which the input can be classified and the entry itself represents how confident the network is that the input belongs to a specific class. The most common activation function used in

classification layers nowadays is the softmax function, for which the formula can be found in equation 2-3:

$$f(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \text{ for } i = 1, \ldots, K \text{ and} \tag{2-3}$$

This activation function takes the exponent of its input, which is calculated in the same way seen in figure 2-2 and normalises it against the other outputs of the activation layer. This normalisation step is the reason why this activation function needs a vector input instead of a scalar input used by the ReLU function. The reason behind the normalisation step is that a classification layer with a softmax activation function has an output vector with entries bounded between zero and one and all of the entries sum up to one. This can be interpreted rather intuitively as a probability distribution over the available classes. A probability vector is also excellent for usage in gradient-based learning, which is introduced in the next section along with the algorithms that are commonly used for training networks.

2-1-2 Training of Artificial Neural Networks

The training phase of the neural network is where the network achieves its desired performance by giving it inputs, compare the output of the network to the desired output of the network and then update the weights and biases of the neurons in such a way that the performance of the network is improved. This type of learning is called supervised learning. Other types of learning are unsupervised learning and reinforcement learning, however these are out of the scope of this thesis, as the types convolutional neural networks used in object detection are trained with a supervised learning method.

Most supervised learning algorithms use some sort of loss function to compare the current output of the neural network with the output the network should give when it is functioning optimally. An example of such a loss function can be found in Equation 2-4:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{c=1}^{K} y_c \log(\hat{y}_c) \tag{2-4}$$

Where $\mathbf{y}$ is the expected output (also called the ground truth), $\hat{\mathbf{y}}$ is the output of the network and c is the specific class of the output. This particular loss function is called a cross entropy loss function, which is mostly used when the neural network outputs a probability distribution, such as when classifying objects in images. Another common loss function is the Mean Squared Error function, common abbreviated as MSE function. An example of this function can be seen in Equation 2-5:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{K} \sum_{c=1}^{K} (y_c - \hat{y}_c)^2 \tag{2-5}$$

The loss function can also be augmented with a regularization term, which penalize large weights in the network. This will prevent the weights to grow too large, which causes the gradient to increase exponentially and making the training process unstable. There are currently two norms in use for regularizing weights, the L1 norm, which is the absolute distance between weights and the L2 norm, which is the euclidean distance between weights. After

determining the network structure and choosing an appropriate network output with a fitting loss function, an input can be given to the network and a loss scalar will be calculated by the network. This is called the forward pass and is the first calculation done by the network at the training stage. The second calculation is the backward pass. Beginning from the loss function, the derivative of the loss function with respect to the network weights are calculated via the chain rule: Each network node calculates their local derivatives with respect to their weights, biases and inputs. These gradients are then added to the gradient that is passed to the node from the output side. A visualisation of this process can be seen in figure 2-3.

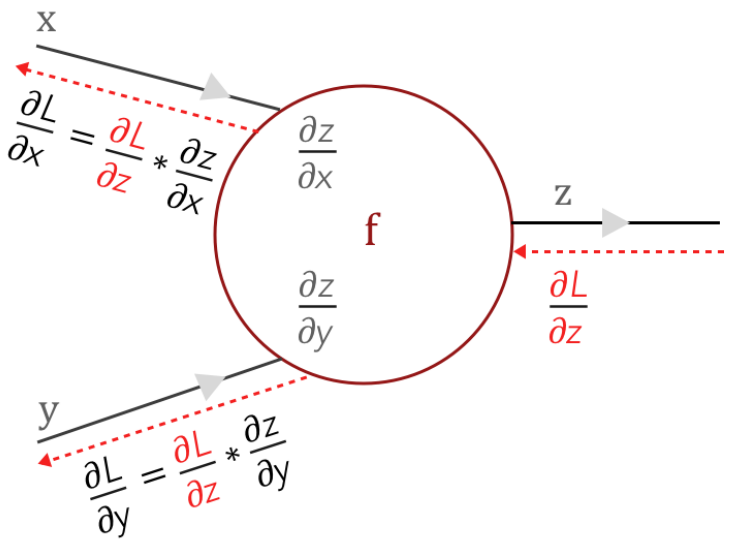

Figure 2-3: Backpropagation visualised on a neural network node [3]

The gradient with respect to the input variables are passed to the previous layer of the network (or ignored if the layer in question is the input layer of the network). This is why the ReLU activation function from Equation 2-2 is widely used nowadays: the gradient of the ReLU is either one, when the input is larger than zero, or otherwise zero. The gradients with respect to the weights and biases are put in one large gradient vector which is used for the next step in training: updating the weights of the neural network in an algorithmic way using the calculated gradients.

Gradient Descent Algorithms

Updating the weights of a neural network with a given gradient from the loss function is done with a gradient descent algorithm: a first-order optimization method that finds a local minimum of a differentiable function. This is done by iterative calculating a gradient with respect to the relevant variables, setting a step in the opposite direction of the gradient and calculating the gradient in the new point, until a local minimum is reached. This descent algorithm can be applied to a loss function of a neural network, which results in Equation 2-6:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \epsilon_n \nabla_{\mathbf{w}} L(f(\mathbf{x}, \mathbf{w}_n), \mathbf{y}) \tag{2-6}$$

Where $\mathbf{w}$ is the vector containing the network weights, n the iteration count, $f(\mathbf{x}, \mathbf{w})$ the output of the neural network given the used dataset, $\mathbf{y}$ the ground truths of the dataset, L the loss function and ϵ the step size, also called the learning rate when working with neural networks. The latter is an important hyperparameter and can have a significant impact on the network performance. The learning rate can be fixed or adaptive. A common practice of using an adaptive learning rate is to decrease the learning rate when the iteration count increases, which is called learning rate decay. The other way around often leads to weights increasing exponentially in size, which is the reason it is rarely used.

When training a network with a large amount of weights, a significant amount of data is needed for creating a network with acceptable performance. This leads to extreme long training times with normal gradient descent, since the algorithm has to calculate the gradient for every data point before upgrading the network weights. This becomes unfeasible for modern datasets, which can hold up to 100000 data entries and sometimes even more. The solution to this is the Stochastic Gradient Descent (SGD) algorithm, described in equation 2-7:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \frac{\epsilon_n}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}_n, \mathbf{y}^{(i)}) \tag{2-7}$$

SGD randomly selects m amount of data points, which is called a mini-batch: an important hyperparameter that needs to be chosen properly for a good training result. The average gradient is then calculated and used in the same way as with normal gradient descent. It is proven that if the examples in the mini-batch are independent and identically distributed and the learning rate used adheres to the conditions of equation 2-8, SGD will converge to a local minimum.

$$\sum_{n=1}^{\infty} \epsilon_n = \infty, \ \sum_{n=1}^{\infty} \epsilon_n^2 < \infty \tag{2-8}$$

Building on the SGD algorithm, more learning algorithms were developed that perform even better than the SGD algorithm. This section showcases three popular variants: momentum, RMSprop (Root Mean Square Propagation) and ADAM (Adaptive Moment Estimation).

Sometimes, learning algorithms such as SGD do not converge fast enough to a local minima. This can be because of almost flat gradients, high curves in the weight space that make the gradient descent algorithm bounce around. The momentum algorithm tries to accelerate or decelerate learning in those cases by introducing a velocity term that represents the speed and direction in which the weights from the previous training step were going. This velocity updates the current weights every training step and is updated by the gradient and the previous velocities. A hyperparameter that is multiplied with the previous velocity determines how fast the contributions of previous gradients decay. It can be seen that updating the weights this way makes it harder for the gradient to drastically change the direction of the weights, which stops it from bouncing around or accelerate learning when the directions of the velocity and the negative gradient are equal.

The momentum algorithm with velocity $\mathbf{v}$ and velocity decay hyperparameter β can be seen

in equation 2-9:

$$\begin{aligned} \mathbf{v}_{n+1} &= \beta \mathbf{v}_n - \frac{\epsilon_n}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}_n, \mathbf{y}^{(i)}) \\ \mathbf{w}_{n+1} &= \mathbf{w}_n + \mathbf{v}_{n+1} \end{aligned} \tag{2-9}$$

Earlier in this section, it is stated that the learning rate is an important hyperparameter for training a neural network and that it can be chosen static or dynamic. RMSprop is a learning algorithm that uses an adaptive learning rate based on the history of the calculated gradients. The algorithm can be seen in equation 2-10, with $\mathbf{r}$ representing a moving average of the element-wise squared gradient and small constant δ to avoid an exploding learning rate through dividing by small numbers.

$$\begin{aligned} \mathbf{g} &= \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}_n, \mathbf{y}^{(i)}) \\ \mathbf{r}_{n+1} &= \rho \mathbf{r}_n + (1-\rho)\mathbf{g} \odot \mathbf{g} \\ \Delta \mathbf{w} &= -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}_{n+1}}} \odot \mathbf{g} \\ \mathbf{w}_{n+1} &= \mathbf{w}_n + \Delta \mathbf{w} \end{aligned} \tag{2-10}$$

It can be seen in the equation above that the RMSprop algorithm squares the gradient element-wise and adds it to a moving average of the gradients. How long the gradients influence the moving average is decided by a hyperparameter ρ. The squaring is done to only get positive number, such that the learning rate will stay positive and does not affect the sign of the gradients. Lastly, the weights are updated with the negative gradient times the learning rate divided by the square root of the moving average. This is all done element-wise, which means that the learning rate is different for each network weight.

ADAM [32], which stands for Adaptive Moments, is a combination between the RMSprop and momentum algorithm with a few small changes. The complete algorithm can be seen in equation 2-11.

$$\begin{aligned} \mathbf{g} &= \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}_n, \mathbf{y}^{(i)}) \\ \mathbf{s}_{n+1} &= \rho_1 \mathbf{s}_n + (1-\rho_1)\mathbf{g} \\ \mathbf{r}_{n+1} &= \rho_2 \mathbf{r}_n + (1-\rho_2)\mathbf{g} \odot \mathbf{g} \\ \hat{\mathbf{s}} &= \frac{\mathbf{s}_{n+1}}{1-\rho_1^{n+1}} \\ \hat{\mathbf{r}} &= \frac{\mathbf{r}_{n+1}}{1-\rho_1^{n+1}} \\ \Delta \mathbf{w} &= -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta} \\ \mathbf{w}_{n+1} &= \mathbf{w}_n + \Delta \mathbf{w} \end{aligned} \tag{2-11}$$

ADAM uses two helping variables: $\mathbf{r}$ for storing the moving average of the gradient, used in momentum and $\mathbf{s}$ for storing the moving average of the squared gradient, used for RMSprop.

Both of these variables are initialized at the origin, which makes them have a bias which is corrected by scaling the vectors down according to the used hyperparameter and the amount of training steps taken. After the bias correction, the weights are updated by the negative gradient with momentum applied and a gradient scaled by the RMSprop weight scaling.

A big advantage of the ADAM algorithm is that setting the hyperparameters to a default value suggested by the developer often leads to good training performances.

2-1-3 Introduction to Convolutional Neural Networks

Convolutional neural networks (often abbreviated to CNN) are neural networks that try to mimic biological vision processing through the layered structure of a neural network. This biological process was first described by Hubel and Wiesel [33] in 1959. They discovered that individual neurons in the visual cortex of a cat only respond to changes in small regions of the visual field and that some neurons fire rapidly when a line at one angle is presented and others react similar when a line with a different angle is presented. In 1968, they presented [34] a model for visual perception based on two different cells: simple cells (S-cells) that react the most when edges in a specific orientation are presented in their receptive field and complex cells (C-cells), which have a larger receptive field and whose output is independent from the location where the edges are presented.

The first algorithm that tried to use these cells is the neocognitron by Fukushima [35]. This algorithm uses layers containing multiple S-cells and C-cells in alternating fashion to recognize Japanese handwriting. The S-cells recognize parts of characters and the C-cells groups several of these S-cells together, making the recognition location invariant. Even more important than the location invariance is the use of multiple layers, with the input of the next layer being the output of the previous layer. The first layer will therefore recognize simple features, such as edges with a particular orientation or corners, the next layer combinations of these edges and corners, making more complex shapes. This concept of simple feature extraction in the first layers and combining these features to more and more complex ones with each next network layer can still be found back in modern convolutional neural networks.

In modern convolutional neural networks, the S-cells and C-cells have evolved in to more complex network layers, but the concept is pretty similar to them. The feature extraction, first done by the S-cells is now done by the convolution layer. This layer uses a set of convolutional filters and moves them across an input image or the the output of a previous layer, often called a feature map. At every position of the convolution filter, the values of the image/feature map and the values of the filter are multiplied element-wise and each product is summed up to a total value, which is located in the output feature map. After one operation of these convolution operations, the filters move to the next position, repeating the element-wise multiplication and addition. The amount of spaces the filters move is called the stride, which is a parameter that is decided during the design phase of the network. Common stride values are 1 or 2, but larger strides are possible. Figure 2-4 visualizes this convolution operation, such that it may give more clarity to the reader. In this figure, a bottom row and column on the left side containing zeros can be seen in the input map. This practice is called padding, where extra rows and columns of zeros allow the convolutional filters to cover the image more thoroughly.

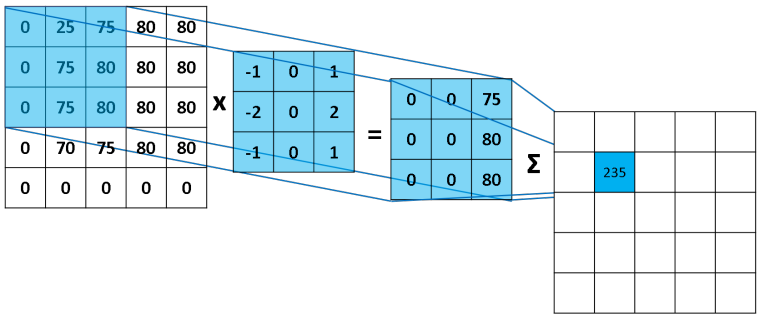

Figure 2-4: The convolution operation visualized [4]

While the convolutional layer also functions partially as the C-cells, since the filters that recognize features are applied all across the image, instead focusing on one spot, the actual combining of multiple cell outputs is now done by the pooling layer. This layer has a feature map as input and pools several of the feature map values together to one value, effectively decreasing the size of the map. The most frequent type of pooling used, is max pooling, where the maximum value is taken from multiple cells and put into the output map. Figure 2-5 gives an example of this max pooling practice. The amount of values that are pooled together and the type of pooling used, is up to the network designer.

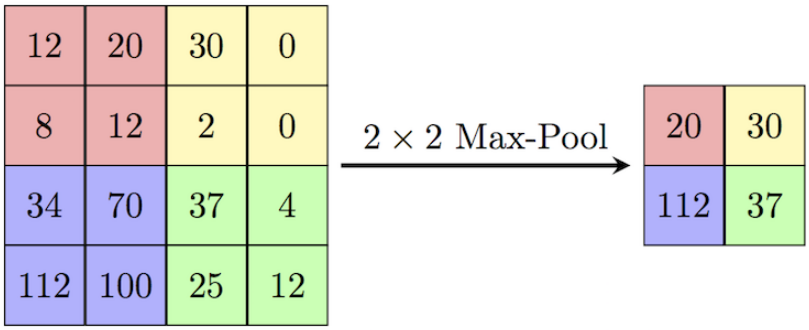

Figure 2-5: An example of max pooling [5]

Looking at both operations, the convolution and max pooling, it is clear that the feature maps decrease in width and height, but this decrease is compensated by an increase in depth. This increase is caused by the amount of convolutional filters used in each convolution layer. This phenomenon is seen in figure 2-6, where a full convolutional neural network is displayed.

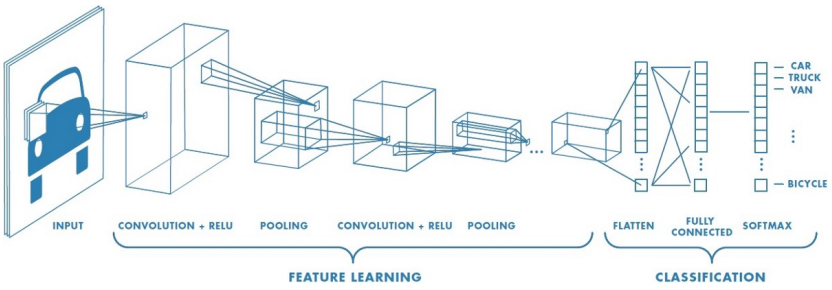

Figure 2-6: The complete structure of a convolutional neural network [6]

The last layers of a convolutional neural network are the fully connected layers. These layers use all the features that the convolutional and pooling layers learned and uses them to classify the input image into a specific class. These layers are similar to the ones of the neural networks described in section 2-1-1.

2-1-4 Influential Convolutional Neural Network Architectures

One of the first convolutional neural network was designed by Yann LeCun in 1989 [7] and was named LeNet-5 after its creator and the amount of convolution and pooling (then called subsampling) layers present in the network. The network was used to recognize digits in handwritten ZIP-codes and was the first of its kind to use the backpropagation to train the network parameters. The full network architecture can be seen in figure 2-7 and consists of 7 layers: C1 and C3 are convolution layers, S2 and S4 are subsampling layers, C5 and F6 is a fully connected layer and the output is a softmax layer that describes how confident the network is about the input representing a certain digit between 0 and 9.

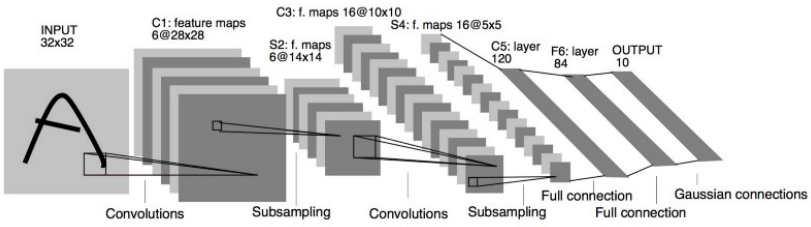

Figure 2-7: Structure of the LeNet-5 convolutional neural network [7]

After LeNet and its evolutions, convolutional neural networks were developed, but remained more on the background while other methods for image recognition gained more traction, such as Histogram of Orientated Gradients (HOG) [36] and Scale Invariant Feature Transforms (SIFT) [37]. This was because of the long training times needed for CNN's and the other methods did not have this problem or were less affected by it. The network that put CNN's back in the spotlight was AlexNet [38]. Created in 2012 for the ImageNet [39] image recognition challenge, it used the GPU for computation, which sped up the process of training significantly. The result was that AlexNet outperformed every other submission with an error rate that was 8% lower than the second best submission. The architecture of AlexNet, which

can be seen in figure 2-8 is in essence similar to LeNet: the first part of the network is alternating between convolutional layers and pooling layers that extract features from images, the second part is a series of fully connected layers, using the features to perform object recognition and the output layer gives a confidence level per object class. The difference between LeNet-5 and AlexNet is that AlexNet is significantly larger in size, which was made possible by training the network on the GPU.

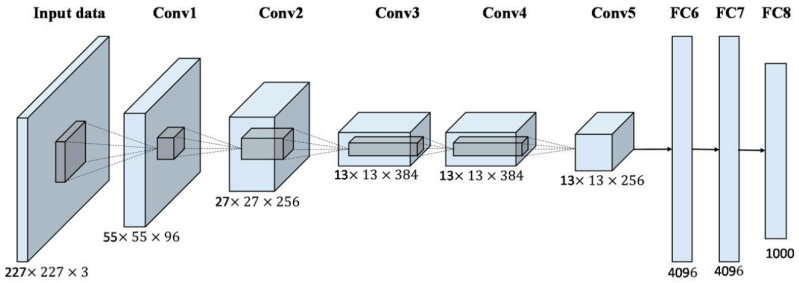

Figure 2-8: Structure of the AlexNet network [8]

After AlexNet showed that convolutional neural networks was a promising way for recognizing objects in images, more and more research was done to develop new network architectures that could outperform AlexNet on recognition precision. VGG [40], which stands for Visual Geometry Group, is a convolutional neural network that uses a fixed convolutional filter size of 3-by-3. This small filter size is compromised by increasing the amount of filters per layer and doubling the amount of convolutional and pooling layers compared to AlexNet, which is visualized in figure 2-9. VGG outperformed AlexNet by a large margin: the authors of the original paper report a error percentage decrease of more than 10%.

The problem with the VGG networks is that the amount of trainable weights in the network almost doubled compared to AlexNet: VGG16 has approximately 138 million and the deeper VGG19 even more, with most of them (102 million) coming from mapping the final feature map to the fully connected layers. The amount of weights causes the VGG networks to have long training times, which is undesirable. Two popular network architectures that try to limit the amount of weights are GoogleNet [10] and ResNet [41].

GoogleNet uses inception modules, which are blocks of convolutional filters with different sizes that have the same input image. The output of these filters are then concatenated to form the input of the next inception module. This causes the network to perform several convolution operations in one layer, effectively putting multiple convolutional layers in one. The use of inception modules also reduces the layer depth of the network significantly, since the input and output of the module must be equal to each other. This reduces the computational time significantly, especially compared to networks with high filter depths in later stages. Figure 2-10 shows an inception module used in GoogleNet. The yellow blocks with 1x1 convolutions are reducing the depth of the layer before the 3x3 and 5x5 convolutions, reducing the computational stress even further. The final network consists of 22 layers, excluding some preliminary convolutions and took the top spot of the ImageNet classification challenge that year with an top-5 error percentage of only 6.67%.

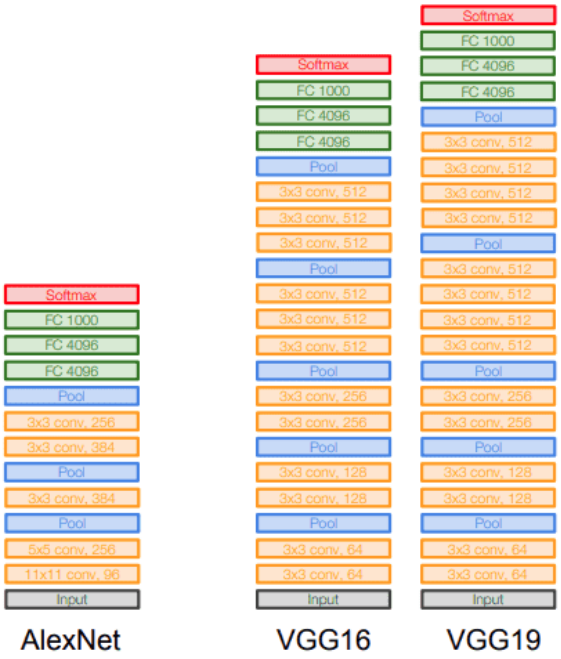

Figure 2-9: Comparison between AlexNet and the VGG networks [9]

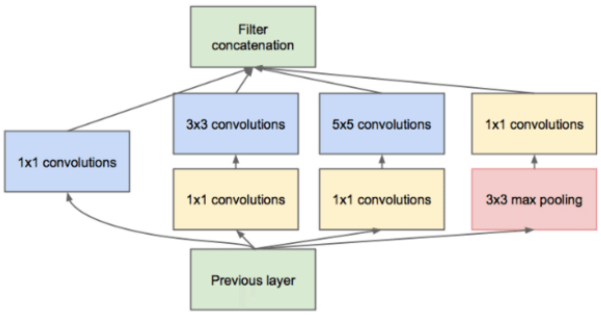

Figure 2-10: Inception module of the GoogleNet network [10]

Another problem with deep neural networks that leads to long training times is the problem of vanishing gradients. When using backpropagation to upgrade the weights, some partial derivatives approach zero. This is due to the multiplication of small partial derivatives with small partial derivatives, respecting the chain rule. The result is that especially the early layers of deep neural networks take a long time to update their weights. ResNet tries to solve this problem of vanishing gradients by introducing shortcuts between network layers. These shortcuts are visualized in figure 2-11, where a relatively shallow ResNet-12 is visualized.

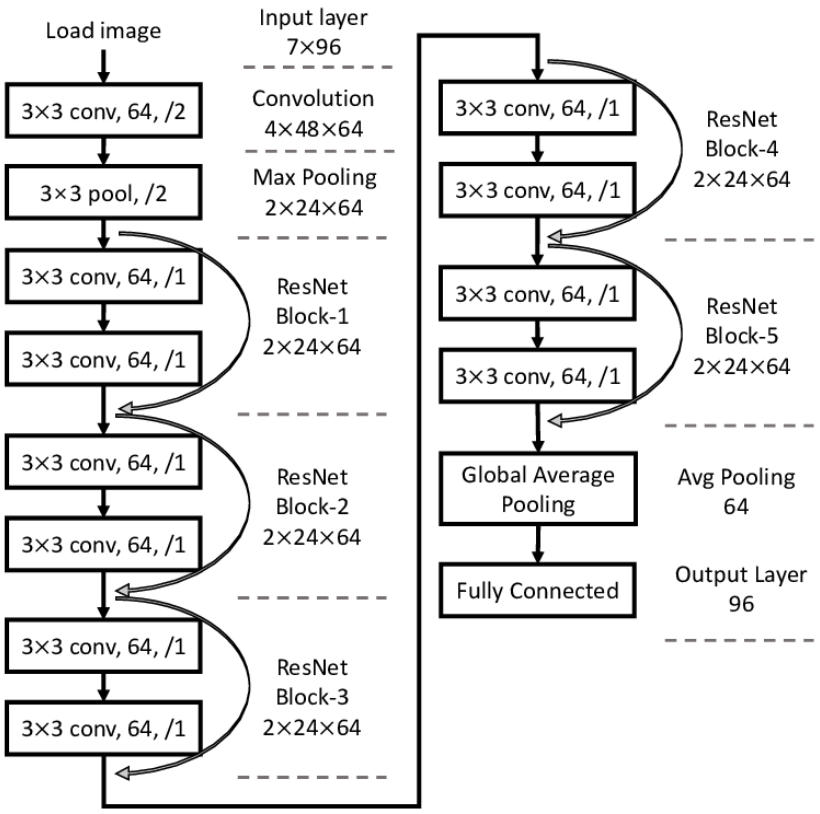

Figure 2-11: ResNet-12 architecture [11]

Using the shortcuts, the problem of the vanishing gradient is virtually eliminated, which enables ResNet to reach an extreme amount of layers: the original paper used a network architecture of 152 layers with 60 million parameters, similar to the original AlexNet.

2-1-5 Convolutional Neural Networks for Object Detection

The convolutional neural networks discussed up until now can be used for object recognition, where the network only returns a certain amount of object classes (e.g. dog, car, house) and the probability that the object in the image is indeed that class. Object detection uses the object recognition, but also needs to give a location of the object in the form of a rectangular box containing the object, often called a bounding box. This section showcases the most popular object detection algorithms using convolutional neural networks.

R-CNN and Evolutions

R-CNN [12], which stands for region based convolutional neural networks, is an object detection algorithm that was designed in 2014 to replace the HOG [36] and SIFT [37] methods that were still the state-of-the-art methods to detect objects and their performance was lacking for that time. It was the first algorithm that bridged the gap between object recognition with convolutional neural networks and object detection, but this came with two problems the

authors needed to overcome: localize the objects and train the complete model with a small amount of data. The latter is because annotated data for object detection, complete with bounding boxes was scarce during that time, while object recognition data was more widely available.

The first problem was solved by the author by introducing an independent region proposal network that generates a large amount, over 2000 per image, of region proposals. These region proposals are projected on the image to create a sub-image, which is warped to accommodate the input of the convolutional network. After object recognition is performed on the warped sub-image, a decision is made: if the network did not recognize any object, the region is rejected and if an object is found and the confidence of the network is high enough, the region and the object containing it is accepted. To further reduce the amount of proposals, greedy non-maximum suppression is used: a region is rejected when it overlaps enough with another region with a higher score. The architecture of R-CNN is visualized in figure 2-12

In the R-CNN paper, selective search [42] was used as an algorithm to create region proposals and AlexNet [38] pre-trained on the ImageNet [39] dataset was used as object recognition network. The pre-training directly solves the second problem of not having enough training data for object detection. The complete network was trained on the PASCAL VOC [43] dataset, one specific for object detection with a new classification output layer. Stochastic gradient descent was used with mini-batches consisting of 32 positive region proposals and 96 negative region proposals. Proposals are positive when the intersection-over-union between the proposal and the ground-truth bounding box is more than 0.5 and the rest are negative proposals. Intersection-over-union between two bounding boxes is defined as follows: suppose there are two bounding boxes B_1 and B_2, the intersection over union (IoU) is defined by equation 2-12

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2} \tag{2-12}$$

Testing on the PASCAL VOC 2010-2012 dataset gives a mean average precision (mAP) of 53.7%, which was 13% higher than the competition of that time. This makes R-CNN a fundamental development in the field of object detection. The mAP metric will be explained more in detail in section 2-3

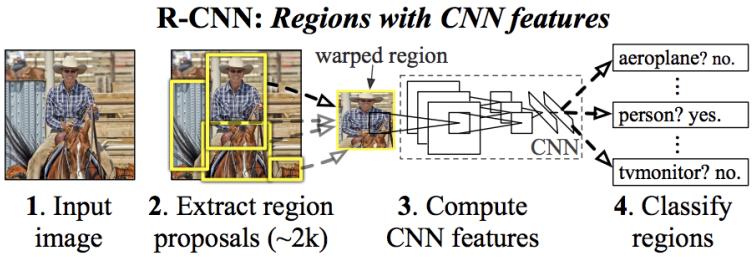

Figure 2-12: Structure of the R-CNN algorithm [12]

After the publication of R-CNN, there were a few successful attempt in improving the object detection algorithm. SPPNet [13] tried to counteract the data loss induced by cropping and warping the input image by pooling the output of the last convolution layer into separate

spatial bins and concatenate those bins into a vector of fixed size (see Figure 2-13, therefore removing the need for cropping and warping. The biggest improvement was in the speed, as SPPNet maps the region proposals onto the last feature map, therefore having to only calculate it once, instead of calculating it for every region proposal.

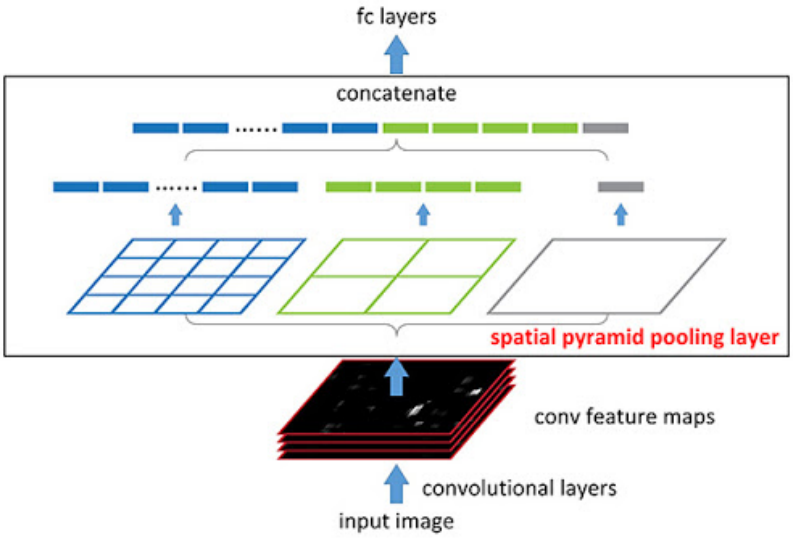

Figure 2-13: The novel Spatial Pyramid Pooling layer introduced in the paper [13]

Fast R-CNN [14] was another improvement to the R-CNN algorithm that proposed a new training process for deep-learning object detection algorithms that uses one training stage instead of multiple stages, which makes it significantly faster in training than the previous algorithms R-CNN and SPPNet. It also introduced a Region of Interest pooling layer (see Figure 2-14) that is comparable to the SPP layer: it extracts features from the output of the last convolutional layer and pools these together to form a fixed-size vector for the fully connected layer. In the end, Fast R-CNN could be trained 9 times faster than normal R-CNN and was 146 times faster at testing time.

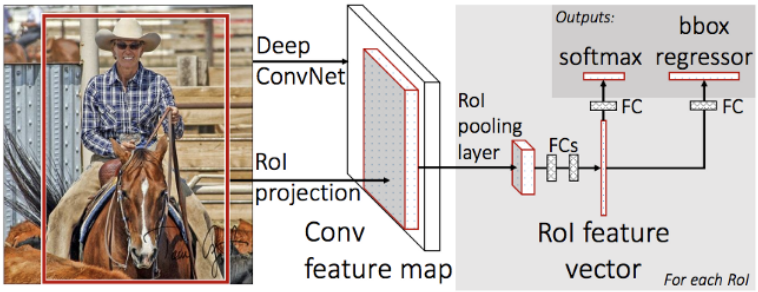

Figure 2-14: Structure of the Fast R-CNN algorithm [14]

Faster R-CNN [15] (see figure 2-15 for the structure) proposed a novel Region Proposal

Network (RPN) that shares the convolutional layers with the detection network and computes the regions based on the output of the convolutional layers, thus removing the need for a separate region proposal algorithm. It is constructed by sliding a small network over the feature map output of the convolutional layers. At each position of the sliding window, a maximum of k bounding boxes is predicted and parameterized relative to k default boxes called anchors. These anchors are centered in the sliding window and represent a certain size and aspect ratio. These anchors are invariant to translation of the objects, which means that when the object is translated, only the proposal is translated. This translation invariance helps reducing the amount of parameters needed for the RPN network significantly. Another advantage of the anchors is that Faster R-CNN can handle images with objects, each with different aspect ratios and sizes. The final output of the RPN network is two times k scores that gives the probability of a region proposal containing an object or no object and four times k bounding box coordinate relative to an anchor box. The full RPN layer is also visualized in figure 2-16.

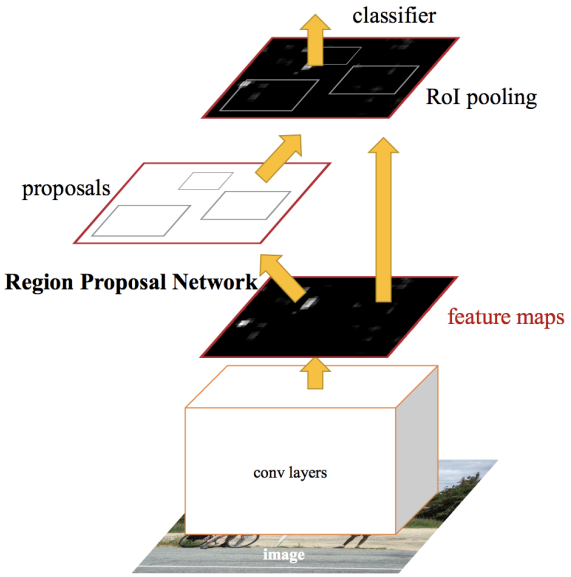

Figure 2-15: The Faster R-CNN algorithm visualized [15]

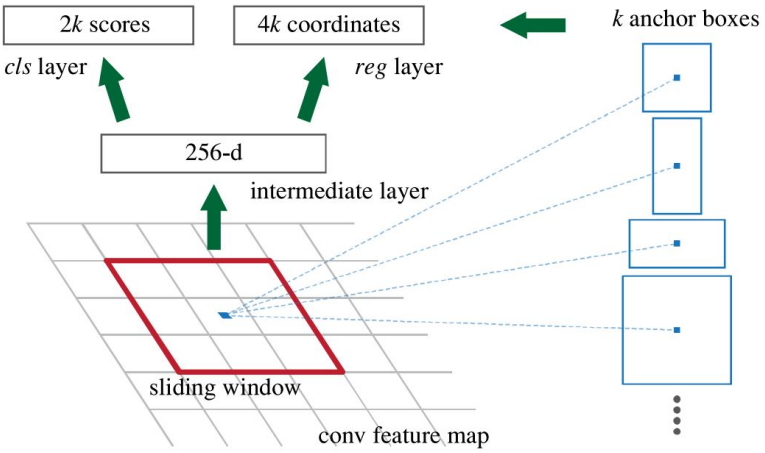

Figure 2-16: Explanation of the RPN [15]

The RPN layer caused Faster R-CNN to severly outclass fast R-CNN: it has better detection precision and has an image processing time of only 59 ms, which corresponds to 17 fps, whereas Fast R-CNN takes 1830 ms to process one image.

2-1-6 Single-Shot Detectors

All R-CNN object detection methods are using a separate region proposal algorithm. These region proposal can be integrated into the algorithm by training both together, such as faster R-CNN, but the region proposal method is still a separate algorithm. The next two object detection methods are so-called single-shot detectors: they do not require the use of a separate module to calculate regions, but locate objects directly from the input image.

YOLO

YOLO [16], which stands for "You Only Look Once" is a single-shot object detector that incorporates the complete object detection algorithm in one neural network instead of modifying object classifiers to perform object detection. It does this by dividing the input image into a grid with a specified size $S \times S$. ($S = 7$ in the paper). Each grid cell then predicts B bounding boxes ($B = 2$ in the paper), each of the boxes with 5 parameters: x, y, w, h, the coordinates of the bounding box relative to the grid cell and the confidence score of the bounding box that tells how confident the model is about the bounding box that is predicted. If the center of an object is located in one of the grid cell, that specific grid cell is then responsible for detecting the object. In addition to the bounding boxes, each grid cell also predicts C conditional class probabilities that indicate how probable it is for an object to be of a specific class, given that there is an object located in the grid cell. Both these outputs are also visualized in the paper by figure 2-17

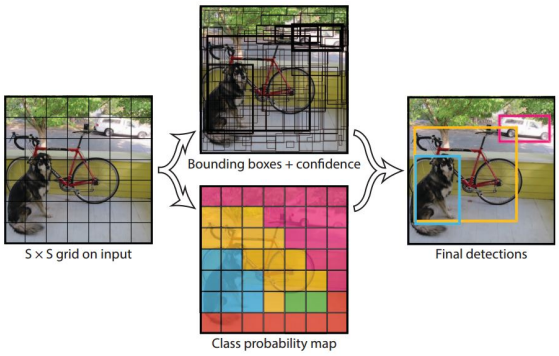

Figure 2-17: Visualization of the bounding boxes and class probabilities used in the YOLO algorithm [16]

The network used for the YOLO algorithm can be seen in figure 2-18. It consists of 24 convolutional layers, 2 fully connected layers and is inspired by the GoogleNet's [10] inception architecture. However, instead of using inception modules to limit the depth of each layer, YOLO uses 1x1 convolution operations to reduce the layer depth before the actual convolution layers. The output is a tensor with a size of $S \times S \times (5 * B + C)$, where all the bounding boxes and class probabilities are encoded into.

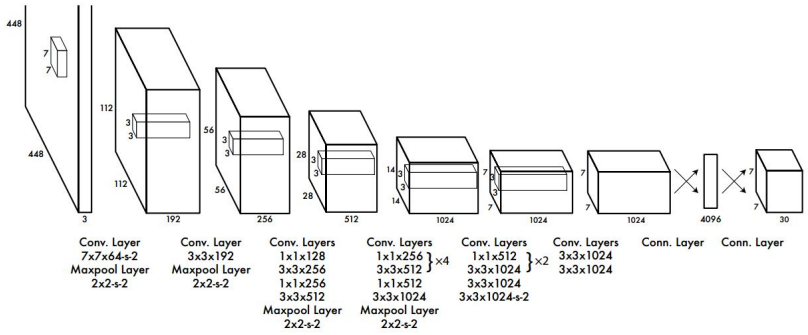

Figure 2-18: Network architecture of the YOLO algorithm [16]

Training of the YOLO network is done by first pre-training the first 20 convolutional layers on the ImageNet [39] competition dataset. After having good enough accuracy on image classification, the rest of the convolutional layers and fully connected layers are added with random initialisation weights.

After training on the PASCAL VOC [43] datasets, YOLO actually performs worse than faster R-CNN with a mean average precision (mAP) of 63.4 %, while faster R-CNN has a mAP of 73.2%. However, the fundamental difference between YOLO and faster R-CNN is not the performance, but the detection speed. Since YOLO only processes the image once, it can detect objects in real-time with a speed of 45 fps (45 images per second). The even faster Fast YOLO, which has only 9 convolutional layers, can detect objects with a speed of 155 images per second, albeit with sub-optimal accuracy. This detection speed made YOLO a state-of-the-art object detector at the time of its creation.

SSD

SSD [17], which stands for Single Shot multibox Detector, is a single-shot object detection algorithm, very similar to YOLO, with two key differences.

SSD divides the image into a grid, but instead of predicting the bounding boxes, it uses a fixed set of bounding boxes, similar to the anchors seen in the RPN of faster R-CNN. These default boxes are then applied to feature maps of varying sizes, enabling SSD to detect objects at different scales, where smaller feature maps account for larger parts of the image and therefore larger default bounding boxes. The full structure of the SSD network can be seen in figure 2-19. When compared to the structure of YOLO, which can be seen in the same figure, there is a noticable lack of fully connected layers in the SSD network, and the amount of predictions from SSD is significantly larger compared to the amount of predictions from YOLO (8732 compared to 98).

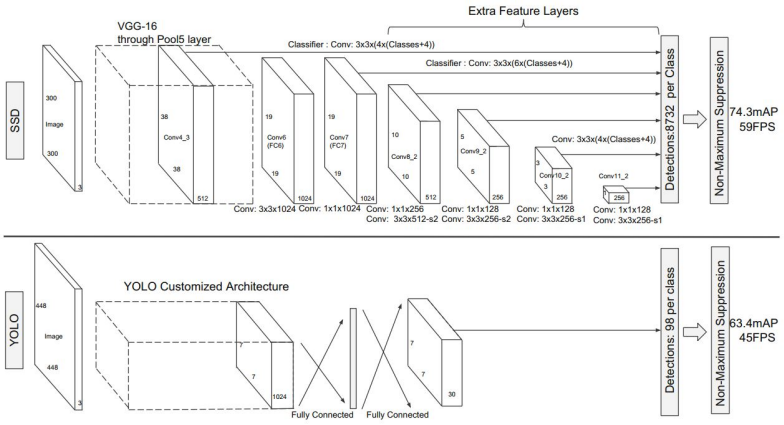

Figure 2-19: Network structures of YOLO and SSD compared [17]

Similar to YOLO, the final predictions consist of 4 coordinates that represent the offset to a default bounding box and a confidence score per object class. To reduce the amount of predictions, non-maximum suppression is used, similar to the method used in R-CNN.

For comparing the SSD algorithm to YOLO, a VGG16 pre-trained on the ImageNet dataset is used for the first convolutional layers and the two fully connected layers are converted to convolutional layers. The additional convolutional layers are then added and the whole network was trained on several object detection datasets, including the PASCAL VOC [43] sets.

In the test results, two different input sizes are used while training, 300 by 300 pixels and 512 by 512 pixels. The results show that SSD300 and SSD512 have both comparable accuracy (around 75% mAP), both outperforming faster R-CNN and YOLO. In terms of speed, SSD300 outperforms YOLO with a detection speed of 59 fps, compared to 45 fps for YOLO. Even SSD512 reaches a speed of 22 fps, which is almost real-time detection. This increase in accuracy and detection speed makes SSD the most interesting object detection algorithm so far discussed.

The Faster R-CNN, YOLO and SSD algorithms have principles that can be found back in the algorithm used for object detection in this thesis project: YOLOv3, which will be discussed in detail in section 3-1.

2-2 KITTI Dataset

The KITTI Dataset [44] [18] is a pioneering autonomous driving dataset created by the Karlsruhe Institute of Technology in 2012 with data sets for evaluating optical flow, stereo vision and visual odometry. Since then, many other benchmarks were added, such as object detection, object tracking and semantic segmentation. Updates to the stereo vision and optical flow tasks were added in 2015, increasing the size of the dataset even more.

The platform on which the data for the KITTI dataset is recorded is a standard road car equipped with four cameras: two that capture video in grayscale and the other two are capable of capturing color. Both the grayscale and color camera are setup in such a way that stereo information is possible from camera images alone. In addition to the four cameras, a LiDAR sensor is mounted on top of the car and a GPS/IMU unit is fitted in the back. The LiDAR sensor captures point clouds with rate of 10 Hz. The cameras are triggered by the LiDAR sensor, synchronizing the camera images and pointclouds, such that both data types can be used together.

The KITTI dataset contains a total of 17 tasks and corresponding benchmarks, which makes the data in the dataset relevant to a wide spectrum of research. However, this unprecedented variety of tasks in the dataset does not translate to the size of the dataset. Most of the tasks are accompanied with a data subset of 400 images (200 for training, 200 for testing), a fraction of the size of modern datasets. Even the larger subsets, such as 2D and 3D object detection, have around 15000 images and matching LiDAR pointclouds, which is significantly smaller than datasets that are discussed later in this chapter.

Of the 17 tasks present in the dataset, only a few of them are relevant to the subject of velocity estimation. The tasks of 2D and 3D object detection and tracking are the most prominent ones. The semantic segmentation task could be used for object detection by dividing a camera image into different areas, each with its own label and derive the position of individual vehicles that way. However, this requires a different neural network architecture than already discussed in previous sections. Also semantic segmentation is focused on generating context for a vehicle and this is not necessary for object detection and/or tracking, since the general context of the scenario is already known a priori.

When the dataset is used for training a convolutional neural network, the annotations are as important as the sensor data. The tasks of object detection come with annotations in the form of 2D and 3D bounding boxes. These boxes indicate the height, width, depth (when 3D boxes are used) and position of an object. Accompanied with this spatial information of an object is information on which class the object belongs to (for example: car, van, bicycle, etc.) and if the object is partially occluded or not. Figure 2-20 gives a visualisation of the annotations present for the object detection task. All of these labelled objects also come with a unique identification number, used for the object tracking task.

Overall, the KITTI dataset is a small, but diverse dataset. This makes it suitable for usage in initial experiments or validating the performance of algorithms, since using a larger dataset

results in much larger training and testing times. However, most modern convolutional neural networks designed for object detection require a lot more training data then the KITTI set offers to give an adequate performance. This can be circumvented by using an existing CNN model that is pre-trained on a larger dataset and apply transfer learning with the training data from the KITTI dataset. However, transfer learning can have a negative effect on the performance of the neural network or cause overfitting. This is why the KITTI dataset will be used as validation dataset for the object tracking algorithm, as can be seen in section 4-2.

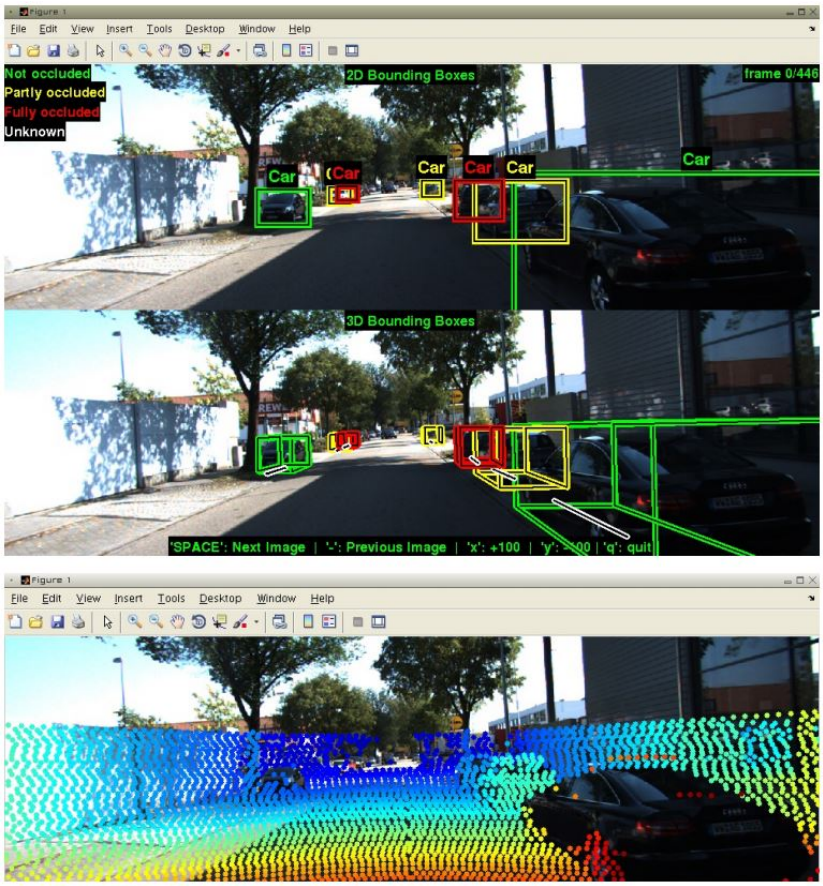

Figure 2-20: Visualisation of the annotations accompanied with the 2D (see top figure) and 3D (see middle figure) object detection tasks. The bottom figure depicts a visualisation of a LiDAR point cloud on a corresponding camera image. [18]

2-3 Berkely Deep Drive 100K Datasets

The Berkeley Deep Drive dataset [19] was created in 2018 at UC Berkeley to overcome the limits that existing autonomous driving datasets were experiencing, such as the lack of scene variation and annotation richness. Therefore, the aim of this dataset is to construct a large-scale visual dataset consisting of diverse driving scenes and multiple tasks that facilitate studies on autonomous driving.

The result is a dataset that consists of a hundred thousand high resolution, high frame rate

videos, including GPS and IMU sensor readings to facilitate the usage of vehicle trajectories in research projects. All of this data is collected through crowd-sourcing, therefore making the video images more diverse then if it was captured on one specific platform and drastically reducing the time needed to capture such a large amount of data.

In total, the Berkeley set contains ten distinct tasks that can be used for training neural networks: image tagging, lane detection, drivable area segmentation, multi-object detection tracking, multi-object segmentation tracking, domain adaptation and imitation learning. A visualistion of these tasks and matching annotations can be seen in figure 2-21. Of these tasks, only one is considered relevant for the velocity estimation algorithm: the multi-object detection tracking.

This task is split into two benchmarks: one for object detection and one for multiple object tracking. Object detection and other image tasks uses the frame on fifth second for annotations and object tracking uses the whole video. The annotations for the object detection come in the form of 2D bounding boxes. These bounding boxes are paired with a class label that separates the objects into ten classes and information on if the object is occluded (partially covered by other objects) or truncated (object is only partially captured on the image). The benchmark used for object detection task is an Average Precision score (AP), a commonly used evaluation metric for object detection algorithms. This score is calculated by taking the integral of the precision-recall curve. The formula for the AP score can be found in equation 2-13

$$AP = 100 \int_0^1 \max\{p(r')|r' \geq r\}dr \tag{2-13}$$

where $p(r)$ is the Precision/Recall curve. This curve is constructed by calculating the precision and recall rates for an object detector at different class score confidence thresholds. The formulas for calculating the precision and recall ratios can be found in equation 2-14

$$P = \frac{TP}{TP + FP}, \qquad R = \frac{TP}{TP + FN} \tag{2-14}$$

Where TP is the amount of true positive detections, FP the amout of false positive detections and FN the amount of false Negatives at a certain confidence threshold. More intuitively, the precision ratio shows how likely it is that a positive detection is indeed positive and the recall ratio shows how conservative the detector is with assigning an object to a class. The mean average precision (mAP), as seen before in section 2-1, is the mean of all the average precision values for a specific class.

The multi-object tracking uses a selection of 2000 videos from the total set. These videos are annotated at 5 fps, resulting in 400000 video frames that can be used for training, validating and testing an object tracking algorithm. The annotations used for object tracking are the same as the ones used for the object detection task but with one addition: 2D bounding boxes with information about the class of the object, the identity of the object and if the object is occluded or truncated. The identity ensures that objects are seen as one entity detected over multiple frames and not as a series of independent, similar-looking entities. The Benchmarks used for the multi-object tracking task are MOTP [45], which stands for Multi Object Tracking Precision and MOTA [45], which stands for Multi Object Tracking Accuracy. The MOTA

score indicates how many times the detector tracked an object correctly in relation to the amount of ground truths present. The MOTP score look at how much the bounding boxes given by the detector coincide with the ground-truth bounding boxes of the set.

Overall, the Berkeley Deep Drive dataset is a large and diverse dataset with a good selection of tasks, ensuring that it can be used for this thesis project as an excellent way of training an object detection neural network.

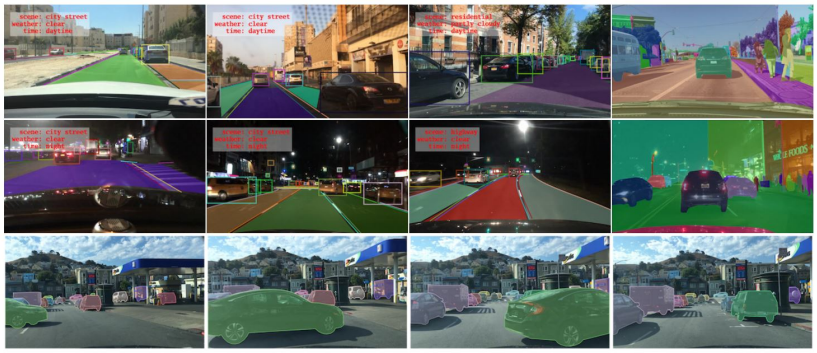

Figure 2-21: A visualisation of the Berkeley Deep Drive dataset and its annotations [19]

Chapter 3

Algorithm Implementation

Up until now, this thesis report went into detail about the background research done to gain knowledge about the inner workings of convolutional neural networks, as most state-of-the-art object detection algorithms use this technology. However, the main part of the thesis is designing a velocity estimation algorithm and explain the design choices and show quantative results, so that the thesis research question can be answered in the end. This chapter covers the design of the three main parts (object detection, object tracking and velocity estimation), explain their inner workings and why they are chosen.

Section 3-1 uses the knowledge gained in the preliminary research to select an object detection algorithm based on a convolutional neural network, which will be suitable for a velocity estimation algorithm. Building further upon this object detection algorithm, Section 3-2 showcases an object tracking algorithm that uses bounding box object detections to track objects. Lastly, Section 3-3 shows two methods, where one of them is novel, to transform tracked objects into velocities, therefore answering a part of the thesis' research question.

3-1 Object Detection

For object detection, an evolution of the YOLO single-shot-detector is chosen: YOLOv3. The SSD algorithm found in section 2-1-6 was also considered, however it seemed that at the time of selecting an object detection algorithm for this thesis project, the SSD algorithm did not receive any upgrades, while the YOLO algorithm did multiple times. The YOLOv3 algorithm is implemented in PyTorch, one of the most used software packages for artificial neural networks. This implementation is then retrained on the object detection part of the Berkeley Deep Drive 100K, already seen in Section 2-3.

3-1-1 YOLOv3 architecture

YOLOv3 [20] is an evolution of the YOLO single-shot object detection algorithm, already seen in section 2-1-6. This means that it has inherited several traits from the original YOLO algorithm, but also a significant amount of improvements. The main difference is that YOLOv3

has a "backbone" consisting of a DarkNet-53 (see Figure 3-1) object classification neural network. DarkNet-53 is a residual convolutional neural network, similar to ResNet in section 2-1-4 and an extended version of the DarkNet-19 classification network used in YOLOv2 [46]. DarkNet-53 is first trained on an image classification network, then the last layer is stripped from the network while the convolutional layers are not altered. The first main improvement is that YOLOv3 extracts the objects at three scales, as YOLO struggled with detecting small objects accurately. This is done by first adding a few convolutional layers to the DarkNet backbone, giving it the ability to detect objects with the final backbone feature map. Then, the feature map of a few layers prior is taken and upsampled, increasing the size. This upsampled layer is then concatenated with a feature map even further back in the network. This concatenated map is then used to detect objects at a smaller scale, because of the increased resolution. This practice is repeated with the second detection stage, making YOLOv3 extracting objects at three different scales. The full architecture, including the different scale detections can be seen in Figure 2-18.

At each of the three outputs of the YOLOv3 algorithm, a similar structure as the original YOLO can be found: an $S \times S$ grid with every cell representing a bounding box location with a specified size. Each grid cell contains a vector with B predictions, each prediction containing four offsets to the default bounding boxes, an objectness score, which tells how confident YOLOv3 is that there is an object in that cell and C class scores, which shows the probability of the object inside the bounding box belonging to a certain class. In the end, the output will be a tensor of the size $S \times S \times (B \times (5 + C))$. The difference between YOLO and YOLOv3 here is that the grid positions do not represent an equally divided grid, which is the case with YOLO, but each cell corresponds to an anchor box from which the offset is calculated. These anchor boxes were used in the Faster R-CNN object detection algorithm and the SSD algorithm, which are discussed previously in this report in section 2-1-5 and section 2-1-6. The anchor boxes were hand picked for Faster R-CNN, however for YOLOv3, they are selected using k-means clustering: using the bounding boxes of the set, k anchor boxes are attempted to be found such that these represent all the bounding boxes as accurate as possible, without making the model too complex, which increases training time. The developers of the YOLOv2 algorithm chose to use 5 anchor boxes, which was also adopted by YOLOv3. A problem that arises when using anchor boxes, is that the neural network model becomes unstable, mostly during early stages of training. This is solved by transforming the coordinates in a different way than seen in YOLOv1. This new transformation can be seen in Figure 3-3. The blue rectangle represents the bounding box of the object, whereas the black dotted rectangle represents one of the anchor boxes. The center of the bounding box is transformed to an offset from specified grid positions c_x and c_y. The position outputs from the neural network t_x and t_y are put through a sigmoid function, limiting them between minus one and one and are then added to the grid positions to get the absolute center coordinates. The width and height outputs from the YOLOv3 network t_w and t_h are exponented and multiplied with the width and height of one of the anchor boxes to get the true width and height.

3-1-2 PyTorch Implementation of YOLOv3

Instead of building the YOLOv3 network from scratch, an implementation of YOLOv3 in PyTorch is used for object detection. The main argument of using PyTorch for neural networks,

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 3-1: Table of the DarkNet-53 architecture [20]

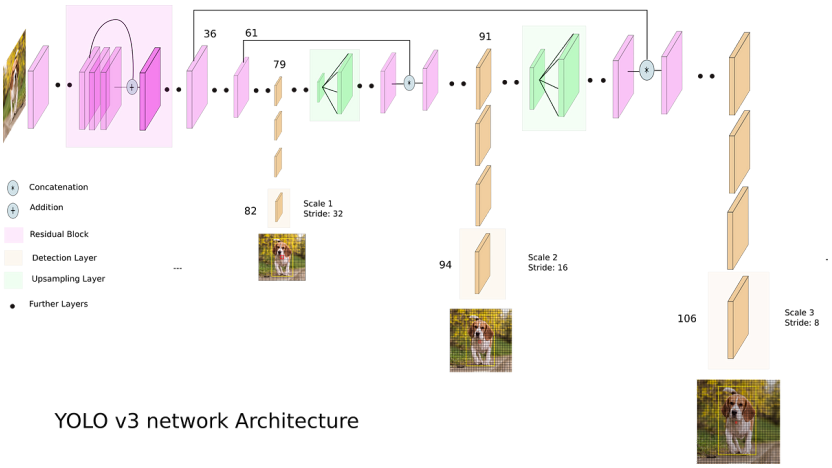

Figure 3-2: Architecture of the YOLOv3 network [21]

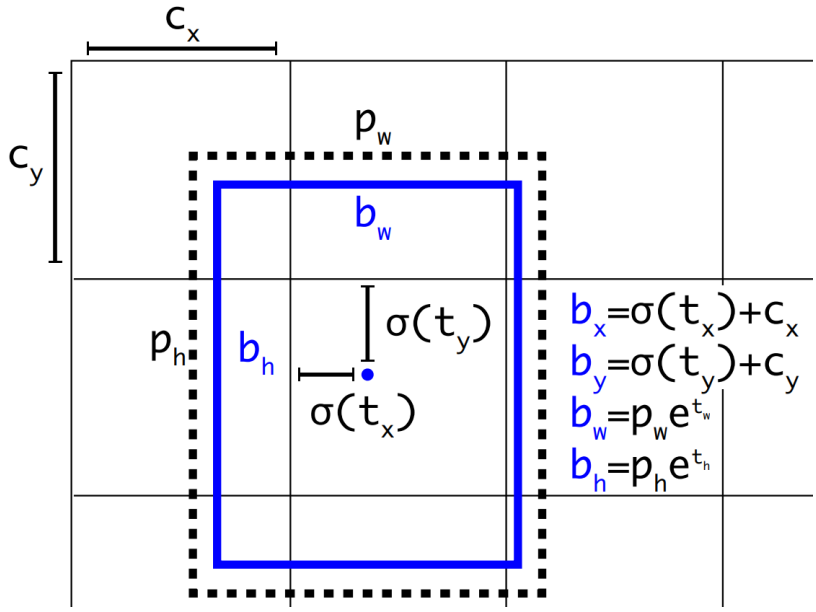

Figure 3-3: Illustration of the YOLOv3 bounding box configuration [20]

is that it is a widely used Python library for building and training neural networks, which makes documentation about it widely available.

The implementation of the YOLOv3 object detection network is done by the company Ultralytics and the complete code can be found on their GitHub page [47]. Instead of using the pre trained weights, which also included classes that are not relevant, the Berkely Deep Drive 100K dataset is used. Before the Berkely dataset could be used for training, the ground truth bounding boxes in the dataset had to be modified, as the Berkely dataset gives the pixel coordinates of the top left and bottom right corner of the bounding box. The Ultralytics YOLOv3 implementation however requires the bounding boxes to consist of the center coordinate, width and height of the bounding box. In addition to this transformation, the bounding box coordinates have to be normalized between zero and one. These two transformations can be combined into one simple bounding box conversion algorithm, found in Equation 3-1.

$$\begin{aligned} c_x &= \frac{x_{tl} + x_{br}}{2w_I} \\ c_y &= \frac{y_{tl} + y_{br}}{2h_I} \\ w &= \frac{x_{br} - x_{tl}}{w_I} \\ h &= \frac{y_{br} - y_{tl}}{h_I} \end{aligned} \tag{3-1}$$

Here c_x, c_y, w and h are the center coordinates, width and height of the bounding box used by the YOLOv3 implementation. x_{tl}, x_{br}, y_{tl}, y_{br} are the coordinates of the ground truth bounding boxes supplied by the Berkely Deep Drive 100K dataset and w_I and h_I are the width and height of the image.

3-2 Object Tracking with SORT

The object detection algorithm used for this project is on its own not capable of estimating velocities of vehicles, as YOLOv3 only provides a list of objects with bounding boxes and class confidence scores per camera frame. If a certain object is in the field-of-view of the camera for multiple frames, YOLOv3 will not recognize that the detections are linked to the same real-world object. This is the task of the object tracking algorithm, which combines detections of multiple frames into sequences of detections, each sequence ideally representing an unique, real-life object. Such an algorithm must first of all be accurate, but also fast at the same time, to keep the complete velocity estimation algorithm be able to estimate velocities in real-time. The tracking algorithm used is SORT, Simple Online and Real-time Tracking [48], which uses the well-known Systems & Control concept of a Kalman Filter to predict the next position of tracked objects and tries to match these predictions with the YOLOv3 detections using the Hungarian algorithm [49] to match new detections to already tracked objects. This section will explain the SORT algorithm in more detail, with Section 3-2-1 showcasing the global structure, Section 3-2-2 the bounding box location predictor using a Kalman filter and Section 3-2-3 will go into detail about matching the predictions to the detections.

3-2-1 Algorithm Structure

The complete structure of the SORT algorithm can be found in Figure 3-4.

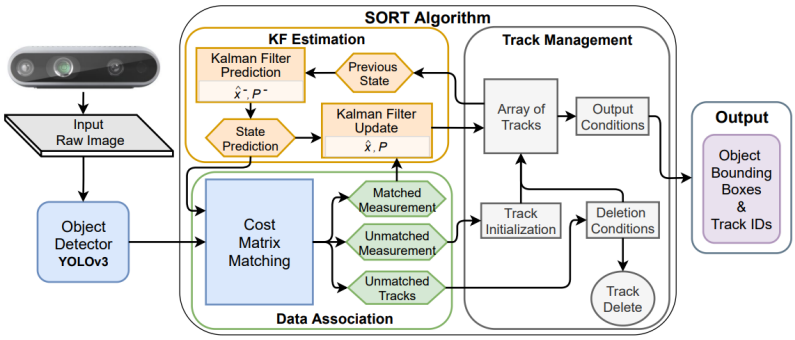

Figure 3-4: Structure of the SORT Algorithm [22]

Each tracked object is represented in SORT with a unique object ID number, starting from one and increasing with each new object. Accompanying this ID are a model describing the movement of the bounding box and an accompanying kalman filter, object age, and time since last matched detection. The first step of the SORT algorithm is that the kalman filter of each existing tracked object predicts the next location of the bounding box. Then all the detections created by YOLOv3 are matched to the predictions by creating a cost matrix and using the Hungarian algorithm [49] to ensure the matching is optimal and time-efficient. After this matching step, there are three possible outcomes for each detection and existing track. The first option is that a detection is matched to an already existing track. This matched detection is then used to update the kalman filter of the track, making the next prediction more accurate. If the time since last detection is non-zero, this will be set to zero as well. The second option is that a detection could not be matched to an existing track. In that case, a new track is initialized with a unique track ID, with the detection serving as the initial estimate. The third option is that an existing track will not receive a new detection that matches the prediction. In that case, the time since last matched detection will increase with one. If this time reaches a certain threshold, the track will be deleted from the list of existing tracked objects. The final output of the SORT algorithm is a list of bounding boxes with their corresponding ID numbers.

3-2-2 Velocity Model and Kalman Filter

As stated before, the position of each tracked object is predicted in advance using a kalman filter. This filter uses measurements over a time period and uncertainty estimations to predict the state of the system as accurate as possible. The model used to describe the movement of bounding boxes across camera images can be described with the following two equations:

$$\mathbf{x}[k+1] = A\mathbf{x}[k] + \mathbf{w}[k],\ \mathbf{y}[k] = C\mathbf{x}[k] + \mathbf{v}[k] \tag{3-2}$$

$$\mathbf{w}[k] \sim \mathcal{N}(\mathbf{0}, Q),\ \mathbf{v}[k] \sim \mathcal{N}(\mathbf{0}, R) \tag{3-3}$$

Where $\mathbf{x}[k]$ is the state vector of the tracked object at the k-th frame from the first detection, $\mathbf{z}[k]$ the measurement vector at the k-th frame. The state transition matrix is denoted by A and the observation model matrix as C. The process noise $\mathbf{w}$ and measurement noise $\mathbf{v}$ are modelled as zero-mean Gaussian noise with the matrices Q and R representing their respective covariance.

The kalman filter algoirthm begins with a prediction of the state vector given the previous prediction. This prediction can either be the initial prediction or the final prediction from the last iteration of the timestep.

$$\hat{\mathbf{x}}_{k+1|k} = A\hat{\mathbf{x}}_{k|k} \tag{3-4}$$

Next step is predicting the next iteration of the covariance matrix P. This covariance describes the uncertainty of each of the state estimations.

$$P_{k+1|k} = AP_{k|k}A^T + Q \tag{3-5}$$

With the covariance matrix prediction, the optimal kalman gain is calculated. This gain will be used in the following steps, where a measurment is used to update both the predictions of the state vector and covariance matrix.

$$K_{k+1} = P_{k+1|k}C^T(CP_{k+1|k}C^T + R)^{-1} \tag{3-6}$$

$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + K_{k+1}(\mathbf{y}_{k+1} - C\hat{\mathbf{x}}_{k+1|k}) \tag{3-7}$$

$$P_{k+1|k+1} = (I - K_{k+1}C)P_{k+1|k}(I - K_{k+1}C)^T + K_{k+1}CK_{k+1}^T \tag{3-8}$$

The updated state vector and covariance matrix are now ready to be used for the next prediction and update step.

Applying the previous kalman filter structure to the SORT tracking algorithm starts with choosing a state vector for the bounding box model. This can be found in Equation 3-9:

$$\mathbf{x} = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}]^T,\ \mathbf{y} = [u, v, s, r]^T \tag{3-9}$$

Here, u and v are the pixel coordinates representing the center of the bounding box, s is the scale of the bounding box represented by the area and r is the ratio between the width and the height of the bounding box. The state vector also contains the rate of which the center coordinates and scale changes, but not the ratio, as it is assumed that this will remain constant. The measurement vector does not contain the state velocities, as they cannot be measured and have to be estimated.

The model chosen by the authors of the SORT algorithm is a constant velocity model, defining the state transition and observation matrices in the following way:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{3-10}$$

The process noise covariance Q and measurement noise covariance R are chosen to be:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix} \tag{3-11}$$

The process noise variances for the bounding box velocities are lower than the bounding box position variances. There is not a specific reason stated for this in the paper that introduced SORT, however it can be assumed that bigger variances for the velocity made the predictions unstable. The last element on the diagonal of Q is the variance of $\dot{u}$ multiplied with the variance of $\dot{v}$, since the scale velocity $\dot{s}$ is the product of $\dot{u}$ and $\dot{v}$.

The measurement noise variance for the scale s and ratio r are higher than the variance for u and v, since both are calculated from multiple bounding box variables. This means that any measurement noise present is multiplied in these variables.

The initial covariance matrix can be found in Equation 3-12 and the initial state prediction in Equation 3-13. The initial velocities of the bounding box are initialized to zero, since it is not feasible to predict these with only one position measurement. This results in the covariance matrix having high initial values for the three velocity components in the state vector.

$$P_{0|0} = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10000 \end{bmatrix} \tag{3-12}$$

$$\hat{\mathbf{x}}_{0|0} = [u_d, v_d, s_d, r_d, 0, 0, 0] \tag{3-13}$$

3-2-3 Matching Detections to Tracks

The remaining step of the object tracking algorithm is to match the detections from the YOLOv3 network to objects already tracked. This is done using a cost matrix, which can be found in Equation 3-14. The cost values are chosen to be the intersection-over-union, often abbreviated to IoU, of the last predictions of each track P_m and the bounding boxes of the detections D_n. The formula of the intersection-over-union can be found in equation 3-14, which makes clear that the IoU is calculated by dividing the overlapping area of two bounding boxes by the total area that the two bounding boxes occupy.

$$J(D,P) = \begin{bmatrix} IoU(D_1,P_1) & \dots & IoU(D_1,P_M) \\ IoU(D_2,P_1) & \dots & IoU(D_2,P_M) \\ \vdots & \ddots & \vdots \\ IoU(D_N,P_1) & \dots & IoU(D_N,P_M) \end{bmatrix} \quad IoU(D_n,P_m) = \frac{D_n \cap P_m}{D_n \cup P_m} \tag{3-14}$$

Figure 3-5: Frame of the TuSimple velocity estimation challenge with annotations shown [23]

The complete cost matrix is used in an assignment problem that is solved with the Hungarian algorithm [49], which assigns detections to track in an optimal way.

3-3 Velocity Estimation

To convert an object tracking algorithm to a velocity estimation algorithm can seem like a simple task, but is in this case quite difficult because of the lack of depth from the object tracking algorithm. However, with calibration values of the camera and knowledge about the camera perspective, an accurate guess can be made about the velocity of an object, without any knowledge of the depth. This section will go into detail about two implementations found in the literature and will explain how the novel aproaches work, inspired by classic Systems and Control Topics.

3-3-1 TuSimple Dataset

The data needed for training velocity estimation algorithms is provided by the TUSimple velocity estimation challenge dataset [23]. This set consists of 1074 training videos and 269 testing videos, each 2 seconds long and captured at 20 frames per second. The scenarios picture several vehicles driving on a highway, as can be seen in Figure 3-5.

Each last frame of the dataset is partially labeled with bounding boxes of vehicles, a position vector that corresponds to the distance between the origin of ego vehicle and the closest point on the labeled vehicle, and a relative velocity vehicle. The intrinsic camera matrix and the mounting height of the camera is also given in the dataset. The objective of the dataset is to determine the relative velocities of these vehicles. The training split of the dataset includes these velocities in the labels, while the labels in the test split only contains bounding boxes as an indication for which vehicles a velocity estimation algorithm needs to estimate a relative velocity. The evaluation metric that is used to quantify the performance of a velocity estimation algorithm is the mean squared error between the ground truth velocity vector and the velocity vector estimated by the algorithm. The authors of the dataset acknowledge

that estimating these velocities becomes harder the further the vehicles are located from the camera. This is why the the evaluation is split up in three categories: vehicles near to the camera (0-20 metres from the camera), vehicles at a medium distance from the camera (20-45 metres from the camera) and vehicles far away from the camera (45 metres and further). The complete evaluation error is the average of the three previous mentioned.

3-3-2 Existing Approaches

This thesis research is not the first that uses the TUSimple dataset for developing a velocity estimation algorithm. There are already two methods existing in academic literature: one article from Kampelmühler *et al.* [24] and one article from McCraith *et al.* [25].

Kampelmühler estimates the velocity from the TUSimple dataset by combining four distinct neural network algorithms. The first network is a vehicle tracking algorithm, very similar to the SORT and YOLOv3 combination used in this thesis. Accompanying this tracker is a depth neural network, which tries to estimate the depth information of the scenario, and a flow neural network, which attempts to estimate which parts of the image are stationary and which parts contain moving objects.

Using the bounding boxes from the tracking network and a procedure called local aggregation, the outputs of the depth network and flow network are reduced to one scalar per vehicle per video frame each. This results in each tracked vehicle being described by six variables per frame, four bounding box coordinates, one depth scalar and one float scalar. To estimate the relative velocity of the vehicle, five key frames are selected out of the 40 video frames to be used for a final neural network algorithm, that takes the in total 30 variables and outputs a 2D relative velocity vector, and a 2D position vector. A diagram of the complete velocity algorithm from Kampelmühler *et al.* can be seen in Figure 3-6.

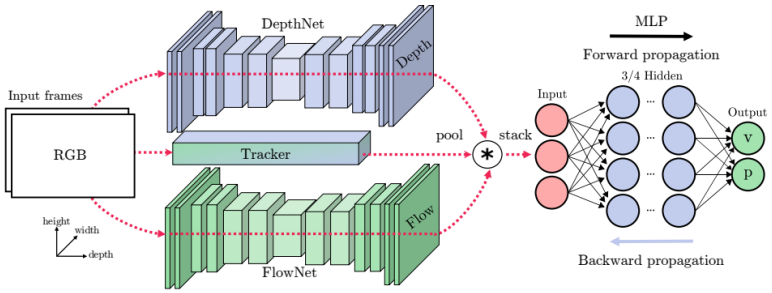

Figure 3-6: Velocity estimation algorithm using the depth and flow neural networks [24]

The algorithm scores high on the TUSimple velocity estimation evaluation metric, with a mean squared velocity error of 0.18 in the nearby vehicle category, an error of 0.66 in the medium vehicle distance category and a 3.07 error in the far away vehicles category. These were the top results at the time, making the algorithm state-of-the-art. The only drawback is that the algorithm requires 423 ms to process the 2 second video into a relative velocity, making it not viable to use in real-time velocity estimation.

The second algorithm is found in a research article published in 2021 [25] by McCraith *et al.*. They state that the depth and flow network used by the first method are not necessary for velocity estimation and that the bounding boxes alone provide adequate information for velocity estimation. This is showed by training a similar network Kampelmühler used to process the five key frames to estimate vehicle velocities. However, instead of using only five key frames, McCraith opted for using the bounding boxes present in all 40 frames, increasing the network input from 30 to 160. Their article reported a decreased error of 1.29, with a mean squared error of 0.18 for near vehicles, 0.70 for medium far vehicles, and 2.99 for far vehicles. The greatest improvement is a significant decrease in computing time, reducing it to 10 milliseconds. A next improvement step that is implemented as well is the use of synthetic data to train the network with more examples. This is not exlored further in this thesis, as the reported results were marginal, decreasing the overall error to 1.28 (0.17 for near, 0.72 for medium, 2.96 for far). A complete schematic of the velocity estimation algorithm from McCraith *et al.* can be found in Figure 3-7.

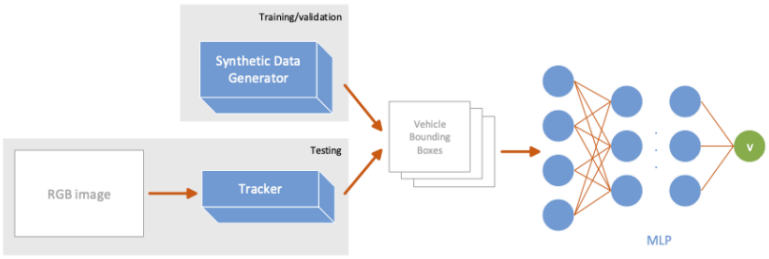

Figure 3-7: Velocity estimation algorithm using synthetic data [25]

3-3-3 Position mapping and Kalman Filter

Both approaches described in the previous section try to estimate the velocity with one prediction per video, using the vehicle and environment information of multiple frames. This comes with the disadvantage that an accurate velocity estimation can only be made two seconds after the discovery of the vehicle by the tracking algorithm, no matter how fast the velocity estimation algorithm is. The velocity estimation algorithm proposed in this thesis uses a more inaccurate bounding box to position mapping, and uses a kalman filter to estimate the relative velocity from these position estimates. This ensures that the whole estimation algorithm, from the object detection algorithm to an estimated velocity is a real-time algorithm.

In total, two bounding box to position maps are being compared to the existing approaches in this thesis. The first map is a quadratic matrix equation (see Eq. 3-15), one for each position coordinate: x and z. The bounding box is represented by the vector $\mathbf{b}$ The matrix values of H_x, H_z, M_x and M_z are determined via a least squares approach, where the ground truth value pairs of position and bounding boxes are from the TuSimple velocity estimation dataset, more specifically the training split.

$$
\begin{aligned}
x &= \mathbf{b}^T M_x \mathbf{b} + H_x \mathbf{b} \\
z &= \mathbf{b}^T M_z \mathbf{b} + M_z \mathbf{b} \\
\mathbf{b} &= \begin{bmatrix} c_x \\ c_y \\ w \\ h \end{bmatrix}
\end{aligned} \tag{3-15}
$$

The second map uses the intrinsic camera matrix and camera orientation to reverse project a point on the bounding into a real-world position value. An illustration with corresponding parameters can be found in Figure 3-8. The reverse projection is done using the intrinsic properties of the camera, such as focal points and the principal point of the image, and the orientation of the camera, which is in the case of the camera used in the TuSimple dataset, only a translation in the Y direction. The full camera projection equation can be found in equation 3-16.

The parameters found in this equation are all given in the TuSimple dataset, as stated before. However, with regular camera projection, a three dimensional representation is reduced into a two dimensional one. This means that in order to do a reverse projection, an extra assumption about the real world coordinates has to be made. Here, the assumption is made that the bottom middle point of a vehicle bounding box, corresponds to a Y-coordinate of zero. This reduces the camera projection equation 3-16 to the one found in equation 3-17, which can be solved with the given the bounding box coordinates.

$$
\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_{cam,x} \\ 0 & f_y & c_{cam,y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & h_{cam} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3-16}
$$

$$
\begin{bmatrix} c_x \\ c_y + \frac{h}{2} \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_{cam,x} \\ 0 & f_y & c_{cam,y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & h_{cam} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ 0 \\ Z \\ 1 \end{bmatrix} \tag{3-17}
$$

Positions from these two bounding box to position maps are then used as measurements for a kalman filter that estimates the relative vehicle velocity. It is based on a constant velocity model, similar to the one used in the SORT tracking algorithm, however this time using the estimated real world coordinates of vehicles. The model can be found in Equation 3-18. ΔT is the time interval between frames, which is 0.05 seconds for the videos present in the dataset.

The process noise covariance matrix is also different from the one used in the SORT tracker, as the process noise is modeled as random, zero mean accelerations σ_a from the vehicles, which have not been accounted for in the constant velocity model. The measurement noise represents the inaccuracies introduced by mapping the bounding box coordinates to real world positions. The accompanying covariance matrix is chosen to be the variance and covariance of the error values between the output of the mapping and the ground truth position values. The

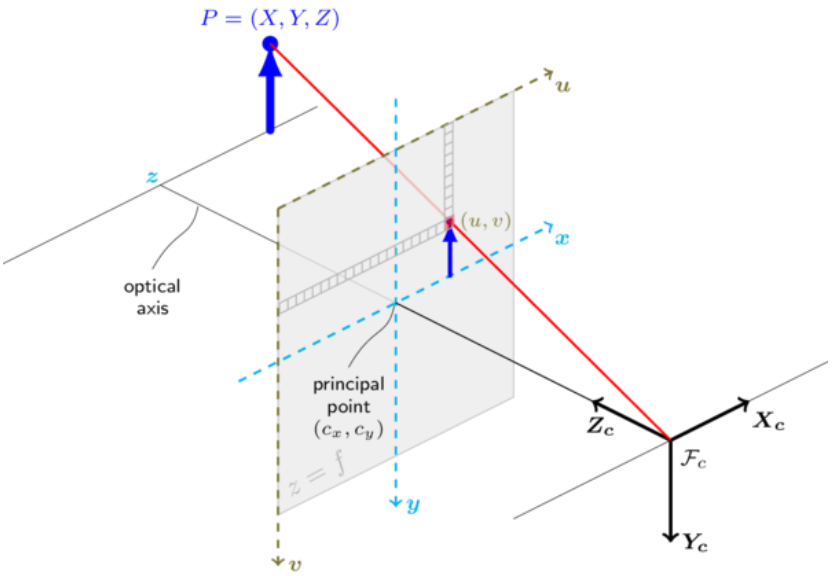

Figure 3-8: Illustration of real-world coordinates projecting on a camera image [26]

variance and covariance values for the camera mapping are slightly higher than the quadratic mapping variance and covariance. This is due to the real-world point chosen by the camera mapping is different that the one provided by the ground truth positions, as this is a point on the vehicle that is the closest to the camera. Both covariance matrices can be found in equation 3-20.

$$\mathbf{x}_{k+1} = \begin{bmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x}_k + \mathbf{w}_k \tag{3-18}$$

$$\mathbf{y}_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \mathbf{x}_k + \mathbf{v}_k \tag{3-19}$$

$$Q = \begin{bmatrix} \frac{\Delta T^4}{4} & 0 & \frac{\Delta T^3}{2} & 0 \\ 0 & \frac{\Delta T^4}{4} & 0 & \frac{\Delta T^3}{2} \\ \frac{\Delta T^3}{2} & 0 & \Delta T^2 & 0 \\ 0 & \frac{\Delta T^3}{2} & 0 & \Delta T^2 \end{bmatrix} \sigma_a^2, \; R = \begin{bmatrix} Var(x) & Cov(x,z) \\ Cov(x,z) & Var(z) \end{bmatrix} \tag{3-20}$$

Chapter 4

Algorithm Testing Results

This chapter will showcase the results from the implementations described in Chapter 3. First, in Section 4-1, results of just the object detection will be shown and compared to other object detection networks. Section 4-2 contains the results from the object tracking algorithm SORT, first with pre-generated detection, not generated by the implemented object detector, and then tracking results with the implemented object detector. Lastly, Section 4-3 will showcase the velocity estimation results and compare them to the solutions found in the literature.

4-1 Object Detection

The Ultralytics YOLOv3 implementation in pytorch was trained on the Berkely Deep Drive 100K dataset, using a NVIDIA 3070 GPU and an intel Core i7-11800H. Due to the limited amount of available RAM, the mini-batch size was fixed on 8, while 32 was recommended by the authors. The initial learning rate was set to 10^{-2}, the final learning rate was 10^{-3} and a weight decay of 5×10^{-4} was selected. After training for 25 epochs, the final validation showed a total mAP of 0.587 at an IoU threshold of 0.5, which means that the Intersection over Union between the detected object and ground truth has to be 0.5 or higher to count as a True Positive detection. The total mAP value is on the low side, but this is misleading as the individual mAP values of the detection classes of rider, motorcycle and bicycle are lowering the total average significantly. This is due to the small amount of labeled examples in the dataset. The car class, which is the most important for the velocity estimation algorithm, has an mAP value of 0.789 at an IoU threshold of 0.5, which is significantly better than the average. The complete precision-recall curve can be found in Figure 4-1. An example of a validation batch, including labelled objects, can be found in Figure 4-3. To confirm that the trained YOLOv3 network does not mislabel objects, the confusion matrix of the final validation run is analyzed as well. This matrix can be found in Figure 4-2. The elements on the diagonal represent the amount of times a ground truth object is detected with the correct class. The darker the square, the larger the amount of ground truth objects are

detected correctly. The squares off the diagonal represent the amount of ground truth objects in the validation set that are detected by the network, but with the wrong class label. The bottom row represents, the amount of false negative detections, where an object should have been detected, but is not being detected by the network. The right most column are the false positive detections, where the network detects an object, but no ground truth object is present at the specific location. The confusion matrix has the darkest squares on the diagonal, which means that missclassification is not an issue with the YOLOv3 network. The most common missclassification is the one between car and truck, which is understandable if looked at intuitively. A van could be seen as both a car and truck for example, causing confusion. One value in the confusion matrix that is also noticeable, is the amount of false postive car detections. However, further research proved that these false detections had a low confidence score. This meant that by setting the confidence threshold high enough, a value 0.5 was chosen, these false positives were almost negligible. Overall, the trained YOLOv3 network showed results that proved good enough for the YOLOv3 implementation to be used in the object tracking algorithm.

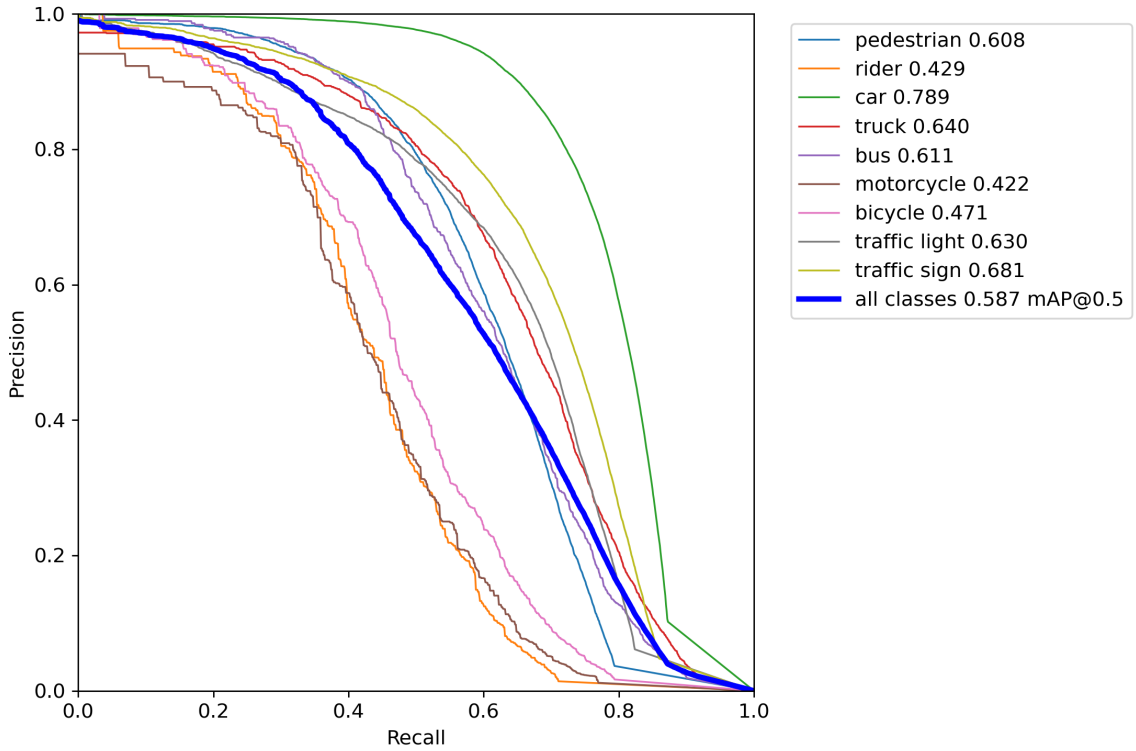

Figure 4-1: Precision Recall curve of the YOLOv3 object detector

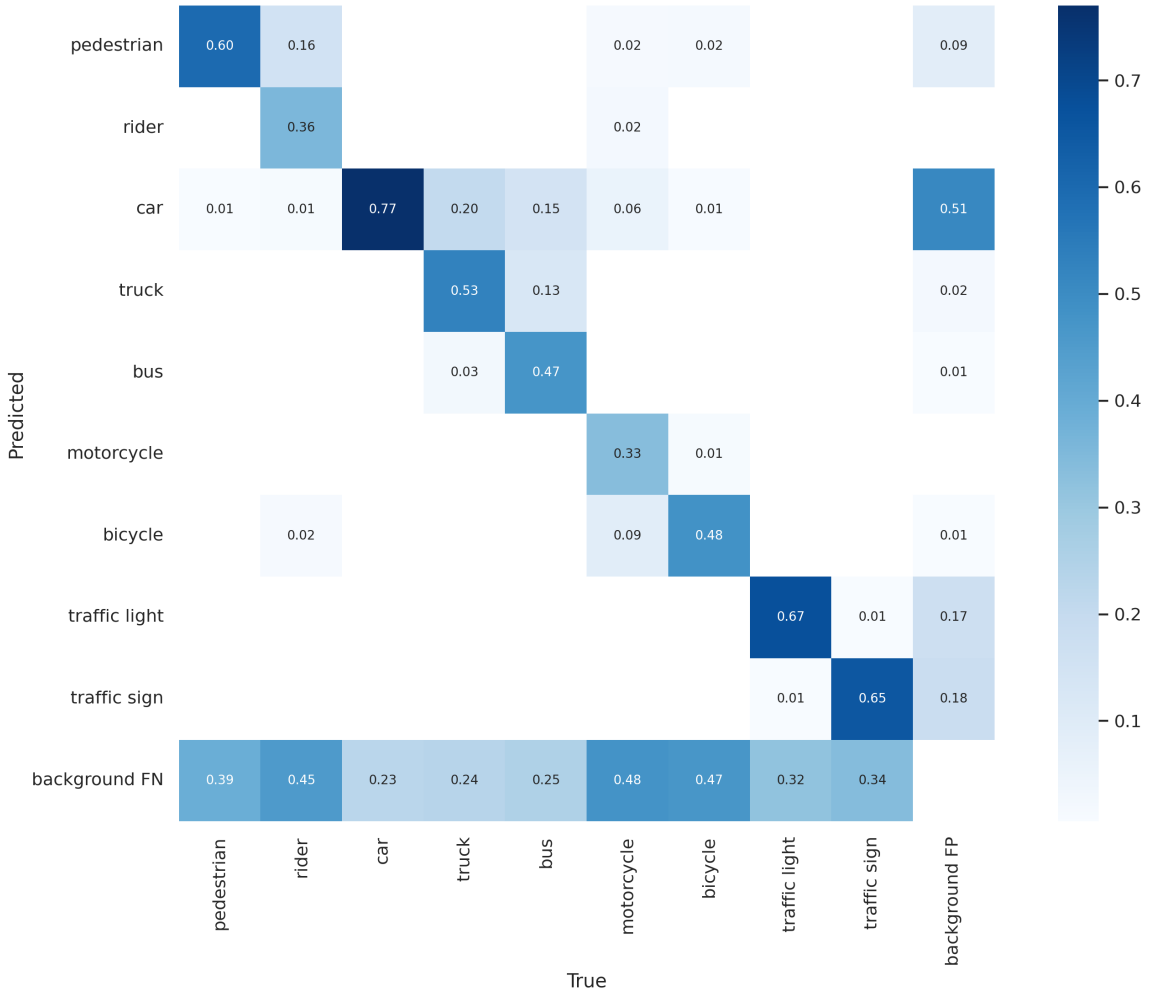

Figure 4-2: Confusion Matrix of the trained YOLOv3 object detection network

Figure 4-3: Example of a prediction batch at validation time

4-2 Object Tracking

The combination of YOLOv3 and the SORT Tracking algorithm is tested on the multi-object tracking part of the KITTI dataset (see section 2-2) as a way to quantify the tracking performance. The object tracking set of the Berkeley Deep Drive set was also considered to be used for evaluation, however the KITTI dataset is smaller and therefore less time consuming to evaluate on and this result is not seen as an end result, but as an intermediate result to a velocity estimation algorithm. The hyperparameters chosen for the SORT algorithm is an IoU threshold of 0.3, which means that if a detection and prediction of a tracked object have an IoU of 0.3 or higher, the detection will be matched to the tracked object, and the maximum time since last detection is set to three frames. The quantitative evaluation is done using a code repository provided by the KITTI dataset [50], which evaluates the tracking for only the pedestrian and car classes. The used metrics are the Multiple Object Tracking Accuracy (MOTA) and Multiple Object Tracking Precision (MOTP) metrics [45], as these metrics are intuitive to interpret.

The definition of MOTA is given in equation 4-1. It is a sum of three different error ratios over all the available frames, that is subtracted from a perfect score of one. The first of the three errors is a false negative matched detection (FN), where the ground truth tracked object has a detection in a specific frame, however the tracker did not match a detection to the tracked object. The second error is a false positive detection match (FP), where the ground truth tracked object did not contain a matched detection at a specific time frame, but the tracker did match a detection to the tracked object. The last error is the identity switch error (IDSW), where a tracker gives a tracked object a new identity number. This is often the case when a tracked object is lost by the tracker and then found again.

$$MOTA = 1 - \frac{\sum_t (FN_t + FP_t + IDSW_t)}{\sum_t GT_t} \tag{4-1}$$

The definition of MOTP is given in equation 4-2. This tracking metric is a measure on how precise the tracker is with localizing the tracked objects with respect to the ground truth tracked objects. Here c_t is the amount of detection matches in a specific frame and $d_{t,i}$ denotes the overlap between the bounding box provided by the tracker and the ground truth bounding box of the tracked object. This overlap is calculated using the Intersection over Union (IoU).

$$MOTP = \frac{\sum_{t,i} d_{t,i}}{\sum_t c_t} \tag{4-2}$$

The complete training split of the KITTI tracking dataset is used as an evaluation set for the YOLOv3 plus SORT tracking algorithm, as it has not been used for training by both algorithms. This results in an MOTA value of 71.82% and an MOTP value of 76.88% for the car class. The pedestrian class is not taken into account, as this class is not relevant for the task of velocity estimation. A main source of errors in the validation seem to be the identity switches that the object tracking algorithm makes, as an object gets occluded for more than three frames and the tracker assigns a new identity to the object. The ground truth dataset contains 564 cars that are tracked, however the tracker reported 777 tracked car objects, with a total of 201 identity switches over the complete dataset.

4-3 Velocity Estimation

After confirming that the detection and tracking algorithm both are performing adequately, it is possible to use the tracker in conjunction with the two different velocity estimation algorithms to answer the research question of this thesis and compare the results to the state-of-the-art methods using the same dataset. The velocity estimation was done in several stages, but the algorithm is capable of estimating and updating the estimation in real-time. First evaluation stage consisted of processing the 269 videos of the testing split with the YOLOv3 plus SORT tracking algorithm. This resulted in multiple tracked objects per video. Next, the tracked objects were compared with the ground truth bounding boxes present in the last frame, which represented the vehicles that needed their relative velocity estimated. The tracked object with the highest IoU was determined to be one of the vehicles of interest and the track was extracted from the rest. A side result of this was that of the 366 objects, 18 of them were not tracked for the full 40 frames of the video. A velocity can still be estimated using the mapping method, but will be less accurate.

Using the reverse camera projection mapping, a total error of 4.89 was achieved. The quadratic map using least squares resulted in a total error of 7.18, where the near vehicles gave a significantly higher error than the other distance categories. The errors for both of these methods are significantly higher than the state-of-the-art methods described in section 3-3. However, after replicating the algorithm used by McCraith *et al.*, the resulting error was 4.06, which was significantly higher than was reported. A table with the results, including the errors for each distance category can be found in table 4-1.

Method	$Error_{tot}$	$Error_{near}$	$Error_{med}$	$Error_{far}$
Kampelmühler [24]	1.30	0.18	0.66	3.07
McCraith [25] (reported)	1.29	0.18	0.70	2.99
McCcraith (replicated)	4.06	1.93	3.33	6.94
Quadratic Mapping	7.18	14.04	2.76	4.76
Camera Mapping	4.89	1.98	4.35	8.35

Table 4-1: Results of the TuSimple velocity estimation challenge.

Chapter 5

Conclusion

The aim of this report is to present the research done for the Master of Science thesis of Jelle Baltus by answering a question stated in the beginning of the report: "Is it possible to estimate relative velocities of vehicles surrounding the ego vehicle using a monocular camera with such an accuracy that meaningful conclusions can be made about the current traffic state?"

To answer this question, a velocity estimation algorithm is created with three major parts, object detection from images, tracking of objects using bounding box detections and velocity estimation using tracked objects.

After an extensive literature review, the YOLOv3 algorithm is chosen as object detection algorithm, more specifically a PyTorch implementation the algorithm. YOLOv3 is then trained on the Berkely Deep Drive 100K dataset to make the detections focus on vehicles found on public roads. The results after training were shown to be adequate enough, with a mAP value of 0.789 at an IoU threshold of 0.5, to be used as a base for the next stages.

To combine the detections from the YOLOv3 detection algorithm into a series of detections representing a unique real-world object, the SORT algorithm was chosen. SORT, standing for Simple Online and Realtime Tracking, predicts the locations of bounding boxes of existing tracked objects and matches these with incoming detections using the Hungarian algorithm. The main advantage is that this method is fast and accurate, making the detection algoritm the bottleneck. Results on a different metric of the Berkely Deep Drive dataset shows that the YOLOv3 and SORT combination have a MOTA score of 71.82 % and a MOTP score of 76.88%, which makes it an adequate object tracking base for the final stage of the velocity estimation algorithm. The main problem with the tracking algorithm is the switching of identities, as a tracked object is occluded behind another object for more than three frames.

The final stage of the velocity estimation challenge is using the detections from the tracked objects and extract velocities out of them. A novel method using a neural network to map the image coordinates to real-world positions and a kalman filter to estimate the velocity using the position. This is then compared to a state-of-the-art method using a neural network to process a sequence of bounding boxes directly into a velocity. The novel method of mapping

bounding box coordinates to real world positions had a best error of 4.89, which is significantly higher than the state-of-the-art methods found, however an attempt at recreating the results, resulted in similar errors to the position mapping method.

Overall, the question stated in the beginning of the conclusion can be answered with yes, it is possible to estimate relative velocities of surrounding vehicles in such a way that meaningful conclusions can be made about the current traffic state, however it is difficult to only use a monocular camera for estimating velocities, as the loss of depth significantly impairs estimation performance.

5-1 Future Recommendations

There are four major recommendations that can be applied to the presented velocity estimation algorithm, as it is intended to be a proof of concept that can be improved with further research. First of all, a more modern detection algorithm can be chosen, as neural networks are improving constantly, becoming faster and more accurate. For example, the YOLO algorithm already has received several upgrades, improving the detection performance. Secondly, the detection algorithm can be trained more extensively with upgraded hardware to get better detection results. For this project, the detection algorithm was trained for 25 hours on one GPU, but research papers in the field of object detection use significantly better hardware than this. Thirdly, the SORT algorithm has trouble with re-identifying objects that were occluded behind other objects as seen in the KITTI validation. An improvement to SORT, called DeepSORT exists [51], where the identity switching problem is solved by augmenting the SORT algorithm with neural network that reduces the tracked object to 128D feature vector. This feature vector is then used in combination with the Kalman filter to track object. The author shows that the new algorithm is less prone to identity switching, but the neural network reduces the real-time performance of the tracker. It could be argued that the DeepSORT algorithm is more suitable for a velocity estimation algorithm. Lastly, a different velocity estimation dataset could be used, or even created, as only one velocity and position data point per video is very sparse. Ideally, each video frame would contain a velocity vector and position data would be in the form of a LiDAR point-cloud or a stereo camera image, as this can lead to developing a better reverse mapping between bounding box and real-world coordinates.

Bibliography

[1] F. Bre, J. Gimenez, and V. Fachinotti, "Prediction of wind pressure coefficients on building surfaces using artificial neural networks," *Energy and Buildings*, vol. 158, 11 2017.

[2] O. Gibaru, "Neural network." https://www.oliviergibaru.org/courses/ML_NeuralNetwork.html, 2019.

[3] P. Solai, "Convolutions and backpropagations." https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c, Apr 2018.

[4] S. Kini, "Getting started with cnn." https://medium.com/@kinisanketh/getting-started-with-cnn-18c03efc7d06, Apr 2020.

[5] FirelordPhoenix, "Maxpoolsample2.png," Feb 2018.

[6] MATLAB, "What are convolutional neural networks?." https://nl.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512 7html, Mar 2017.

[7] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, pp. 541–551, 1989.

[8] X. Han, Y. Zhong, L. Cao, and L. Zhang, "Pre-trained alexnet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification," *Remote Sensing*, vol. 9, no. 8, 2017.

[9] S. University, "Cs231n lecture 9: Cnn architectures." http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf, 2017.

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[11] H. Choi, S. Ryu, and H. Kim, "Short-term load forecasting based on resnet and lstm," in *2018 IEEE SmartGridComm*, pp. 1–6, 10 2018.

[12] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587, 2014.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[14] R. Girshick, "Fast r-cnn," *IEEE International Conference on Computer Vision*, pp. 1440–1448, 2015.

[15] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on pattern analysis and machine learning*, vol. 39, no. 6, pp. 1137–1149, 2017.

[16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788, 2016.

[17] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu, and A. Berg, "Ssd: Single shot multibox detector," *EECV*, pp. 21–37, 2016.

[18] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013.

[19] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[20] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018.

[21] A. Kathuria, "What's new in yolo v3?." /urlhttps://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b, Apr 2018.

[22] R. Pereira, G. Carvalho, L. Garrote, and U. J. Nunes, "Sort and deep-sort based multi-object tracking for mobile robotics: Evaluation with new data association metrics," *Applied Sciences*, vol. 12, no. 3, 2022.

[23] "Tusimple velocity estimation challenge." https://github.com/TuSimple/tusimple-benchmark/tree/master/doc/velocity_estimation, June 2017.

[24] M. Kampelmühler, M. Muller, and C. Feichtenhofer, "Camera-based vehicle velocity estimation from monocular video," *arXiv preprint*, 2018.

[25] R. McCraith, L. Neumann, and A. Vedaldi, "Real time monocular vehicle velocity estimation using synthetic data," *arXiv preprint*, 2021.

[26] openCV, "Camera calibration and 3d reconstruction." https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[27] E. Russel, "9 things to know about google's maps data: Beyond the map | google cloud blog," Oct 2019.

[28] J. Lau, "Google maps 101: How ai helps predict traffic and determine routes," Sep 2020.

[29] S. Weckert, "Google maps hacks." https://www.simonweckert.com/googlemapshacks.html, 2020.

[30] ACEA, "Vehicles in use europe 2023," tech. rep., European Automobile Manufacturers' Association, January 2023.

[31] J. Baltus, "Developments in traffic congestion detection and object detection," 2022.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[33] D. Hubel and T. Wiesel, "Receptive fields of single neurones in the cat's striate cortex," *J. Physiol*, vol. 148, no. 3, pp. 574–591, 1959.

[34] D. Hubel and T. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *J. Physiol*, vol. 195, no. 1, pp. 215–241, 1968.

[35] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[36] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

[37] D. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[38] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Proc. Adv. Neural Inf. Process Syst.*, pp. 1106–1114, 2012.

[39] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[40] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[42] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders, "Selective search for object recognition," *Internation Journal of Computer Vision*, vol. 104, no. 2, pp. 154–171, 2012.

[43] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010.

[44] A. Geiger, P.Lenz, and R.Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361, 2012.

[45] A. Milan, L. Leal-Taixé, I. Reid, s. Roth, and K. Schindler, "Mot16: A benchmark for multi-object tracking," *arXiv preprint arXiv:1603.00831*, 2016.

[46] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2017.

[47] Ultralytics, "Yolov3 in pytorch." https://github.com/ultralytics/yolov3.

[48] A. Bewley, Z. Ge, F., L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," *CoRR*, vol. abs/1602.00763, 2016.

[49] H. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[50] A. H. Jonathon Luiten, "Trackeval." https://github.com/JonathonLuiten/TrackEval, 2020.

[51] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," 2017.