HAW HAMBURG

BACHELORTHESIS
Ronald Safowat Voth

Real-time detection of German street signs

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department Computer Science

Fakultät Technik und Informatik
Department Informatik

HAMBURG UNIVERSITY OF APPLIED SCIENCES
Hochschule für Angewandte Wissenschaften Hamburg

Ronald Safowat Voth

Real-time detection of German street signs

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Angewandte Informatik
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Michael Neitzke
Second examiner: Prof. Dr.-Ing. Martin Hübner

Day of delivery: 04. August 2021

Ronald Safowat Voth

Title of Thesis

Real-time detection of German street signs

Keywords

Realtime, Object Detection, German Street Signs, YOLO, Convolutional Nueral Network, Computer Vision, Deep learning

Abstract

In recent years, computer vision has become an essential part of our world. There is no lack of exciting tasks and applications for computer vision, from image classification to 3D position estimation. One of the areas we are mainly interested in and have worked on a lot is object recognition and detection. Object detection is employed for applications, including video surveillance, medical imaging, robot navigation, and autonomous driving. Object detection unlike Image classification, does not only predict what is in the image but also predict where in the image the predicted object is located.

Especially in the field of autonomous driving, computer vision and object detection techniques are used to evaluate and understand what is going on around the autonomous vehicle on the road. In every country and on every street in the world, there are rules and regulations to help the conveyance of pedestrians and vehicles be as safe, seamless, and efficient as possible. Street signs are used to ensure most of the regulations on roads. These street signs are continuously replaced and maintained to keep them readable and to adhere to new rules on different streets.

In Germany, each state is responsible for keeping a catalogue of all street signs and their locations at each point in time in their respective country. So when an incident like an accident occurs, there would not be the need for the dispute of whether a street sign was present or not to enforce these regulations.

The aim of this thesis is the research and development of the training and the evaluation of the proposed state-of-the-art object detection neural network YOLO for the real time detection of german street signs on a mobile device. We compare the versions of YOLOv3 (tiny and large) and evaluate whether any of them can sufficiently and reliably detect street signs from a mobile device at a fast and accurate enough rate when driving on the road. . .

Ronald Safowat Voth

Thema der Arbeit

Echtzeit Erkennung von Deutsche Straßenschilder

Stichworte

Echtzeit, Objekterkennung, Deutsche Straßenschilder, YOLO, Convolutional Neuronales Netzwerk, Computer Vision, Deep Learning

Kurzzusammenfassung

In den letzten Jahren ist die Computer Vision zu einem wesentlichen Bestandteil unserer Welt geworden. Es mangelt nicht an spannenden Aufgaben und Anwendungen für die Computer Vision, von der Bildklassifizierung bis zur 3D-Positionsbestimmung. Einer der Bereiche, an dem wir besonders interessiert sind und an dem wir viel gearbeitet haben, ist die Objekterkennung und -detektion. Die Objekterkennung wird für Anwendungen wie Videoüberwachung, medizinische Bildgebung, Roboternavigation und autonomes Fahren eingesetzt. Anders als bei der Bildklassifizierung wird bei der Objekterkennung nicht nur vorhergesagt, was sich auf dem Bild befindet, sondern auch, wo im Bild das vorhergesagte Objekt zu finden ist.

Vor allem im Bereich des autonomen Fahrens werden Computer Vision und Objekterkennungstechniken eingesetzt, um zu bewerten und zu verstehen, was um das autonome Fahrzeug herum auf der Straße vor sich geht. In jedem Land und auf jeder Straße der Welt gibt es Regeln und Vorschriften, um die Beförderung von Fußgängern und Fahrzeugen so sicher, reibungslos und effizient wie möglich zu gestalten. Die meisten Vorschriften auf den Straßen werden durch Straßenschilder gewährleistet. Diese Straßenschilder werden ständig ausgetauscht und gewartet, um sie lesbar zu halten und neue Regeln auf verschiedenen Straßen zu beachten.

In Deutschland ist jedes Bundesland dafür verantwortlich, einen Katalog aller Straßenschilder und ihrer Standorte zu jedem Zeitpunkt in seinem Land zu führen. Wenn sich also ein Vorfall wie ein Unfall ereignet, wäre es nicht nötig, darüber zu streiten, ob ein Straßenschild vorhanden war oder nicht, um diese Vorschriften durchzusetzen.

Das Ziel dieser Arbeit ist die Erforschung und Entwicklung des Trainings und der Evaluierung des vorgeschlagenen Neuronalen Netzes YOLO für die Echtzeit-Erkennung von deutschen

Straßenschildern auf einem mobilen Endgerät. Wir vergleichen die Versionen von YOLOv3 (Tiny und Large) und evaluieren, ob eine von ihnen ausreichend und zuverlässig Straßenschilder von einem mobilen Gerät schnell und genau genug während der Fahrt auf der Straße erkennen kann...

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Feedforward Neural Networks 1

1.1.1 Activation Functions 3

1.1.2 Loss Functions 4

1.1.3 Backpropagation 6

1.2 Adam (Adaptive Momentum) 7

1.3 CNN (Convolutional Neural Network) 8

1.4 Object Detection 9

1.4.1 Two and One Stage detection 11

1.4.2 IOU (Intersection Over Union) 11

1.4.3 NMS (Non-Max Suppression) 12

2 Methodology 14

2.1 Problem Formulation 14

2.2 Proposed Approach 15

2.2.1 YOLO (You Only Look Once) 16

2.2.1.1 Model Architecture 16

2.2.1.2 Detection 18

3 Experiments 21

3.1 Preparation 21

3.1.1 Data 21

3.1.2 Annotation Process 22

3.2 Preprocessing 23

3.2.1 Custom Anchors 24

3.2.2 Improving Class Imbalance 24
3.2.3 Image Augmentation 26
3.3 Experimental Setup 29
3.3.1 Implementation 29
3.3.2 Training 31
3.3.3 Testing 33
3.4 Experimental Results and Comparison 35
3.4.1 Training 36
3.4.2 Testing 38

4 Conclusion **44**

Bibliography **46**

A Appendix **50**

Declaration **51**

List of Figures

1.1 MLP with single hidden layer . . . 1
1.2 MLP with three hidden layer . . . 2
1.3 Sigmoid activation function . . . 3
1.4 Relu activation function . . . 4
1.5 Forward pass algorithm . . . 6
1.6 Backpropagation algorithm . . . 7
1.7 Adam algorithm . . . 8
1.8 An example of 2-D convolution . . . 9
1.9 An example of Image Classification (left) and Object Localization (right) . 10
1.10 IOU calculation . . . 12
1.11 Bounding box suppression . . . 12

2.1 Architecture overview . . . 16
2.2 Large and Tiny Yolov3 Architecture [31] . . . 17
2.3 Yolov1 loss function[23] . . . 18
2.4 Bounding box prediction[25] . . . 19

3.1 Class image samples . . . 21
3.2 Class Distribution Formula . . . 22
3.3 Annotation file sample . . . 23
3.4 K-means Distance Metric [24] . . . 24
3.5 Class Weights Formula . . . 25
3.6 Distortion . . . 27
3.7 Rotate . . . 27
3.8 Scale . . . 28
3.9 Shear . . . 28
3.10 Translate . . . 28
3.11 Implementation Requirements . . . 29
3.12 Model training . . . 30

3.13 Training Sequence diagram 31
3.14 **Stage 1**: Tiny Yolov3 Learning curve 37
3.16 **Stage 2**: Tiny and Large Yolov3 Learning curve 38
3.17 Large YOLOv3 Precision-Recall curves 40
3.18 Tiny YOLOv3 Precision-Recall curves 41
3.20 Large YOLOv3 Detections 42
3.22 Tiny YOLOv3 Good-Detections 42
3.24 Tiny YOLOv3 Bad-Detections 43

List of Tables

3.1 Class Sample Distribution 22
3.2 Custom Anchors 24
3.3 Class Weights 26
3.4 Training Stage 1 32
3.5 Training Stage 2 32
3.6 Training hardware specifications 33
3.7 Test Configuration 34
3.8 Testing hardware specifications 34
3.9 Stage 1 loss 36
3.10 Stage 2 loss 36
3.11 Processing results on device 38
3.12 Class Detections 39
3.13 Processing results on device 41

1 Introduction

To begin, we will cover the most relevant aspects of the theory of deep learning with respect to the model to be trained. We will start with the basics and work our way through to more advanced topics such as convolutional neural networks and object detection.

1.1 Feedforward Neural Networks

Feedforward neural networks, also known as multilayer perceptrons (MLPs), are the backbone of deep learning. Basically, they are a group of parametric functions that together, sufficiently approximate any given real valued function. Each of the parametric functions is represented by a layer in the network, and the scalar output of a layer is called a unit or a neuron. Therefore, the deeper the neural network is, the more complex the function that it approximates [12].

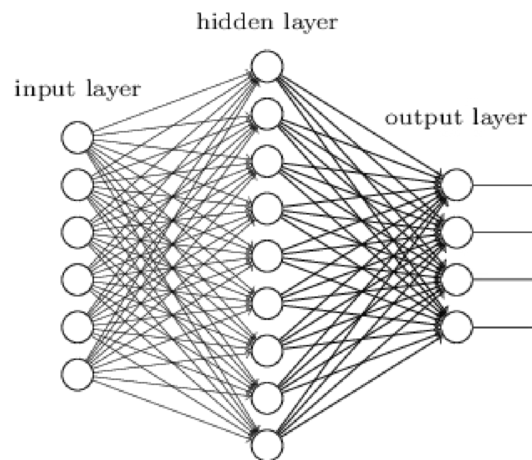


Figure 1.1: MLP with single hidden layer

Multilayer perceptrons were one of the first successful non-linear learning algorithms [27]. There are two types of perceptron networks, a multilayer perceptron with more than one hidden layer shown in Fig:1.2, and a single layer perceptron (SLP) that has exactly one hidden layer illustrated in Fig: 1.1. Single layer perceptron is also referred to as a vanilla MLP. MLPs can be considered as universal approximators in that with a high enough number of hidden units, they can sufficiently approximate any arbitrarily complex smooth function. This is because any function that is seemingly not linearly separable can become separable through non-linear transformations into a new feature space [12].

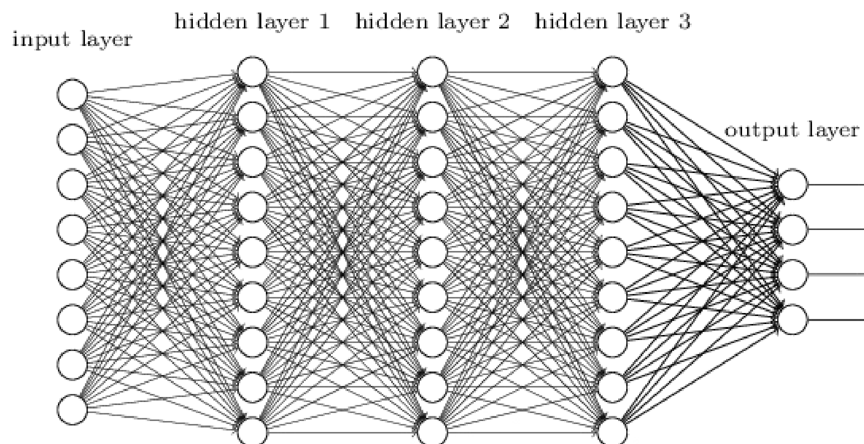


Figure 1.2: MLP with three hidden layer

The rationale behind MLPs is to apply simple affine transformations to obtain highly non-linear transformations [12]. This can be achieved by combining affine transformations and element-wise non-linearities. A deep feedforward neural network can be defined as

$$h^k = g(\beta^k + W_k h^{k-1})$$

where $g(.)$ is some activation function, β is the bias, W is a weight matrix, and h^{k-1} the output of the previous layer. Here k denotes the layer. If $k = 0$, then $h^0 = X$ is the input of the network. The output of h^k (for $k > 0$) is defined by the activation function $g(.)$, which is usually non-linear but can also be piecewise linear or simply linear.

1.1.1 Activation Functions

Activation functions play an essential role in neural networks because they define the range of the output and therefore directly impact the performance of the network [12]. We will cover the the sigmoid activation and the rectified linear unit (relu) activation function as they are relevant to our model to be trained. Activation functions in general are combined with the affine transformation $\beta + Wx$, which is applied element-wise: $z = \beta + Wx$, such that $h = g(z)$ for $g(.)$ being the activation function of choice. From here on, we will refer to the affine transformation $\beta + Wx$ simply as z.

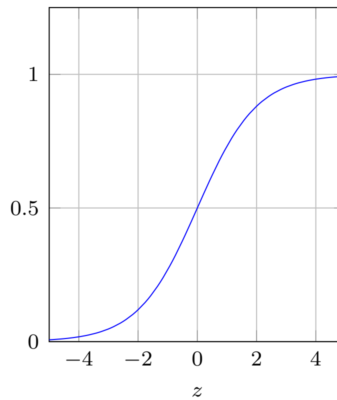


Figure 1.3: Sigmoid activation function

The Sigmoid activation function, in Fig: 1.3, is defined as

$$g(z) = \frac{1}{1 + exp(-z)}$$

.

The Sigmoid function is usually written as σ, such that $\sigma(z)$ and its derivative is $\sigma * (1 - \sigma)$. The output of the Sigmoid is defined in the range $[0, 1]$. It is commonly used for binary classification and as such is rooted motivationally in probability theory. The Sigmoid activation function is typically found in the output layer of the network that solves a classification problem, to denote the likelihood or probability of the predicted label.

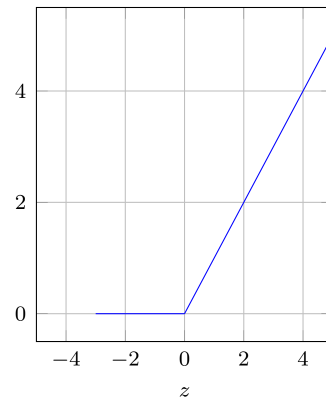


Figure 1.4: Relu activation function

First introduced by [13], the rectified linear unit (relu) has been demonstrated to improve the performance of neural networks [16]. Consequently, it has became one of the more frequently used activation functions [22]. ReLU depicted in Fig:1.4 and defined as $g(z) = max(0, z)$, is a piecewise linear function and one of its major benefits is that it introduces sparsity into the network [11]. Sparsity is the property that a subset of the model parameters has a value of exactly zero [10]. Sparsity facilitates better linear separability, in that sparse representations are very likely to be linearly separable, or at least require fewer non-linear transformations to become linearly separable than dense representations. Another advantage of the relu activation function is that it reduces the probability of the vanishing gradient problem and therefore allows for faster learning through backpropagation, which is the mechanism by which neural networks learn, described in section:1.1.3. The vanishing gradient problem arises from the fact that gradients become extremely small and the parameter updates of the network through backpropagation become insignificant, which means that no real learning takes place. The gradient for relu is a constant value, 1 for $z > 0$, which makes its derivation fast and trivial. In contrast, the sigmoid activation functions gradient becomes increasingly small as the absolute value of x in z increases.

1.1.2 Loss Functions

The loss function, also known as the error or objective function, is used to measure how well a model is performing and it is the function that the network is trying to minimize. The loss function defines the problem that the network is learning to solve and therefore should by no means be chosen arbitrarily [12].

The squared error loss, defined as

$$L(\hat{y}, y) = ||\hat{y} - y||^2$$

is based on the Euclidean distance.

The squared error loss is the backbone of other loss functions because it is convex and smooth. The mean squared error (MSE) loss as its name suggests, is based on the squared error loss and is one of the most widely used loss functions in linear regression. Linear regression is commonly used when the dependent variable y is continues (usually numeric) and the nature of the regression is linear.
MSE is defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2,$$

where N is the number of total observations, $\frac{1}{N}\sum_{i=1}^{N}$ is the mean, y_i the actual observation or ground truth and $\hat{y}_i$ is the prediction made, i.e. $\hat{y}_i = \beta + Wx_i$.

Two of the most widely used loss functions in logistic regression, is the Binary Cross Entropy (BCE) also referred to as Log Loss and the Categorical Cross Entropy. Logistic regression is employed when the dependent variable y is binary in nature, thus having a value of 1 or 0.
BCE is defined as:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)$$

and Categorical Cross Entropy is defined as:

$$CategoricalCrossEntropy = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(\hat{y}_i),$$

where N is the number of scalar values in the model output, $\hat{y}_i$ is the ith scalar value in the model output and y_i is the corresponding target value.
The BCE is used for classification problems with exactly two exclusive classes and Categorical Cross Entropy is used for classification problems with more than two exclusive classes. BCE is commonly done with a sigmoid activation function where as the Categorical Cross Entropy is done with a softmax activation function.

1.1.3 Backpropagation

The backpropagation algorithm is the method by which the loss function of the network is minimized, thus this algorithm is only part of the learning process of a neural network and not the method by which it learns [12].
Given a scalar loss function L, the backpropagation algorithm computes its derivative with respect to the model parameters by passing the gradient of the difference of the predicted and target value, backwards through the network thereby adjusting the weights in each layer. The partial derivative of the loss function L with respect to the parameters determines whether the weights should be decreased or increased in order to produce better predictions.
The backpropagation algorithm is defined through composite functions such that $\hat{y}(g(f(x)))$. If x influences y(.) through some real-valued function f(.) and some activation function g(.), then we are interested in the change in x that propagates a tiny change in y via g and f. The focus of backpropagation is on the loss function L which is defined as

$$L(y - \hat{y}(g(f(x)))).$$

Lets take a look at the forward pass and the backpropagation algorithms as defined by [12].

Algorithm 1: The forward pass algorithm

1. $h_0 = x$
2. **for** k = *1, ...,* K **do**
3. $\quad a^{(k)} = b^{(k)} + W^{(k)}h^{(k-1)}$
4. $\quad h^{(k)} = f(a^{(k)})$
5. **end**
6. $\hat{y} = h^{(K)}$
7. $J = L(\hat{y}, y)$

Figure 1.5: Forward pass algorithm

Algorithm 2: The backpropagation algorithm

1 $g \leftarrow \nabla_{\hat{y}} L = \nabla_{\hat{y}} L(\hat{y}, y)$
2 **for** $k = K, \ldots, 1$ **do**
3 $\quad g \leftarrow \nabla_{a^{(k)}} J = g \odot f'(a^{(k)})$
4 $\quad \nabla_{b^{(k)}} L = g$
5 $\quad \nabla_{W^{(k)}} L = g h^{(k-1)T}$
6 $\quad g \leftarrow \nabla_{h^{(k-1)}} L = W^{(k)T} g$
7 **end**

Figure 1.6: Backpropagation algorithm

In the above Algorithms, x is the input matrix, K is the number of layers, W is the weight matrix, b is the bias, $\hat{y}$ is the prediction, and J is the error term. The $\odot$ symbol denotes the Hadamard (element-wise) product.

1.2 Adam (Adaptive Momentum)

Adaptive momentum, or simply Adam, is a first-order gradient based optimization algorithm for stochastic objective functions introduced by [17]. Adam is computationally efficient and has low memory requirements. It computes adaptive learning rates individually for the parameters from estimates of first and second moments of the gradients

Algorithm 3: The adaptive momentum algorithm, or Adam.

1 $\alpha; \beta_1, \beta_2 \in [0, 1); f(\theta), \theta_0$
2 $m_0 \leftarrow 0; v_0 \leftarrow 0; t \leftarrow 0$
3 **while** θ_0 *not converged* **do**
4 $\quad t \leftarrow t + 1$
5 $\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
6 $\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
7 $\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
8 $\quad \hat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}$
9 $\quad \hat{v}_t \leftarrow \frac{v_t}{(1-\beta_2^t)}$
10 $\quad \theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)}$
11 **end**
12 **return** θ_t

Figure 1.7: Adam algorithm

In the adam algorithm shown in Fig: 1.7, α is the learning rate, $\beta 1$ and $\beta 2$ are the decay rates for first and second order moment estimates respectively, $f(\theta)$ is the objective function with parameters θ, θ_0 is the initial parameter vector, m_0 and v_0 are the first and second moment vectors respectively, and t is the timestep.

1.3 CNN (Convolutional Neural Network)

A Convolutional neural network, also referred to as CNN or ConvNet, is a type of neural network that employs a mathematical operation called convolution, for processing data with a known grid-like topology [12]. Depending on the data, the convolution is either 1- or 2-Dimensional. For example, time-series data can be seen as a 1-Dimensional grid of samples taking at regular time intervals and image data can be seen as a 2-Dimensional grid of pixels.
Generally, a convolution is an operation on two functions of a real-valued argument. It basically refers to mathematically combining two functions to produce a third function. i.e. it merges two sets of information into one. In 2-Dimensional convolution, the first function $x(.)$ is the image data or pixel matrix, the second function $w(.)$ which is another matrix of values is combined to produce the output $y(.)$. In CNN terminology, the functions $x(.)$ and $w(.)$ are referred to as the **input** and the **kernel**, respectively and their output $y(.)$, as the **feature map**. The kernel is first flipped in both the horizontal and vertical direction then it slides over the input data (left to right then top to bottom) and performs an element-wise multiplication with the region of the input data it is currently on, then the result is summed up into a single output pixel (see Fig:1.8). The 2-Dimensional convolution operation is defined as follows:

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)K(i-m,j-n).$$

In the context of machine learning, the learning algorithm learns the appropriate values for the kernel in the appropriate place[12]. (i.e. which weights at which layers will produce the feature map that can be classified and predicted from the given input).
The convolution operation utilizes sparse interactions, parameter sharing and equivariant representations, and all of these can improve the performance of a neural network. A convolutional neural networks sparse interactions introduces a non-linear property to its

hidden layer. This non-linear transformation is achieved with a kernel which is smaller than the input, yielding fewer parameters and thus reducing the memory requirements of the neural network and the operations needed to compute the output. For example, a normal neural network with i inputs and o outputs requires matrix multiplication with i x o parameters, whereas a CNN with m sparse interactions requires only m x o parameters [12].

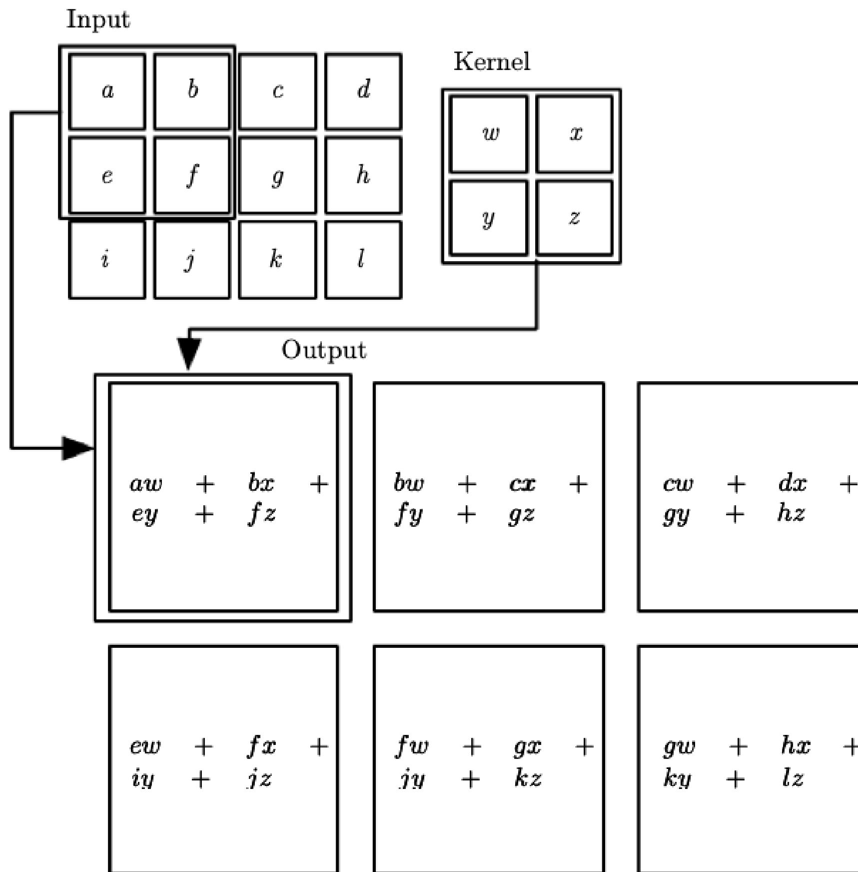


Figure 1.8: An example of 2-D convolution

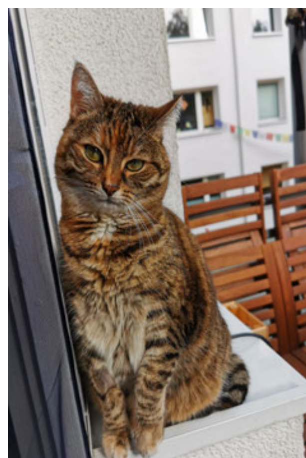
1.4 Object Detection

This section will describe object detection by covering what components make up object detection and the different approaches to it.

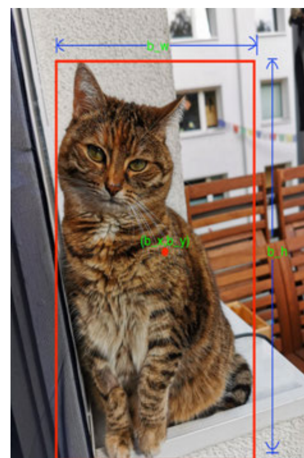
Object detection is one of the areas in computer vision rapidly being adopted and is currently working a lot better than a few years ago. To understand object detection, there are two fundamental tasks that we need to be aware of; image classification and object localization. These two tasks are, in essence, the sub-tasks or -components of object detection.

Image classification is the task of categorizing an image depending on its contents, i.e. an algorithm is shown an image, and it is responsible for classifying the image into a category. For example, an algorithm or neural network is shown a cat photo (**input**), and it identifies the cat in the image and categorizes the image as an image of a cat (**output**). The output is usually a probability that the input is of a specific class, (i.e. there's a 99% probability that this input is a cat).

In the task of **object localization**, the algorithm is not only responsible for classifying the image but also for drawing a rectangle around the position of the object in the image it classifies. The term localization refers to finding where in the photo the object is. The rectangle drawn around the object in the photo is known as a **bounding box**, as it specifies the boundaries of the object in the image. The parameters used to describe this bounding box by a neural network are b_x, b_y which represent the pixel Cartesian coordinates denoting the centre of the bounding box and b_h, b_w which are the height and width of the bounding box. These output parameters are normalized, i.e. if b_x is 0.5 and b_y is 0.5, then the centre of the bounding box is located at the centre of the image. So the actual pixel value for b_x is 0.5 of the maximum pixel width of the image, etc.



(a) $Cat : 0.99, Dog : 0.03$



(b) b_x, b_y, b_h, b_w

Figure 1.9: An example of Image Classification (left) and Object Localization (right)

Object detection takes this a step further by not only classifying and localizing one object in the image but multiple instances of objects belonging to different categories.

1.4.1 Two and One Stage detection

There are two main approaches to solving the task of object detection. The primary and most prominent difference between the two is that one is faster but less accurate, and the other is accurate but very slow.

Two-stage object detection, as the name suggests, splits the task of object detection into two stages. The first stage is the selection of a region that may or may not contain an object (this directly corresponds to the bounding box for the object if it exists in the region) and the second stage is to run image classification on the selected region. One of the first techniques of two-stage object detection is the sliding windows technique. In this technique, a window (bounding box) size is chosen and slid from left to right, up to down of the image. After each step in the slide, the window's content is cropped and passed to a ConvNet for classification. This process is repeated with a slightly larger window size over and over until the window size is that of the image. The computational cost of this approach is very high and therefore causes the whole detection process to be very slow. Still, when an object is detected, the accuracy is very high because the neural network used for the classification can be as extensive as needed, and its only task is to classify. So, in general, the two-stage detection process involves a region proposal stage, and then the proposed regions are piped to a classifier for prediction. Some of the detectors that take this approach are the R-CNN family of detectors.

Unlike in the two-stage, detection is done directly in the **One-stage** object detection. Thus, from image to classification and localization of bounding boxes, all in one forward pass. This approach is very fast and has lead to some real breakthroughs in the application of object detection, such as real-time object detection. The most famous object detector that takes this approach is the YOLO family of detectors. This process will be described in detail in the next chapter.

1.4.2 IOU (Intersection Over Union)

Intersection Over Union, or simply IOU, is an evaluation metric used to measure the accuracy of an object detector on a particular dataset [26]. Any algorithm that predicts bounding boxes as an output can be evaluated using IOU.

To evaluate a predicted bounding box, the ground-truth bounding box is needed. A ground-truth bounding box refers to a "hand labelled" bounding box, i.e. the bounding

box manually drawn around objects of interest for the training of a neural network. Since a predicted bounding box is highly unlikely to have x, y coordinates that are exact matches to the x, y coordinates of the ground-truth bounding box, IOU is used to measure how much the predicted bounding box overlaps with the ground-truth bounding box. Computing the IOU is as follows:

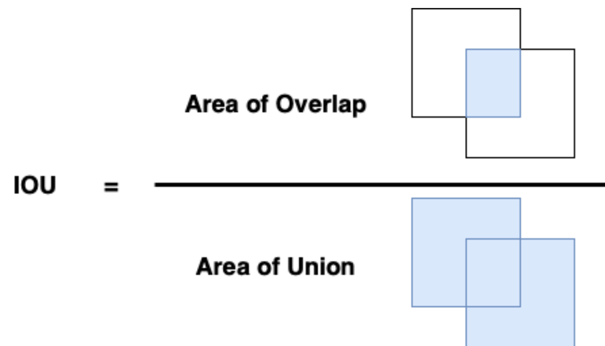


Figure 1.10: IOU calculation

This formula outputs a value between 0 and 1, the higher the output the better the predicted bounding box.

1.4.3 NMS (Non-Max Suppression)

Object detection algorithms create multiple bounding boxes per object during detection. Ideally, we want one single bounding box per object detected. Non-Max Suppression or NMS is a technique used to suppress or remove bounding boxes that are less likely to fully represent the predicted object. Fig: 1.11[23] illustrates the effects of applying NMS.

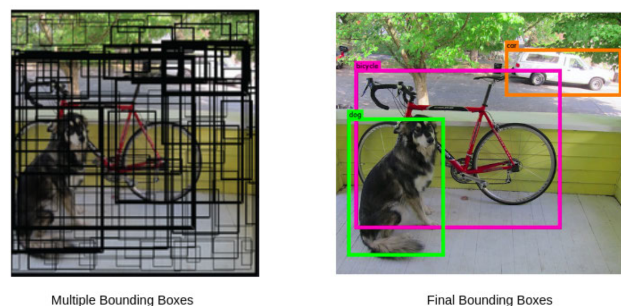


Figure 1.11: Bounding box suppression

NMS utilizes the IOU calculation and the objectness score, which is an output of the neural network representing the probability of an object being present in a particular bounding box.
The steps for selecting the best bounding box with NMS is as follows:

- Select the bounding box with highest objectness score
- Eliminate all boxes that have an IOU larger than a specific threshold (usually 0.5) with the selected box
- Repeat steps 1 and 2 until all boxes have being processed.

2 Methodology

In this chapter, we will describe the problem and the proposed solution.

2.1 Problem Formulation

Street signs are placed at the side of or above roads to provide information to pedestrians and vehicles. These street signs sometimes became unreadable due to dirt, age, accidents, etc. There have even been occasions where some German street signs became obsolete, either because they weren't relevant anymore or their look and design changed. Each state in Germany is required by law to take inventory of which Street-signs are on which roads. Most of the states outsource this task to third party companies, and this comes with a high cost.
The client for whom this project is developed is the owner of a company contracted by one of the states in Germany to regularly take inventory of the Street-signs in this state. The question asked by the client is whether there is a more cost-efficient way to accomplish this task.

This task is currently accomplished by manually taking a record of all the street signs in the state with a two-week interval and a car, a camera and at least two people are needed to do this.
The process is as follow:

- To begin, the current date and time, the starting location (street name and gps coordinate) and the driving direction of the car is recorded.
- Then one person drives while the other takes photographs of all the street-signs that come along the road.
- Additionally, the name of the street, the direction in which the car is driving and the location of the street sign is also recorded every time a street-sign is recorded.

This means that when a street-sign comes along the driver stops the car and the other passenger takes a couple of pictures and records the additional data before the journey is

continued. As you can imagine, this is a costly and time-consuming way of taking inventory of the street-signs. And so an efficient way of doing all of this is to be proposed.

2.2 Proposed Approach

The suggested solution is to use machine learning, specifically supervised learning[8], to automatically detect the street signs by installing a neural network model onto a mobile device to have access to the GPS coordinates of the device while inferring. Then finally, make use of garbage trucks since they drive through all the streets once a week by mounting the mobile phone on them.
With this in mind, several requirements have to be fulfilled for this to be possible. These requirements are categorized into two parts, the computational needs and the model-capability requirements.
The neural network used in this application should:

- be able to detect N number of different classes
- be able to detect K number of signs simultaneously
- have a low trade-off between accuracy, speed and computational cost

The mobile device housing the model has to have enough computational power for the selected model. Preferably, the mobile device has a neural engine on its SOC [4] and a dedicated GPU.

The state of the art object detection neural network YOLO was preselected to be used in this evaluation because it is one of the fastest object detectors and can infer on a dedicated server as well as a mobile device with a GPU. YOLOv3 (the current version of the network) comes in 2 varieties, the Large YOLOv3 and the Tiny YOLOv3, which is a slimmed-down version of the Large YOLOv3, only able to detect at two scales but can infer at faster speeds.

In this thesis, we look at the real-time detection of German street signs using an application on a mobile phone while driving on the street. We evaluate whether any of the models installed on a mobile device is best suited for the real-time detection of street signs.

2.2.1 YOLO (You Only Look Once)

This section will introduce and describe the architecture and the detection process of the state-of-the-art object detection neural network YOLO. We will focus on Yolov3[25], the version of the neural network used in our experiments.

YOLO is an abbreviation for You Only Look Once. Joseph Redmon first introduced this object detection neural network in 2016[23] and since then has become very popular in the world of object detection. YOLOv3 improves upon the previous version of the YOLO by featuring multi-scale detection, a larger and stronger feature extractor and changes in its loss function. These improvements enable the network to detect many objects of different sizes. YOLOv3 is a one stage object detector that can infer in real-time on GPU devices.

2.2.1.1 Model Architecture

YOLOs' network architecture can be divided into two main components: the Backbone or Feature extractor and the Detector. Images are feed into the Backbone, which extracts the features and outputs a feature map at three different scales. These features go through their corresponding scale branches of the Detector, and the final output is the bounding boxes and the class information (See Fig: 2.1).

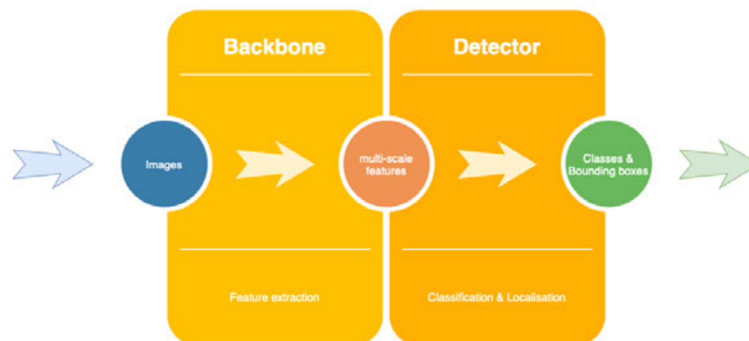


Figure 2.1: Architecture overview

Darknet-53 is the name of the feature extractor used in YOLOv3[25]. As its name implies, it is a successive 3 x 3 and 1 x 1 53 convolutional layer network. The previous version of YOLO(YOLOv2)[24] used a custom deep architecture "darknet-19", a 19-layer network extended with 11 more layers for object detection. This 30-layer architecture often struggled with small object detections due to the loss of fine-grained features because

the layers downsampled the input. To solve this, YOLOv3 adds another 53 convolutional layers to the darknet-53 network for detection purposes, and features are upsampled and concatenated with feature maps from previous layers to capture low-level features[25]. YOLOv3 boasts a total of 106 convolutional layers with skip connections and residual blocks similar to ResNet[15], which makes it a fully convolutional network (FCN). Each convolutional layer is followed by a batch normalization layer[24] and a Leaky ReLU activation [23].
In comparison to YOLOv3 (Large YOLOv3), Tiny YOLOv3[2] is based on the Darknet-19 network used in YOLOv2, and an additional five layers are added to it, making it a total of 24 layers. The main difference in the architecture between the Tiny and Large YOLOv3 is that the Tiny YOLOv3 uses a max pool layer and lacks residual blocks. The Tiny YOLOv3 detects at only two scales where as the Large YOLOv3 detects at three scales (See Fig:2.2).

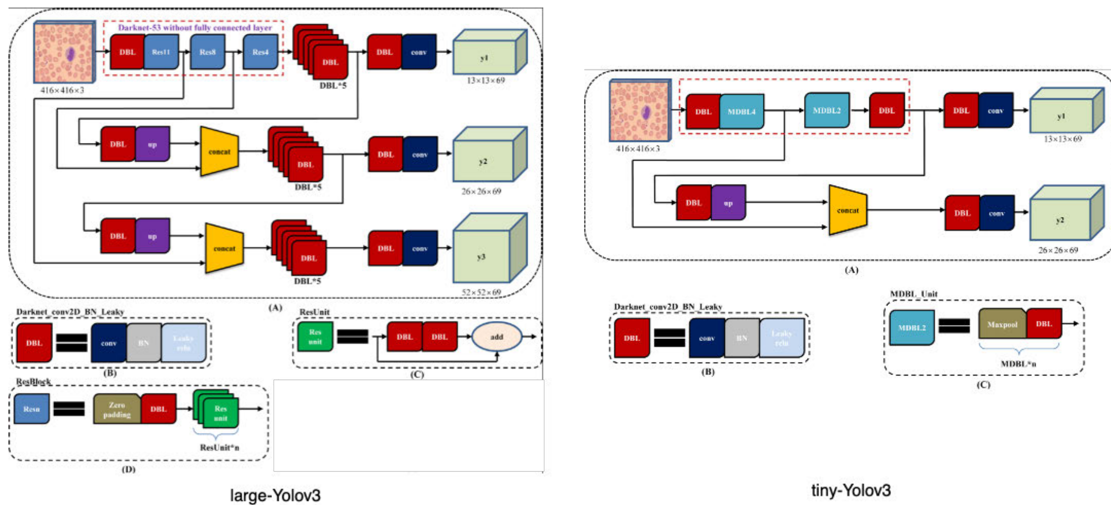


Figure 2.2: Large and Tiny Yolov3 Architecture [31]

The loss function used in the original version of YOLO[23] is depicted in Fig:2.3. In contrast to YOLOv2, YOLOv3 uses logistic regression to predict the confidence score for each bounding box. i.e. the last three terms in the Fig:2.3 have been replaced by cross-entropy error terms. YOLOv3 uses a binary cross-entropy loss for each label, which implies that the output labels are non-mutually exclusive. For example, if the output labels are "pet" and "dog", then they're non-mutually exclusive so using a sigmoid activation for each label will preserve the non-mutually exclusiveness.

In this experiment, the classes are mutually exclusive, therefore the binary cross-entropy loss is replaced with a categorical cross-entropy loss.

$$\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$
$$+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$
$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$
$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$
$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

Figure 2.3: Yolov1 loss function[23]

Before moving on to the detection process, it is necessary to understand the concept of anchor boxes or bounding box priors used in YOLOv3[25]. Anchor boxes are a set of predefined bounding boxes of a certain height and width used to capture the scale and aspect ratio of the target objects to be detect. They are chosen based on the object sizes in the training datasets.

2.2.1.2 Detection

Unlike the two-stage detectors using techniques like sliding window and Selective Search[32], YOLOv3 as a one-stage detector approaches the task of detection entirely differently. It resolves object detection as a regression problem by detecting bounding box coordinates and class probabilities directly from the image. The whole image is forwarded once through the network, and bound boxes and class probabilities are the outputs, i.e. it unifies the separate components of object detection into a single neural network [23].

First, the input image is divided into a N x N grid of cells, and each grid cell predicts B bounding boxes and their confidence or objectness score using logistic regression. As mentioned in Section 1.4.3, objectness score reflects the probability of a predicted box containing an object $Pr(Object)$ as well as the accuracy of the predicted box. Calculating the objectness score is done by measuring its IOU with the ground truth. Thus, if the bounding box prior overlaps a ground truth object by more than any other bounding box,

the objectness score is 1. Otherwise, the prediction is ignored if the bounding box prior is not the best but overlaps a ground truth object by more than some threshold (IOU threshold). YOLOv3 assigns one anchor box for each ground truth object. Anchor boxes not assigned to a ground truth object incurs no loss for coordinate or class predictions, only objectness.[25]
Each predicted box has 5 components (b_x,b_y,b_w,b_h, *objectness*). The Bounding box coordinates are given by the follow equations illustrated in Fig:2.4,

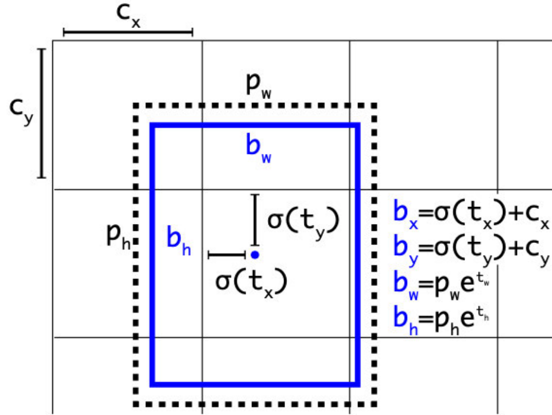


Figure 2.4: Bounding box prediction[25]

where (t_x,t_y,t_w,t_h) are the predicted coordinates by the network, (c_x, c_y) is the offset of the cell from the top left corner of the image and, (p_w, p_h) are the width and height of the anchor box [25].
So, in essence, each grid cell is assigned three (in the Large model) anchor boxes. If the centre of an object falls within the cell, then the cell is responsible for predicting the object. A grid cell can predict multiple objects by using anchor boxes if their centre falls within the cell. After the network makes the predictions, NMS (see Section:1.4.3) is used to keep only the best bounding boxes.
The predicted tensor is of a N x N x [B x 5 + C] dimension, where B is three for the large model and two for the tiny model, and in the case of this experiment, C which is the number of classes is four. YOLOv3 predicts at three different scales. The detection layer makes predictions on feature maps of three different sizes, which have a stride of 32, 16 and 8, respectively. The stride of the network is the factor by which the output layer is smaller than the input image. Given an input of 416 x 416, it makes detections at scales of 13 x 13, 26 x 26, and 52 x 52.

As illustrated in Fig:2.2, the network upsamples the input image to the first detection layer, where it makes detections using feature maps from a layer with stride 32. Further layers are upsampled by a factor of 2 and concatenated with feature maps from a previous layer with identical feature map sizes. It performs another detection on the layer with stride 16. The same upsampling procedure is repeated, and a final detection is performed on the layer with stride 8.

3 Experiments

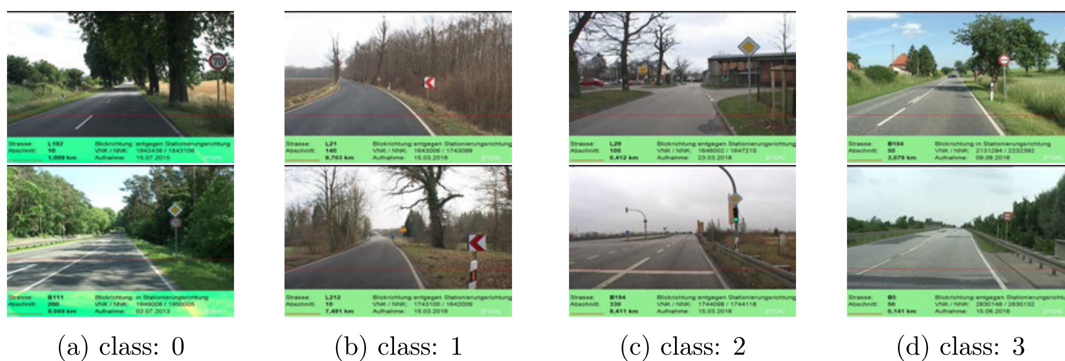
This chapter describes the preparation and preprocessing of the data, the implementation and set-up for the training and testing process, and presents and compares the results from training and testing the Tiny and Large YOLOv3 models.

3.1 Preparation

Preparing the data for the training of a neural network is a manual and tedious process. The raw dataset images are put through a selection phase. This process is done by looking at each image and selecting only those containing the relevant street signs that are clearly visible. i.e. other objects in the images do not partly obscure the target objects. The selected images are then manually annotated with bounding boxes and class labels for training.

3.1.1 Data

Over 6 million images belonging to 35 classes are available for training the models for the detection of street signs. Still, to evaluate the performance of YOLOv3 models on a mobile device, images from four preselected classes are used. Working with a smaller dataset helps to reduce the time needed for developing and testing the initial prototype.



(a) class: 0 (b) class: 1 (c) class: 2 (d) class: 3

Figure 3.1: Class image samples

This section describes the dataset from the four classes used to train and validate the proposed models.

The raw dataset consists of over 4000 non-annotated images of varying quality. The final dataset consists of 2728 fully annotated images containing 3442 target objects and their corresponding bounding boxes distributed over four classes. All the images have a dimension of 720 x 576 and vary in lighting and weather conditions. Samples of images in the final dataset is shown in Fig:3.1.

$$Distribution(C) = \frac{N_c}{N}$$

Figure 3.2: Class Distribution Formula

Ideally, each class should have a thousand or more samples for training. However, this is not the case for this dataset. The distribution of the classes is imbalanced and, therefore, might lead to the trained models having a predisposition towards the majority class and being less accurate when predicting the minority class. With the formula in Fig:3.2, where N is the total number of class samples in the dataset, and N_c is the total number of samples in the class C, the distribution for each class is calculated and shown in Table:3.1.

Label	Class	Samples	Distribution
156	0	1107	32.16%
631	1	1123	32.63%
210	2	466	13.54%
171	3	746	21.67%

Table 3.1: Class Sample Distribution

3.1.2 Annotation Process

The annotation process is done by manually drawing a bounding box around each object of interest in every image in the dataset. The corresponding class label for the object of interest is also recorded. For each image file in the dataset, a text file with the same name is created and the centre point (x, y) coordinates, the width and height of the drawn bounding boxes from the centre point as well as the class is saved in file. Each image can have multiple bounding boxes. Each of the bounding box coordinates belonging to the

image is written on a new line its corresponding text file. The recorded bounding box data is as follows: x,y w,h c, where x,y is the centre point of the box, w,h is the width and height from the centre point and c is the class of the target object.
After drawing and recording the bounding boxes, their coordinates are converted into the final annotation format used to train the model. The final output is one text file with all the images and their bounding box coordinates separated on each line. Figure:3.3 shows a sample section from the final annotation file.

```
4   image4.jpg 483,251,520,300,1 299,246,322,276,1 309,205,345,245,2
5   image5.jpg 500,256,536,299,1 249,245,266,265,1 149,241,163,258,1 159,219,181,245,2
6   image6.jpg 601,252,671,333,1 437,254,465,286,1 237,246,253,265,1 143,245,157,259,1 156,225,173,247,2
7   image7.jpg 535,250,588,312,1 422,253,448,282,1 238,248,253,266,1 145,243,159,260,1 158,227,176,247,2
8   image8.jpg 580,255,628,315,1
9   image9.jpg 550,255,604,318,1 323,249,343,275,1
```

Figure 3.3: Annotation file sample

$$Pathtoimage\ x1,y1,x2,y2,classlabel\ x1,y1,x2,y2,classlabel\ ...$$

A single white space separates the annotation data information for an image. In this case, the first part is the relative path to the image file, and the second is the bounding box coordinate and the class. If there are multiple bounding boxes in an image, the bounding boxes are separated by white space. For the information representing the bounding box coordinate, $x1$ and $x2$ are the leftmost and rightmost pixel values of the box and $y1$ and $y2$ are the uppermost and lowermost pixel values of the box.

3.2 Preprocessing

This section describes the preprocessing needed for training the models and the changes made with the dataset to improve their performance.

YOLOv3 [25] uses a predetermined set of boxes called anchor boxes to help predict bounding boxes. Thus, instead of directly predicting a bounding box, the model predicts offsets for a predetermined set of boxes with particular aspect ratios. Meaning, the network should not predict the final size of the object but should only adjust the size of the nearest anchor box to the size of the object.
From the annotated dataset, custom anchor boxes are generated for training and testing the models. Since there is an imbalance in the distribution of the classes, there will be some bias in the model. The bias introduced by the imbalance in class samples is handled with the help of a Class-Balanced loss and Class sample weights. Finally, the selected augmentations used to handle the problem of insufficient data is presented.

3.2.1 Custom Anchors

K-means clustering is used to generate the custom anchor boxes for the YOLOv3 models. The clustering is done across all ground truth anchor boxes from the final annotation file. This process finds the most common bounding boxes aspect ratios that best encompass the targets in the dataset.
To create the custom anchors for the dataset with K-means, the number of clusters must be manually specified. The Large YOLOv3 model uses nine anchors, and the Tiny YOLOv3 model uses six anchors. The number of clusters chosen corresponds to the number of anchor boxes used by each model. The initial centroid bounding box is randomly chosen, but the metric used to measure the distance is as shown in Fig:3.4.

$$d(box, centroid) = 1 - IOU(box, centroid)$$

Figure 3.4: K-means Distance Metric [24]

Table:3.2 shows the custom anchors generated by the K-means clustering process on the dataset:

Model	Anchors
Tiny	$(15, 18), (29, 34), (38, 43), (46, 51), (55, 60), (68, 75)$
Large	$(13, 16), (17, 21), (24, 28), (31, 35), (36, 42), (42, 47), (49, 54), (58, 64), (72, 80)$

Table 3.2: Custom Anchors

3.2.2 Improving Class Imbalance

Class imbalance is a problem that occurs in machine learning classification problems, in which the occurrence of one or more of the classes is very high or very low compared to the other classes present in the dataset, i.e. there is a skewness towards the majority class.
There are many ways to tackle the problem of imbalanced class distribution in datasets. As it is a common and well-known problem, there are many different solutions for it. These solutions usually take one of two approaches:

- To directly increase or decrease the samples of the minority or majority class

- To handle the imbalance in the loss function

Some of the popular techniques used in the first approach include,

- undersampling, which in essence removes samples from the majority class,
- oversampling does the opposite by adding samples to the minority class, though this could lead to overfitting [34], if not done carefully
- and synthetic minority oversampling technique(SMOTE)[5], which is a way of oversampling by generating synthetic data for the minority class.

In the second approach, the standard technique is sample weighting which means, weighting the class samples in the loss function to yield a class-balanced loss. Simply put, a weight is added to each sample in the dataset [7].

To prevent the YOLOv3 models from being biased toward the dominant class in the dataset, the second approach is chosen, and Class weights are introduced to make up for the imbalanced data. In essence, we want the model to be aware of the imbalance in the class distribution and consider that when calculating the loss.

$$W = \frac{\max(\{|X|, X \in C\})}{|X| \forall X \in C}$$

Figure 3.5: Class Weights Formula

In our dataset, for every sample of class 2, there are approximately 2.4 samples of class 1, and for every sample of class 3, there are approximately 1.5 samples of class 1, see Table: 3.3. With the simple formula Fig: 3.5, where C is the set of classes in our dataset, the weights for each class is calculated and used in the training of the models. By applying this strategy, the cost of misclassifying class 2 will be 2.4 times higher than the cost of misclassifying class 1. This way, the under-represented class is taking into account when learning, i.e. classes with fewer samples have a higher class weight, and the model gets a higher penalty for misclassifying those samples. The higher the penalty for a sample, the more the model is going to try to include this sample within its decision boundary.

Class	Samples	Weights
0	1107	1.0145
1	1123	1.0000
2	466	2.4099
3	746	1.5054

Table 3.3: Class Weights

3.2.3 Image Augmentation

Deep neural networks need large amounts of training data to achieve good performance, and limited data is a big problem in applying deep learning models like convolutional neural networks. If the model learns from only a few examples of a given class, it is less likely to predict the class in validation and real-world applications correctly.
Image augmentation is the duplication of images with some variation to enable the model to learn from more examples and better generalize. Augmentations such as random rotation, shifting, shearing, scaling, flipping, translation and distortion can be applied or even combined depending on the use cases. When making these variations, it is vital to ensure that the key features for making predictions are preserved. Otherwise, it will be counter-productive if the augmented images are very dissimilar to those used to test the model. [30]

We carefully analyzed the effects of several augmentations on the images in our dataset. We selected those that did not drastically alter the images and compromise the critical features needed for prediction. Since most of the target objects in our dataset images are located close to the rightmost or leftmost edge of the images frame, position augmentations cannot be done in high magnitudes; otherwise, our targets will shift out of the image frame.
We apply five augmentations randomly at a random magnitude:

- Rotation of the input images by a random value between -10 and 10 degrees. (See Fig: 3.7)
- Scaling of the input images to a value of 90 to 110% on x- and y-axis independently. (See Fig: 3.8)

- Shearing of the input images by a random shear value between -10 and 10 on x- and y-axis independently. (See Fig: 3.9)
- Translating of the input images by -10 to +10% on x- and y-axis independently. (See Fig: 3.10)
- Distortion is finally applied to the augmented image by adding a random Hue between -10% and +10% and a random Saturation and Intensity between 0% and 50%. (See Fig:3.6)

Figure 3.6: Distortion

$$Rotate(\theta) : (x, y) \to (x\ cos(\theta) + y\ sin(\theta), -x\ sin(\theta) + y\ cos(\theta))$$

Figure 3.7: Rotate

For (x, y), being pixel Cartesian coordinates, the augmentations are applied to each image and target bounding-box before the model sees it.

$$Scale(a, b) : (x, y) \rightarrow (ax, by)$$

Figure 3.8: Scale

$$Shear(a, b) : (x, y) \rightarrow (x + ay, y + bx)$$

Figure 3.9: Shear

$$Translate(a, b) : (x, y) \rightarrow (x + a, y + b)$$

Figure 3.10: Translate

3.3 Experimental Setup

This section describes the implementation, training, and testing setup for the tiny and large YOLOv3 models.

3.3.1 Implementation

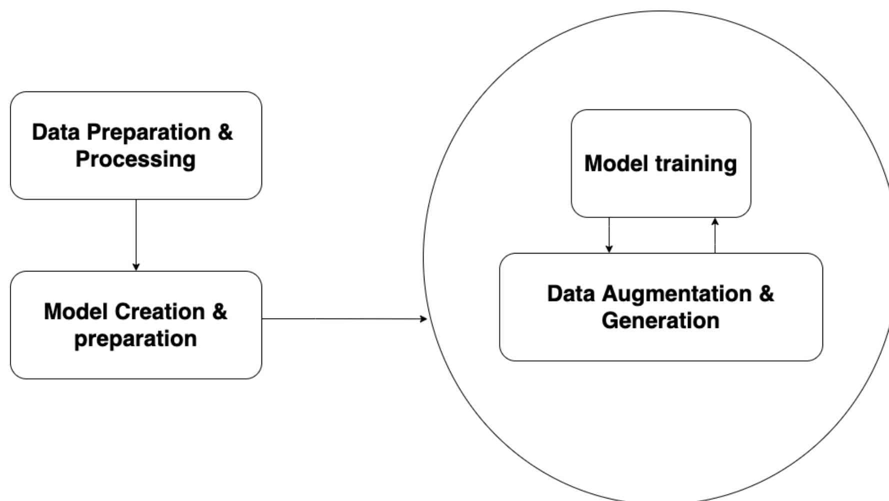


Figure 3.11: Implementation Requirements

The implementation of sampling the data and training the model is divided into simple requirements as shown in Fig:3.11. The four primary responsibilities (Data preparation, Data augmentation, Model creation and Model training) are separated into five object classes representing the different abstraction layers in our system. These classes are DataProcessor, TrainGenerator, ValidGenerator ModelGenerator and Trainer, and they follow the single responsibility principle [19], which leads to them having high cohesion but low coupling [33]. The object oriented approach to the implementation, keeping the class functions small and the code clean, improves readability, scalability and testability.

```
73    def _train_model(self, model, train_generator, valid_generator, train_steps, valid_steps):
74        log = TensorBoard(log_dir=LOG_DIR)
75        checkpoint = ModelCheckpoint(
76            os.path.join(
77                LOG_DIR, "ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5"
78            ),
79            monitor="val_loss", save_weights_only=True, save_best_only=True, mode="min",
80            save_freq="epoch"
81        )
82        model.fit(
83            x=train_generator,
84            steps_per_epoch=max(1, train_steps),
85            validation_data=valid_generator,
86            validation_steps=max(1, valid_steps),
87            max_queue_size=100,
88            epochs=self._epochs,
89            callbacks=[log, checkpoint],
90            use_multiprocessing=self._use_multiprocessing,
91            workers=self._num_workers
92        )
```

Figure 3.12: Model training

The Trainer class is ultimately responsible for the whole process of training the models and has a composition type of association with the other classes, precisely an aggregation type [33]. It instantiates the DataProcessor object, which is responsible for most of the file operations such as reading and preparing the class labels and the anchors, reading the images and their targets into NumPy [14] arrays in unit8 datatype used to represent pixel values, calculating the class weights in our dataset and splitting the data into the train and validation dataset used to fit and evaluate the model.
The ModelGenerator is used to create a Tiny or Large YOLOv3 model, depending on the number of anchors and compile it with the adam optimizer, discussed in section 1.2 and an initial learning rate. The train and validation dataset is used to instantiate TrainGenerator and ValidGenerator objects responsible for feeding the model with batches of augmented and non-augmented image data during training and validation. It is essential that the validation data is not augmented so that the evaluation of the progress of the models during training is as close to the real-world application as possible. The ModelGenerator is also responsible for loading the pre-trained model weights for transfer learning [12] and adding the class weights to the classification part of the models' loss function.
The Sequence diagram in Fig: 3.13 illustrates the steps performed to prepare and train the models.
Finally, the Trainer then creates checkpoints for saving the trained weights while training, activates logging of the training and validation loss and starts the training process in only a few lines of code as shown in Fig: 3.12.

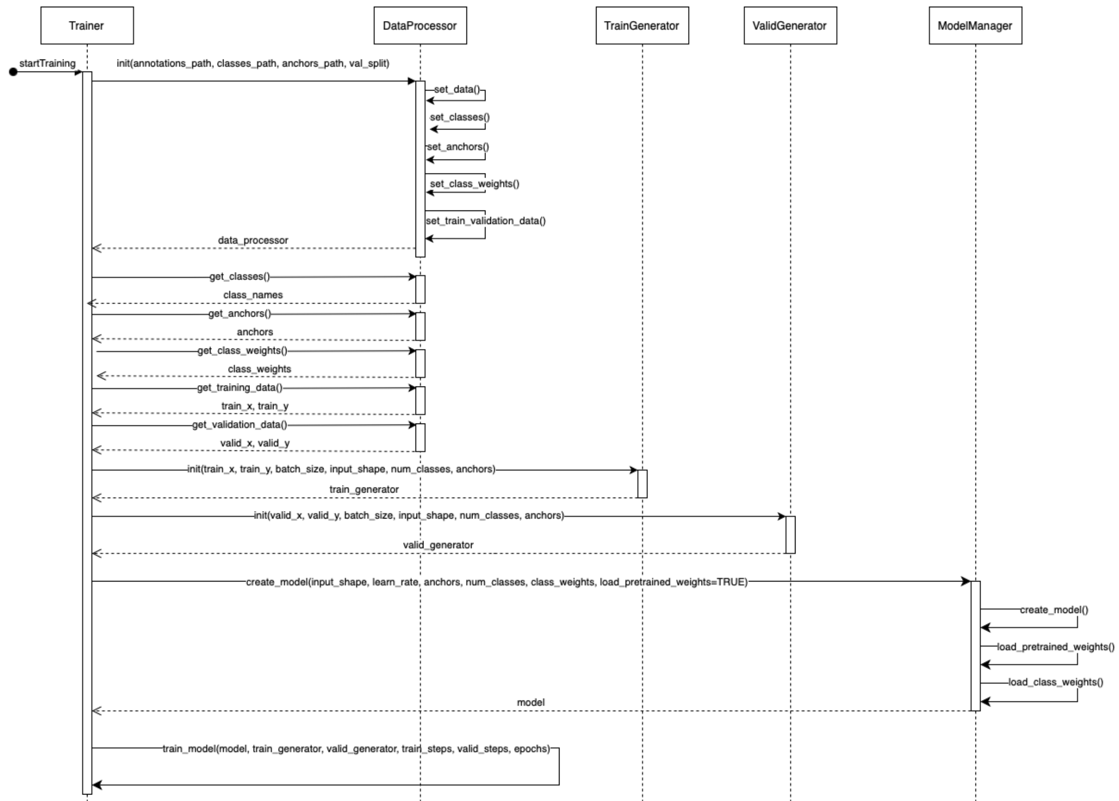


Figure 3.13: Training Sequence diagram

The Keras [6] and Tensorflow [1] frameworks provide API functions for machine learning projects and were used in the implementation.

3.3.2 Training

Transfer learning is a widespread technique in deep learning, in which knowledge learned by a network in one task can be transferred and used as a foundation for learning in another task [12], especially in object detection or image classification tasks. For example, if task A is to detect bicycles and task B is to detect motorcycles, then the knowledge acquired in task A can be transferred to task B since their respective targets share standard features which need to be learned in both tasks. The domain or context from which the features are transferred has to be general enough to improve the training of the second task.

YOLOv3 was trained on the COCO Dataset[18], which is a publicly available dataset with 80 classes. COCO stands for Common Objects in Context, and the training dataset was made available by Microsoft. The features learned from the COCO Dataset training are transferred and used as the foundation for the training in this experiment. The pre-trained weights are publicly available on the creator and author of YOLO, Joseph Chet Redmons' website.
Training of the yolov3 network is divided into two stages.
In **stage 1**, the network is fine-tuned with the transferred weights. The earlier layers of the model trained on the COCO Dataset contain more generic and useful features for this task but not the later layers. The pre-trained weights are loaded into the model, and the earlier layers of the network are not trained (frozen) but only the output layers. Freezing the earlier layers helps the model converge faster with little data, stabilizes the loss and prevents overfitting [12]. The selected batch size is the maximum possible batch size for both models due to hardware limitations. Table: 3.4 shows the hyperparameters used in this stage and the duration of the training.

Model	Learning Rate	Batch	IOU threshold	Epochs	Duration
tiny	1e−3	32	0.7	50	3.5 days
large	1e−3	16	0.5	50	4.5 days

Table 3.4: Training Stage 1

In **stage 2**, all the layers of the network are trained. By unfreezing all the layers to be trained, most of the GPU ram available on the training hardware is occupied by the model to be trained, so in stage 2, the batch size is further limited. This is especially the case for the Large model. Table: 3.4 shows the hyperparameters used in this stage and the duration of the training.

Model	Learning Rate	Batch	IOU threshold	Epochs	Duration
tiny	1e−4	24	0.7	250	19.5 days
large	1e−4	8	0.5	250	21 days

Table 3.5: Training Stage 2

During training, the TrainGenerator delivers a batch of augmented images data to the model on every step in an epoch, resized to the shape 416 x 416, which is the input shape of the model. The model weights are also saved after every epoch if they are better than the previously saved ones.

For evaluating the model's performance during training, the primary dataset is split into train datasets used to fit the model and validation datasets used to evaluate the fit model. A train-validation split of 80% and 20% is used in this experiment.
Training is done on an ASUS ROG Zephyrus S Laptop with a dedicated Cuda [20] enabled GPU. The technical specification of the laptop is shown in Table3.6 below.

Device	ASUS ROG Zephyrus S
Processor	3.9 GHz Intel core i7
RAM	16 GB DDR4
Memory Speed	2666 MHz
Hard Drive	512 GB Hybrid Drive
Graphics Coprocessor	NVIDIA GeForce GTX 1060
Graphics Card Ram Size	6 GB
Operating System	Windows 10 pro

Table 3.6: Training hardware specifications

A **Learning Curves** is a plot that can be used to tell how well a model learns over time. Specifically, it shows the changes in learning performance over time. i.e. the time is plotted on the x-axis and the learning improvement on the y-axis. Learning curves or LCs are deemed practical tools for monitoring the performance of workers exposed to a new task. LCs provide a mathematical representation of the learning process that takes place as task repetition occurs [3].
The metric used to evaluate learning can be maximizing, meaning the higher the number, the better the model performs, or minimizes the exact opposite, and smaller numbers indicate better performance. Examples of these are; classification accuracy, a maximizing metric, and loss or error, which is a minimizing metric.
In this experiment, the loss is used to create learning curves for the models during training. The Learning curves calculated from the training dataset gives an idea of how well the model is learning, and those from the validation dataset gives an idea of how well the model is generalizing.
The learning curves to diagnose problems with learning, i.e. whether the model is underfitting [12] or overfitting [28].

3.3.3 Testing

Testing the trained models is done on a mobile device, specifically the Apple iPhone 11 *pro*. The Large and Tiny model is installed onto the mobile phone, and detection is done

from within a car. Even though the mobile device used for testing has Optical Image Stabilization (OIS) capability, it is kept on a mount in the car to keep the stream of frames from the camera as steady as possible. Both the Tiny- and Large-model is tested with the configurations shown in Table3.7 below.
Several considerations are taking into account during the evaluation of the models:

- The amount of space the model takes up in the application bundle (how much memory does it add to the download size of the application).
- The amount of memory the model takes up at runtime (how much RAM is consumed by the GPU for inference)
- How fast the model runs (how long does the model take to process a single image)
- How quickly the model drains the battery or makes the device too hot to hold.

A route with the street signs the model was trained on is chosen, and each model is tested from starting point A to destination B without interruption individually. The testing of both models took 6 hours in total, excluding the time it took to drive back to the starting point A.

Model	IOU threshold	Confidence threshold	Driving speed
tiny	0.5	0.5	40 - 70 kmh
large	0.5	0.5	40 - 70 kmh

Table 3.7: Test Configuration

The technical specifications of the testing hardware are shown in Table:3.8 below.

Device	Apple iphone 11 pro
Chipset	Apple A13 Bionic (7 nm+)
CPU	Hexa-core (2x 2.65GHz + 4x 1.8GHz)
GPU	Apple GPU (4-core graphics)
Memory	64GB, 4GB RAM, NVMe
Operating System	IOS 14.3

Table 3.8: Testing hardware specifications

The metric used to evaluate the trained tiny- and large-yolov3 model is the Mean Average Precision at an IOU of 0.5(**mAP**@0.5). The mAP metric is the product of **precision**

and **recall** of the detected bounding boxes[29]. The mAP values range from 0 to 1, and the higher the value, the better the model is performing. The mAP is computed by calculating the average precision (AP) separately for each class, then dividing their sum by the number of classes[21]. Average precision or AP computes the average precision value for recall value over 0 to 1, i.e. the area under the Precision-Recall curve. Precision-Recall (PR) curves are often used in Information Retrieval and are optimal for tasks with a large skew in the class distribution [9].

The precision measures how accurate the predictions are (i.e. the percentage of the correct predictions) and is defined as

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}.$$

The recall measures how well the model finds all the positives and is defined as

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives},$$

where **True Positives** is the number of detections with an IOU larger than 0.5 with the ground truth, **False Positives** is the number of detections with an IOU less than 0.5 with the ground truth or with duplicate Bounding Boxes and **False Negatives** is the number of objects that were not detected or detected with an IOU larger than 0.5 but with the wrong classification.
Mean average precision is defined as

$$mAP = \frac{1}{C}\sum_{i=1}^{C} AP_i,$$

where AP_i is the average precision value for the i-th class and C is the total number of classes[21].

3.4 Experimental Results and Comparison

This chapter presents the experimental results. First, the results from training and the convergence of the training- and validation-loss of the tiny- and large-Yolov3 models, then their learning curve is evaluated. After that, the results from testing them on the mobile device are shown and compared.

3.4.1 Training

The loss from the training of the Tiny- and Large-model in **Stage 1** is shown in Table:3.9. In the first epoch, the models started with a large training loss that was rapidly reduced in the validation. The difference in the loss between the train and the validation of the first epoch was 70.69 for the Tiny model and 86.3 for the Large model. Meaning both models were able to significantly reduce the loss after this epoch by capturing the features in the earlier layers of the network needed to classify the targets in our dataset. After the final epoch of stage 1, the Tiny- and the Large-model minimized their validation loss by 8.67 and 79.27. This concluded the stage with a validation loss of 10.89 and 20.97, and a training loss of 11.82 and 22.38 for the Tiny- and Large-model, respectively. The achievement of such a loss in only 50 epochs of training can be attributed to the use of the transfer-learning technique (see Section:3.3.2). As evident in the loss values achieved by each model, the Tiny model was able to reach a smaller loss than the Large model as it has fewer layers and therefore fewer weights to update. i.e. the features learned do not have to be distributed across many weights as in the Large model.

	Epoch 1		**Epoch 50**	
Model	Train	Validation	Train	Validation
Tiny	90.25	19.56	11.82	10.89
Large	186.54	100.24	22.38	20.97

Table 3.9: Stage 1 loss

	Epoch 51		**Epoch 250**	
Model	Train	Validation	Train	Validation
Tiny	6.79	4.57	2.04	2.44
Large	13.83	12.75	3.21	2.90

Table 3.10: Stage 2 loss

In **Stage 2**, the final validation loss achieved was 2.44 by the tiny-model and 2.90 by the large-model. The difference between the loss of the $51th$ and that of the $250th$ epoch was very minimal for both the train and validation (see Table:3.10). This behaviour is not surprising, i.e. since the layers are all unfrozen, many forward passes and therefore backpropagation's are needed to trigger a noticeable change in the output.

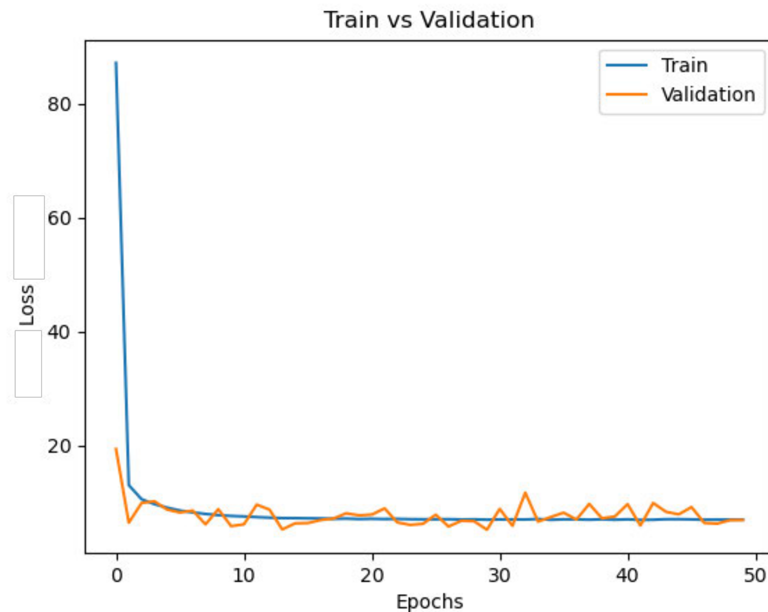


Figure 3.14: **Stage 1**: Tiny Yolov3 Learning curve

Fig:3.14 shows the learning curve of the Tiny model from **Stage 1** of the training. Despite the jumps in the validation learning curve, both the training and the validation loss show a good fit learning curve which is the goal of the learning algorithm. A good fit learning curve exists between an overfit and underfit model and is identified by the training and validation loss decreasing to the point of stability with a minimal gap between the two final loss values. The difference between the final validation and training loss values is 0.93 for the tiny model and $1,41$ for the large model.

In **Stage 2** the learning curve for the training loss looks like a good fit, but the learning curve for the validation loss shows a lot of noisy movements around the training loss (see Fig:3.16 Left: Tiny Model, Right: Large Model). The noisy movement in the validation learn curve shows that the validation dataset is unrepresentative, i.e. the validation dataset does not provide sufficient information to evaluate the ability of the model to generalize. This sometimes occurs if the validation dataset has too few samples compared to the training dataset, as is the case in this experiment. The learning curve of the validation loss of the tiny model is slightly less volatile than that in the validation loss of the large model. Due to the size of the Tiny model, the impact of the insufficient validation data is not as high as in the case of the large model. The difference between

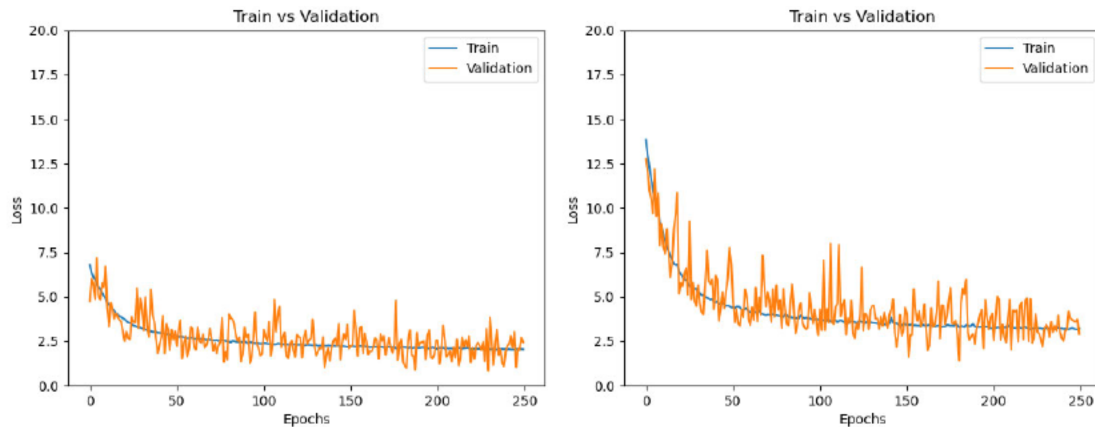


Figure 3.16: **Stage 2**: Tiny and Large Yolov3 Learning curve

the validation and training loss values is 0.40 for the Tiny model and 0.31 for the Large model.
At the end of the training, the Tiny model converges faster than the large model and achieves a lesser loss with a difference of 0.46.

3.4.2 Testing

Model	Frames per second	Processing Time
Tiny	~ 30	0.02 seconds
Large	~ 19	0.62 seconds

Table 3.11: Processing results on device

At the rate of $\sim$ 19 frames per second, the Large model took an average time of 0.62 seconds to process each frame as shown in Table:3.11. Thus by the time the bounding box is drawn, the target street sign is no longer located in the original pixel coordinates when the model began processing the image. This caused all of the drawn bounding not to encompass any of the street signs that were to be detected (see Fig:3.20), meaning all the detections were False Negatives (see Section:3.3.3).
In contrast, the Tiny model took an average time of 0.02 seconds to process each frame at the rate of $\sim$ 30 frames per second (see Table:3.11). Due to the short processing time, the model was able to correctly detect and encompass almost all of the target street signs (see Fig:3.22), except for very few. Figure:3.24 shows examples of the bad detections of

the tiny model, in which the target objects where detected multiple times within a frame making them False positive detections(see Section:3.3.3).

The observations made while testing the models show that:

- The Tiny model added 34.7MB and the large model added 246.4MB to the total size of the application.
- At runtime the tiny model took up a memory size of 6MB and the large a size of 21MB.
- The Large model took an average time of 0.60 seconds longer to process each frame than the Tiny model, which significantly impacted its performance.
- The mobile device stayed cool to the touch and the Tiny model drained the battery capacity by 15% and the Large model by 35%.

	Class 0			**Class 1**			**Class 2**			**Class 3**		
Model	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Tiny	35	43	12	83	39	111	39	61	37	19	15	29
Large	0	96	47	1	103	193	0	82	76	0	49	48

Table 3.12: Class Detections

In Table:3.12 above, where TP is the number of True Positive detections, FP is the number of False Positive detections, and FN is the number of False Negative detections, the performance difference between the Large and Tiny model is illustrated. The Large model made only one True Positive (TP) detection for Class 1 and otherwise no other True Positive detections for any other Classes. All other detections were either False Positive or False negative ones, in that they were detections with an IOU less than 0.5 with the ground truth or the target objects were not detected at all. On the other hand, the Tiny model was able to detect at least one-third of all targets in all of the Classes. The False Positive detections made by the Tiny model were mostly detections with duplicate Bounding Boxes or with the wrong classifications (see Fig:3.24).

The results from the mAP@0.5 evaluation also reflected the performance difference between the Tiny- and Large-model.

Figure:3.17 shows the precision-recall curves and the average precisions for the Large models' detection in each class. The Large model had one True Positive detection in one class and none in the other three. It, therefore, incurred an average precision of zero for all four classes.

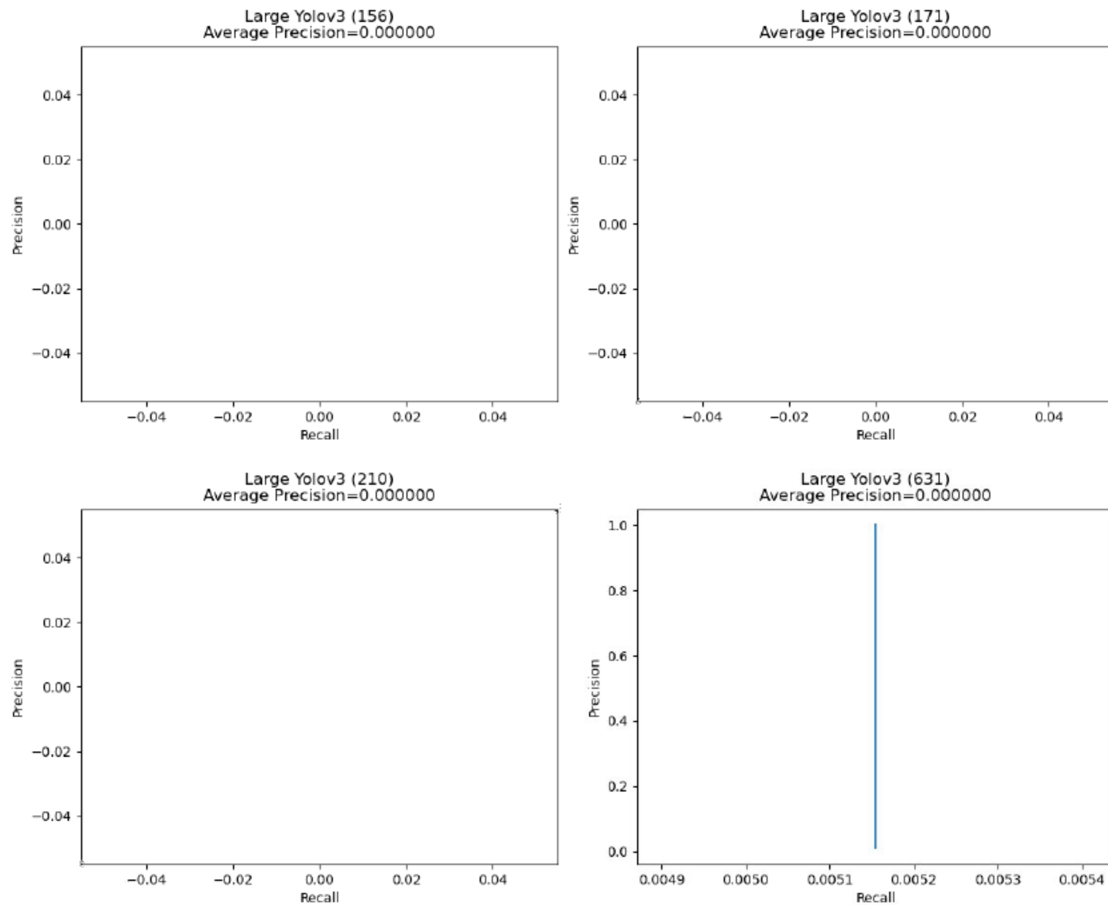


Figure 3.17: Large YOLOv3 Precision-Recall curves

Figure:3.18 shows the precision-recall curves and the average precisions for the Tiny models' detection in each class. The Tiny model achieved an AP of 0.46 in class 0, 0.33 in class 1, 0.22 in class 2, and 0.25 in class 3. The results reflect the imbalance in distribution, in that the classes with a lower sample distribution achieved a lower average precision, and those with a higher distribution achieved a higher average precision. Nonetheless, the Tiny model did not achieve an AP above 0.5 in any of the classes.

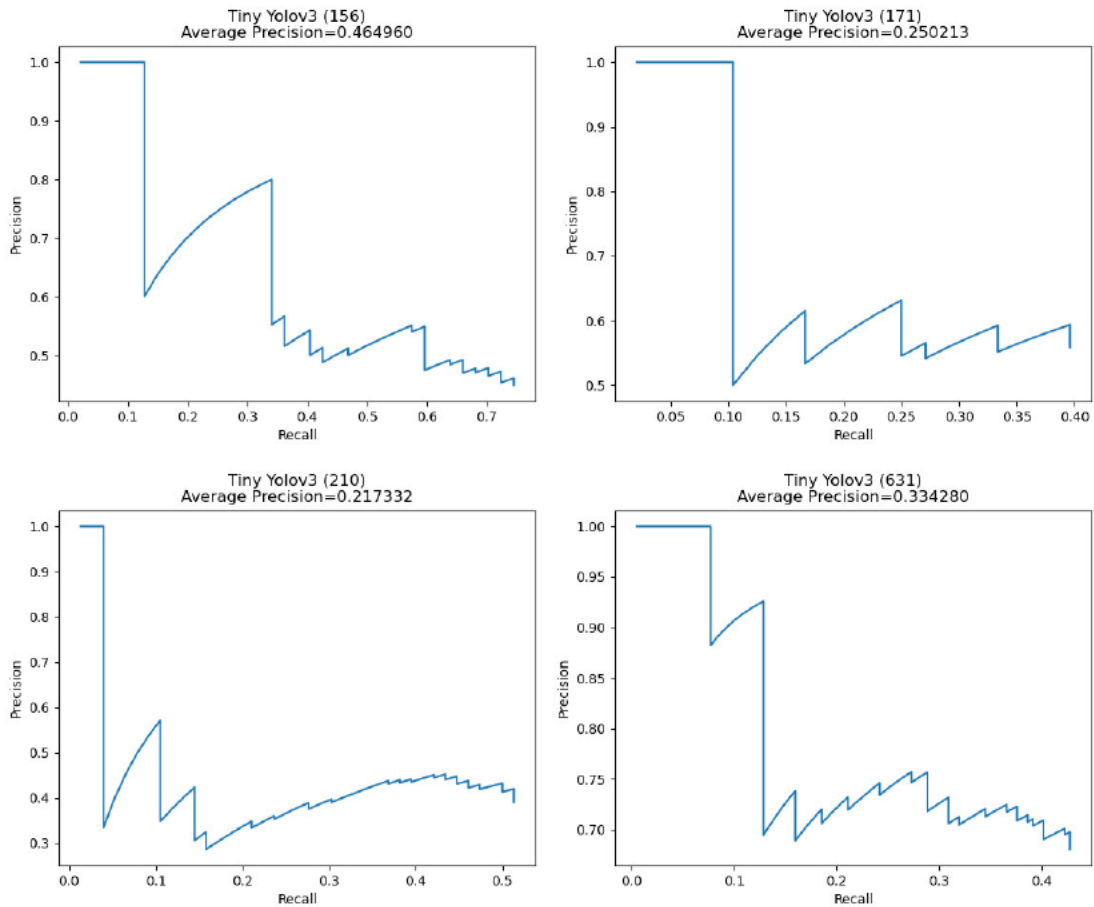


Figure 3.18: Tiny YOLOv3 Precision-Recall curves

Table:3.13 shows the mean average precision achieved by both models. The Tiny model has a better mAP result than the Large model, which achieves the lowest possible value for the mean average precision.

Model	mAP
Tiny	0.316696
Large	0

Table 3.13: Processing results on device

Apart from the performance difference between the Tiny and Large models, it is notable that none of the bounding boxes from the Large model were duplicates, i.e. multiple bounding boxes were not generated to encompass the same object. However, the tiny

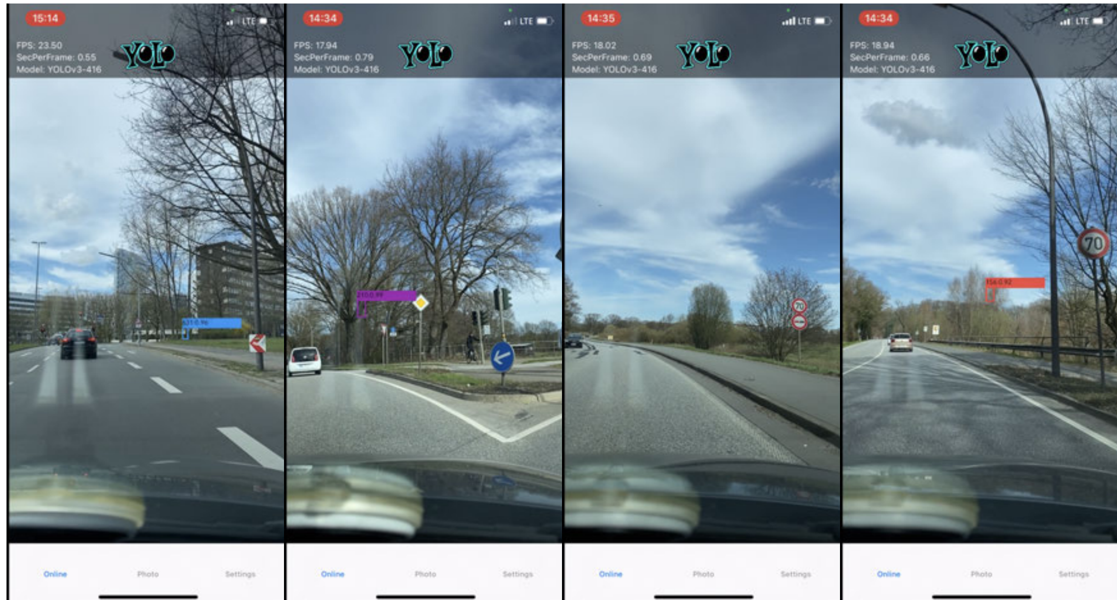


Figure 3.20: Large YOLOv3 Detections

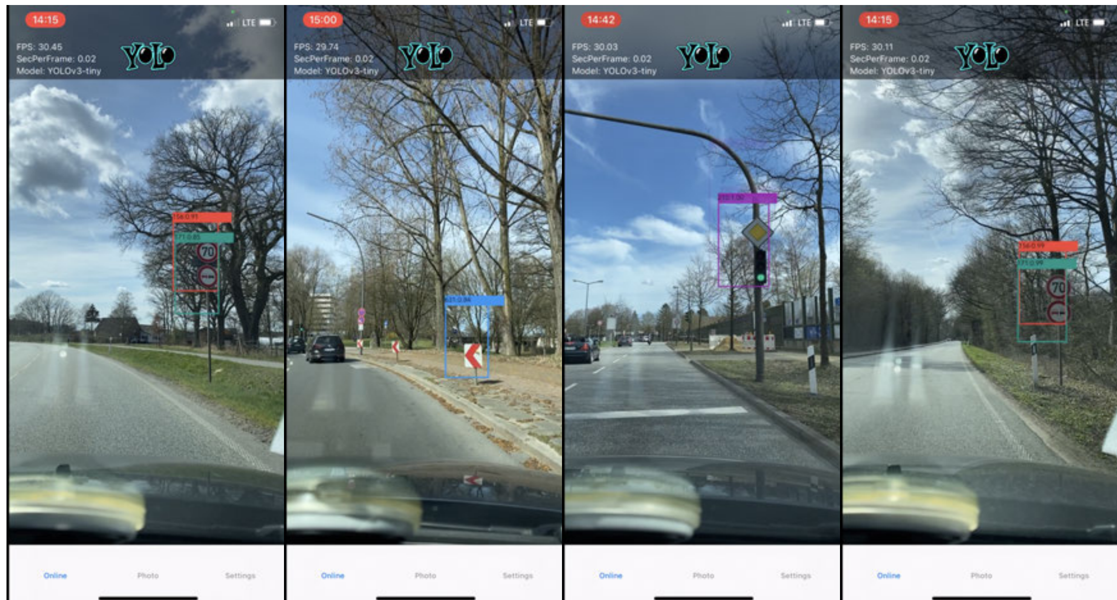


Figure 3.22: Tiny YOLOv3 Good-Detections

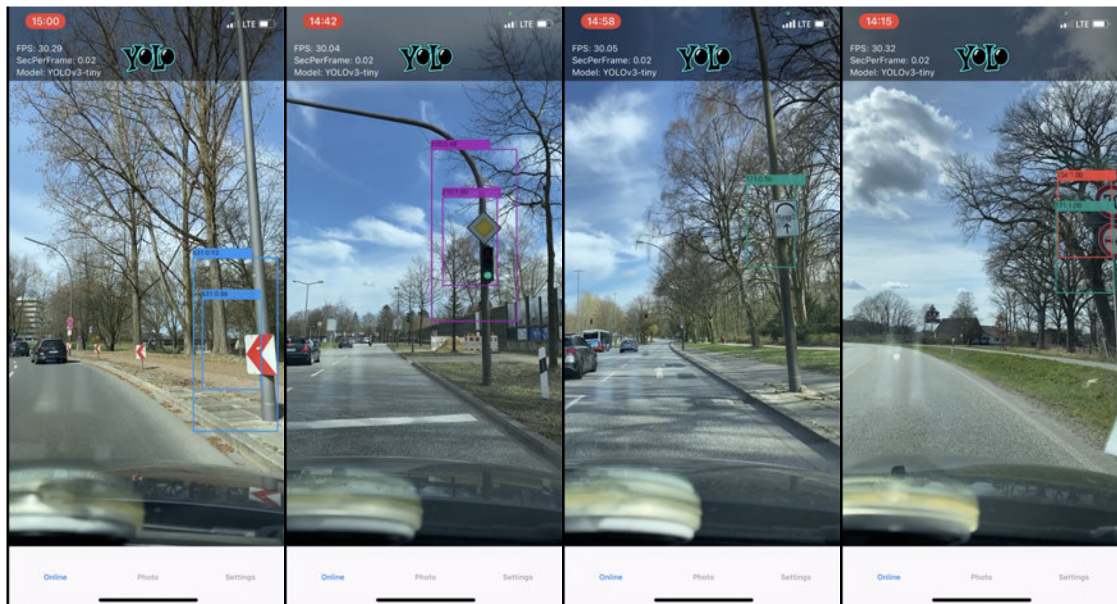


Figure 3.24: Tiny YOLOv3 Bad-Detections

model had many cases where multiple boxes were generated for the same target object. This could be because the tiny model can only detect at two scales, or the IOU threshold selected for detection was too low for the Tiny model to ensure that NMS was effective.

4 Conclusion

The objective of this thesis was, firstly, the research and implementation of the training of state-of-the-art object detection neural network YOLO and, secondly, the evaluation of the two versions of this neural network, whether any of them can sufficiently and reliably detect street signs from a mobile device at a fast and accurate enough rate when driving on the road.

With these objectives in mind, we showed that the implementation of the training process of the neural network could be made very modular with an object-oriented approach to enable different models to be trained with the exact same implementation and only switch out the models. We also discovered how important the training data is to the training of a neural network, in that much data is needed, and the distribution of the data should be ideally equal in all classes. Using the class-balanced loss approach to handle the imbalance in the class sample distribution did not yield significant results; in that, the skewness towards the majority class was still noticeable during the testing of the models.

Comparing the results from testing the Tiny and Large YOLOv3 models, we discovered that the Large model is not ideal for inference on a mobile device in real-time as it scored an mAP of zero. Meaning it did not successfully detect any of the target objects in real-time. The Large model's poor performance was primarily due to needing 60 seconds to process each frame on the mobile device. Therefore it could not accurately capture any of the objects in real-time. On the other hand, the Tiny model was able to detect most of the target objects in real-time because it needed only 0.02 seconds to process each frame. The size of the Tiny model and the speed at which it can infer makes it the better option between the two for real-time detection on a mobile device. However, the Tiny model scored an mAP of 0.316696, which concludes that it is generally not sufficiently reliable for real-time detection on a mobile device while driving. Since most of the False detection of the tiny model was of ones with multiple bounding boxes for the same target and those of the Large model was caused by processing speed, we believe that the tiny model is generally inferior.

So the answer to the question whether any of the models can detect street signs in real-time on a mobile device while on the road is; yes, the Tiny model can do this but not at a sufficiently reliable rate.
An alternative approach to solving real-time detection of street signs with the YOLOv3 model will be to use a laptop such as the device used for training and a web camera. With better hardware, we could install the Large model on it and infer real-time with better accuracy and reliability.

Bibliography

[1] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dan ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. – URL http://tensorflow.org/. – Software available from tensorflow.org

[2] Adarsh, P. ; Rathi, P. ; Kumar, M.: YOLO v3-Tiny: Object Detection and Recognition using one stage improved model. In: *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, S. 687–694

[3] Anzanello, Michel J. ; Fogliatto, Flavio S.: Learning curve models and applications: Literature review and research directions. In: *International Journal of Industrial Ergonomics* 41 (2011), Nr. 5, S. 573–583. – URL https://www.sciencedirect.com/science/article/pii/S016981411100062X. – ISSN 0169-8141

[4] Badawy, Wael ; Julien, Graham A.: *System-on-Chip for Real-Time Applications*. Springer Publishing Company, Incorporated, 2012. – ISBN 1461350344

[5] Chawla, Nitesh V. ; Bowyer, Kevin W. ; Hall, Lawrence O. ; Kegelmeyer, W. P.: SMOTE: Synthetic Minority Over-sampling Technique. In: *J. Artif. Intell. Res.* 16 (2002), S. 321–357. – URL http://dblp.uni-trier.de/db/journals/jair/jair16.html#ChawlaBHK02

[6] Chollet, François u. a.: *Keras*. https://keras.io. 2015

[7] Cui, Yin ; Jia, Menglin ; Lin, Tsung-Yi ; Song, Yang ; Belongie, Serge J.: Class-Balanced Loss Based on Effective Number of Samples. In: *CVPR*. [7], S. 9268–9277. – URL http://dblp.uni-trier.de/db/conf/cvpr/cvpr2019.html#CuiJLSB19

[8] Cunningham, Pádraig ; Cord, Matthieu ; Delany, Sarah J.: *Supervised Learning*. S. 21–49. In: Cord, Matthieu (Hrsg.) ; Cunningham, Pádraig (Hrsg.): *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – URL https://doi.org/10.1007/978-3-540-75171-7_2. – ISBN 978-3-540-75171-7

[9] Davis, Jesse ; Goadrich, Mark: The Relationship between Precision-Recall and ROC Curves. In: *Proceedings of the 23rd International Conference on Machine Learning*. New York, NY, USA : Association for Computing Machinery, 2006 (ICML '06), S. 233–240. – URL https://doi.org/10.1145/1143844.1143874. – ISBN 1595933832

[10] Gale, Trevor ; Elsen, Erich ; Hooker, Sara: *The State of Sparsity in Deep Neural Networks*. 2019

[11] Glorot, Xavier ; Bordes, Antoine ; Bengio, Y.: Deep Sparse Rectifier Neural Networks. In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011* 15 (2011), 01, S. 315–323

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

[13] Hahnloser, RH ; Sarpeshkar, R ; Mahowald, MA ; Douglas, RJ ; Seung, HS: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. In: *Nature* 405 (2000), June, Nr. 6789, S. 947—951. – URL https://doi.org/10.1038/35016072. – ISSN 0028-0836

[14] Harris, Charles R. ; Millman, K. J. ; Walt, St'efan J. van der ; Gommers, Ralf ; Virtanen, Pauli ; Cournapeau, David ; Wieser, Eric ; Taylor, Julian ; Berg, Sebastian ; Smith, Nathaniel J. ; Kern, Robert ; Picus, Matti ; Hoyer, Stephan ; Kerkwijk, Marten H. van ; Brett, Matthew ; Haldane, Allan ; R'io, Jaime F. del ; Wiebe, Mark ; Peterson, Pearu ; G'erard-Marchant, Pierre ; Sheppard, Kevin ; Reddy, Tyler ; Weckesser, Warren ; Abbasi, Hameer ; Gohlke, Christoph ; Oliphant, Travis E.: Array programming with NumPy. In:

Nature 585 (2020), September, Nr. 7825, S. 357–362. – URL https://doi.org/10.1038/s41586-020-2649-2

[15] He, K. ; Zhang, X. ; Ren, S. ; Sun, J.: Deep Residual Learning for Image Recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, S. 770–778

[16] Jarrett, Kevin ; Kavukcuoglu, Koray ; Ranzato, Marc'Aurelio ; Lecun, Yann: What is the Best Multi-Stage Architecture for Object Recognition?, 09 2009

[17] Kingma, Diederik P. ; Ba, Jimmy: *Adam: A Method for Stochastic Optimization*. 2017

[18] Lin, Tsung-Yi ; Maire, Michael ; Belongie, Serge J. ; Bourdev, Lubomir D. ; Girshick, Ross B. ; Hays, James ; Perona, Pietro ; Ramanan, Deva ; Dollár, Piotr ; Zitnick, C. L.: Microsoft COCO: Common Objects in Context. In: *CoRR* abs/1405.0312 (2014). – URL http://arxiv.org/abs/1405.0312

[19] Martin, Robert C. ; Coplien, James O.: *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.] : Prentice Hall, 2009. – URL https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00. – ISBN 9780132350884 0132350882

[20] NVIDIA ; Vingelmann, Péter ; Fitzek, Frank H.: *CUDA, release: 10.2.89*. 2020. – URL https://developer.nvidia.com/cuda-toolkit

[21] Padilla, Rafael ; Passos, Wesley L. ; Dias, Thadeu L. B. ; Netto, Sergio L. ; Silva, Eduardo A. B. da: A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit. In: *Electronics* 10 (2021), Nr. 3. – URL https://www.mdpi.com/2079-9292/10/3/279. – ISSN 2079-9292

[22] Ramachandran, Prajit ; Zoph, Barret ; Le, Quoc V.: *Searching for Activation Functions*. 2017

[23] Redmon, Joseph ; Divvala, Santosh ; Girshick, Ross ; Farhadi, Ali: *You Only Look Once: Unified, Real-Time Object Detection*. 2015. – URL http://arxiv.org/abs/1506.02640. – cite arxiv:1506.02640

[24] Redmon, Joseph ; Farhadi, Ali: *YOLO9000: Better, Faster, Stronger*. 2016. – URL http://arxiv.org/abs/1612.08242. – cite arxiv:1612.08242

[25] Redmon, Joseph ; Farhadi, Ali: *YOLOv3: An Incremental Improvement*. 2018

[26] Rezatofighi, Hamid ; Tsoi, Nathan ; Gwak, JunYoung ; Sadeghian, Amir ; Reid, Ian ; Savarese, Silvio: *Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression*. 2019

[27] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning Representations by Back-propagating Errors. In: *Nature* 323 (1986), Nr. 6088, S. 533–536. – URL http://www.nature.com/articles/323533a0

[28] Salman, Shaeke ; Liu, Xiuwen: *Overfitting Mechanism and Avoidance in Deep Neural Networks*. 2019

[29] Shanmugamani, R. ; Moore, S.M.: *Deep Learning for Computer Vision: Expert Techniques to Train Advanced Neural Networks Using TensorFlow and Keras*. Packt Publishing, 2018. – URL https://books.google.de/books?id=dgdOswEACAAJ. – ISBN 9781788295628

[30] Shorten, Connor ; Khoshgoftaar, T.: A survey on Image Data Augmentation for Deep Learning. In: *Journal of Big Data* 6 (2019), S. 1–48

[31] Sun, Minglei ; Wang, Di ; Wang, Qiwei ; Bi, Shusheng ; Wang, Yuliang ; Yang, Shaobao: Deep learning approach to peripheral leukocyte recognition. In: *PLOS ONE* 14 (2019)

[32] Uijlings, J. R. ; Sande, K. E. ; Gevers, T. ; Smeulders, A. W.: Selective Search for Object Recognition. In: *Int. J. Comput. Vision* 104 (2013), September, Nr. 2, S. 154–171. – URL https://doi.org/10.1007/s11263-013-0620-5. – ISSN 0920-5691

[33] Weisfeld, Matt: *The Object-Oriented Thought Process*. 4th. Addison-Wesley Professional, 2013. – ISBN 0321861272

[34] Ying, Xue: An Overview of Overfitting and its Solutions. In: *Journal of Physics Conference Series* Bd. 1168, Februar 2019, S. 022022

A Appendix

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ____________________

Vorname: ____________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Echtzeit Erkennung von Deutsche Straßenschilder

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original