UPPSALA
UNIVERSITET

IT 19 007
Examensarbete 30 hp
Juni 2019

Real-time object detection for autonomous vehicles using deep learning

Roger Kalliomäki

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Real-time object detection for autonomous vehicles using deep learning

Roger Kalliomäki

Self-driving systems are commonly categorized into three subsystems: perception, planning, and control. In this thesis, the perception problem is studied in the context of real-time object detection for autonomous vehicles. The problem is studied by implementing a cutting-edge real-time object detection deep neural network called Single Shot MultiBox Detector which is trained and evaluated on both real and virtual driving-scene data.

The results show that modern real-time capable object detection networks achieve their fast performance at the expense of detection rate and accuracy. The Single Shot MultiBox Detector network is capable of processing images at over fifty frames per second, but scored a relatively low mean average precision score on a diverse driving-scene dataset provided by Berkeley University. Further development in both hardware and software technologies will presumably result in a better trade-off between run-time and detection rate. However, as the technologies stand today, general real-time object detection networks do not seem to be suitable for high precision tasks, such as visual perception for autonomous vehicles.

Additionally, a comparison is made between two versions of the Single Shot MultiBox Detector network, one trained on a virtual driving-scene dataset from Ford Center for Autonomous Vehicles, and one trained on a subset of the earlier used Berkeley dataset. These results show that synthetic driving scene data possibly could be an alternative to real-life data when training object detecting networks

Handledare: Ahmed Assal
Ämnesgranskare: Anca-Juliana Stoica
Examinator: Mats Daniels
IT 19 007
Tryckt av: Reprocentralen ITC

Acknowledgements

First of all, I would like to thank my supervisor Ahmed Assal for guiding me towards the final goal by helping me to develop my idea from wanting to work with computer vision for self-driving cars, to a structured master thesis. He also suggested to include the experiments with synthetic data, which made the project broader and more interesting. I would also like to thank all the persons at the Cybercom office in Kista, and especially the machine learning team, for a great experience in an outstanding working environment. I want to thank my advisor docent Anca-Juliana Stoica for all the help on developing a good content and well-structured thesis project report. Last but not least, loving thanks to my girlfriend Tanja, for her endurance and support through the sometimes stressful times working on this thesis.

Contents

Abstract **iv**

Acknowledgements **v**

List of Figures **xii**

List of Tables **xiii**

Abbreviations **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 2

2 Background **3**
2.1 Machine Learning . . . 3
2.2 Artificial Neural Networks . . . 4
2.2.1 Activation Functions . . . 4
2.3 Training Artificial Neural Networks . . . 6
2.3.1 Loss Functions . . . 6
2.3.2 Backpropagation . . . 7
2.3.3 Optimization Algorithms . . . 7
2.3.4 Overfitting . . . 8
2.3.5 Regularization . . . 9
2.3.6 Hyperparameters . . . 10
2.3.7 Data Preprocessing . . . 10
2.3.8 Parameter Initialization . . . 11
2.4 Deep Neural Networks . . . 12
2.5 Computer Vision . . . 13
2.6 Convolutional Neural Networks . . . 14
2.6.1 Convolution Layers . . . 14
2.6.2 Pooling Layers . . . 15
2.6.3 CNN Architecture . . . 15
2.7 Object Detection . . . 16
2.8 Visual Perception for Autonomous Vehicles . . . 17
2.9 Real-time Object Detection for Autonomous Vehicles . . . 18
2.10 Virtual Environments . . . 19
2.11 Single Shot MultiBox Detector . . . 20

3 Problem Formulation **25**
3.1 Performance Evaluation and Documentation . . . 26

4 Experimental Work – Preparations **29**
4.1 Virtual Computing Platform . . . 29
4.2 Tools . . . 30
4.2.1 TensorFlow . . . 30
4.2.2 Keras . . . 30
4.3 Datasets . . . 30
4.3.1 BDD100K . . . 31
4.3.2 FCAV . . . 33
4.3.3 KITTI . . . 34
4.3.4 PASCAL . . . 34
4.3.5 Annotations . . . 35
4.4 Data Analysis and Preprocessing . . . 35
4.5 Implementation Example of Building a CNN in Keras . . . 40

5 Implementation of the Complete SSD Network in Keras **43**
5.1 Train-, Validation-, and Test-data . . . 44
5.2 Training . . . 44
5.2.1 PASCAL . . . 44
5.2.2 BDD100K . . . 45
5.2.3 Virtual and Real Data . . . 47
5.3 Testing . . . 48
5.4 Results, Analysis and Evaluation . . . 49
5.4.1 PASCAL . . . 49
5.4.2 BDD100K . . . 52
5.4.3 Virtual and Real Data . . . 56

6 Related Work **59**
6.1 One-stage networks . . . 59
6.1.1 YOLO . . . 59
6.2 Two-stage networks . . . 60
6.2.1 Faster R-CNN . . . 60
6.2.2 R-FCN . . . 61

7 Discussion and Future Work **63**
7.1 Discussion . . . 63
7.1.1 Research Questions . . . 65
7.2 Future Work . . . 66

8 Conclusion **69**

Bibliography **71**

A SSD Architecture **77**

B Original Project Code **81**
B.1 JSON to XML Annotations . . . 81
B.2 Determine and Save Bounding Box Sizes . . . 82
B.3 Plot Bounding Box Sizes . . . 84
B.4 Determine and Save Aspect Ratios . . . 84
B.5 Plot Aspect Ratios . . . 86
B.6 Determine and Save Object Centers . . . 86
B.7 Plot Heat Map . . . 87

B.8 Determine and Save Mean RGB Values 89
B.9 Plot Mean RGB Values 90
B.10 Split Dataset 91
B.11 Plot Training History 91
B.12 Annotate Video 92

List of Figures

2.1 Simple neural network 4
2.2 Activation functions 5
2.3 Visualization of loss surfaces 6
2.4 Overfitting loss functions 8
2.5 Data preprocessing pipeline 10
2.6 Simple deep neural network 12
2.7 Convolutional operation 14
2.8 Visualization of CNN kernels 15
2.9 Max pooling 15
2.10 Layer activations visualized in a small CNN 16
2.11 Visualization of a deep CNN 16
2.12 Visualization of ground truth bounding boxes 17
2.13 Real-world image with corresponding virtual image 19
2.14 A comparison between two single shot detection networks 20
2.15 SSD framework 21
2.16 Receptive fields of SSD 22
2.17 Non-maximum suppression 23

3.1 Intersection over Union 26
3.2 Precision-recall curve examples 27

4.1 Example images from BDD100K 31
4.2 Distribution of images scenarios in BDD100K 32
4.3 Object counts in BDD100K 32
4.4 Object size distribution in BDD100K 32
4.5 Example images from FCAV 33
4.6 Example images from KITTI 34
4.7 Example images from PASCAL 35
4.8 Scatter plots showing bounding box sizes 36
4.9 Histograms with aspect ratios 37
4.10 Histograms with bounding box sizes 38
4.11 Heatmap of object instance 38
4.12 3D scatter plot of mean RGB values 39
4.13 Mean subtraction for BDD100K 39
4.14 Mean subtraction for FCAV 40
4.15 Mean subtraction for KITTI 40

5.1 Architecture of SSD 43
5.2 Training and validation losses for PASCAL 45
5.3 Training and validation losses for BDD100K 46
5.4 Training and validation losses for cars in BDD100K 47
5.5 Training and validation losses for FCAV 48
5.6 Precision-recall curves for PASCAL 51

5.7 Precision-recall curves for BDD100K 54
5.8 SSD predictions on BDD100K test images 55
5.9 Precision-recall curves for BDD100K 56
5.10 Precision-recall curves for FCAV 57

6.1 YOLO detection model 60
6.2 R-CNN type network overview 61
6.3 R-FCN network overview 62

7.1 Visualization of detections by SSD 64

List of Tables

5.1 Final AP values for all classes in PASCAL 49
5.2 Final mAP values on PASCAL 50
5.3 Final AP values for all classes in BDD100K 53
5.4 Final AP values for cars in BDD100K 56
5.5 Final AP values for FCAV 56

Abbreviations

AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
AP	**A**verage **P**recision
BDD100K	**B**erkeley **D**eep **D**rive **100k**
BAIR	**B**erkeley **A**rtificial **I**ntelligence **R**esearch
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CV	**C**omputer **V**ision
DNN	**D**eep **N**eural **N**etwork
FCAV	**F**ord **C**enter **A**utonomous **V**ehicles
FPS	**F**rames **P**er **S**econd
GCE	**G**oogle **C**ompute **E**ngine
GCP	**G**oogle **C**loud **P**latform
GPU	**G**raphics **P**rocessing **U**nit
IoU	**I**ntersection **o**ver **U**nion
KITTI	**K**arlsruhe **I**nstitute of Technology and **T**oyota **T**echnological **I**nstitute
mAP	**m**ean **A**verage **P**recision
ML	**M**achine **L**earning
NMS	**N**on-**M**aximum **S**upression
PASCAL	**P**attern **A**nalysis **S**tatistical Modelling and **C**omputational **L**earning
PCA	**P**rincipal **C**omponent **A**nalysis
RPN	**R**egion **P**roposal **N**etwork
SGD	**S**tochastic **G**radient **D**escent
SSD	**S**ingle **S**hot MultiBox **D**etector
SVM	**S**upport **V**ector **M**achine
VM	**V**irtual **M**achine
WAD	**W**orkshop of **A**utonomous **D**riving
YOLO	**Y**ou **O**nly **L**ook **O**nce

Chapter 1

Introduction

Enabling a computer to extract information from digital images or videos falls into the field of computer vision (CV). This field is rapidly growing along-side the rise of deep learning and is used in conjunction with numerous problems within artificial intelligence (AI). A common problem in CV, which for a long time was considered to be a hard problem to solve, is image classification. Nowadays, the popularity and substantial research in deep learning have enabled a special kind of deep neural network, called Convolutional Neural Network (CNN), to surpass human image classification performance [1].

However, for a self-driving car, simply classifying images from its surroundings is not good enough. Instead, the system must be able to detect, localize and classify multiple objects using camera imagery. This process is commonly called object detection [2] and is a much harder task than merely classifying images into distinct classes. Moreover, safe driving requires more than just detecting the road you are driving on, along with other vehicles, persons, and animals; you also have to know how they are most likely to act, and how to respond. To be able to draw these conclusions and use the information to act accordingly, it is desirable to have a system that is capable of processing images from onboard cameras in real-time [3].

Although the pursuit of self-driving cars has been going on since the 60s, getting them out on real roads and interacting with real-life traffic scenarios, is something that has not been possible until the last ten years [4]. During these years, driverless cars have used technologies like Radar, LiDAR, GPS and other sensors for mapping the surroundings of the car [4]. However, in the last few years, there have emerged some deep neural network (DNN) architectures capable of live video-feed object detection, like YOLO (You Only Look Once) [5], and SSD (Single Shot MultiBox Detector) [6], with the potential to be used as part of autonomous vehicle systems.

One drawback with these object detection architectures, along with other deep neural networks, is that they require vast amounts of labeled data to train [7] — in the case of autonomous vehicles, images with objects, such as cars, street-signs, roads, and humans with accurate annotations. Creating these datasets is both time-consuming and expensive due to the requirements for them to be representative for a variety of driving conditions and scenarios. One possible alternative for collecting data in real-life driving scenes would be to use virtual environments to collect synthetic data. Deep neural network object detectors could then be trained using this virtual data to recognize real-life objects in real-life scenes.

1.1 Motivation

Self-driving systems are commonly categorized into three subsystems: perception, planning, and control [4]. The perception system is responsible for translating raw sensor data into a model of the surrounding environment, the planning system

makes purposeful decisions based on the environment model, and the control system executes planned actions. The objective of this thesis is to study the perception problem in the context of real-time object detection for autonomous vehicles.

Although we are standing at the edge to a new era of automated driving, this technology is still far from being mature. Self-driving cars have to interact and operate in very unpredictable dynamic environments with multiple actors such as pedestrians, other vehicles, and animals. Therefore, there is a need to provide autonomous vehicles with reliable perception systems to facilitate and improve the vehicle's ability to understand the surrounding environment. In the past, self-driving cars mostly have relied on technologies like Radar, LiDAR, GPS and other sensors when mapping the surroundings [4]. However, compared to these sensors cameras are substantially less expensive, and therefore it would be, from a purely financial point of view, highly beneficial to incorporate them more in self-driving systems. Moreover, cameras have a higher resolution than any other used sensors, and there exist more available image-based datasets [4].

In the last few years, there have emerged some live video-feed object detection architectures [5, 6, 8, 9, 10, 11] with the potential to be used as part of autonomous vehicle visual perception systems. In addition to these new object detection architectures, Berkeley Artificial Intelligence Research (BAIR) just recently[1] released Berkeley Deep Drive 100k (BDD100K), which is the largest and most diverse driving scene dataset to this date [12].

With the rise of deep learning, data has proven to be both the limiting and driving factor when training DNNs, particularly within CV [13, 14, 15, 16]. Creating diverse real-life datasets for driving scenarios is both time-consuming and expensive. Instead, virtual datasets can be automatically constructed and labeled with much higher speed and accuracy, and thereby they could help to solve the big deficit in labeled data for self-driving systems [17, 18, 19]. Therefore, if real-time object detectors trained on virtual images is shown to have comparable performance to real-time object detectors trained on real-life collected data, or if synthetic data can be used as complementary data during training, it could accelerate the development of self-driving systems.

1.2 Contributions

- A systematic review on the background and theory for a representative cutting-edge real-time object detection deep neural network architecture (Single Shot MultiBox Detector (SSD)).
- Training a state-of-the-art object detection network (SSD), capable of real-time object detection, utilizing the newly released diverse driving scene dataset BDD100K and evaluating the performance.
- Training a state-of-the-art object detection network (SSD), capable of real-time object detection, utilizing synthetic driving scene data and comparing the results with the BDD100K results. This comparison will give an indication of how big the gap is between real and virtual driving scene data, from the perspective of object detection architectures.

[1] BDD100K was released May 30, 2018.

Chapter 2

Background

This chapter consists of a review of the background and theory of object detection networks and computer vision for self-driving systems. This review includes an introduction to the basics of machine learning, artificial neural networks, deep learning, computer vision, and convolutional neural networks. These concepts play a crucial part in object detection and visual perception for autonomous vehicles. In the later parts of the chapter, an introduction to virtual environments is given. Virtual environments could be used to collect synthetic data for training the deep neural networks utilized by self-driving visual perception systems. The last section is dedicated to explaining the structure and key operations of a representative, cutting-edge, real-time object detection deep neural network architecture (SSD) which is implemented and evaluated in this thesis project.

2.1 Machine Learning

Today, Artificial Intelligence (AI) and Machine Learning (ML) are thriving fields with many practical applications and active research topics. AI systems are used in many everyday situations in our life's like for example search engines, digital personal assistants, chat-bots and spam filters. We also use intelligent software to automate routine labor, make diagnoses in medicine and support basic scientific research [20, p.28-29].

AI systems often need to be able to look at raw data and achieve knowledge by extracting patterns from that data, without being explicitly programmed. This capability is known as ML [21]. Although AI suffers from shifting definitions and is often used to describe whatever is yet to be discovered, ML is usually divided into three distinct categories: supervised learning, unsupervised learning, and reinforcement learning.

Unsupervised learning uses patterns in input data to do classification without knowing which data belong to which class. Reinforcement learning is to learn by trial-and-error through a series of rewards or punishments to maximize or minimize some long-term scalar value. The goal of supervised learning is to approximate a mapping function which can take never seen input data and predict the output variables for that data. This approximation is made by letting the system learn by searching through the space of possible solutions. Given N number of training input-output vector pairs $(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$, where each y_j is given by an unknown function $y = f(x)$, the task for the system is to find a function h that approximates the unknown true function f [20, p.695].

Most modern deep learning models are based on multilayered artificial neural networks. Although there exist techniques and methods to train artificial neural

networks using unsupervised learning, they are more commonly trained using supervised learning [7].

2.2 Artificial Neural Networks

Fundamentally, an Artificial neural network (ANN) is a simple computational system consisting of an input layer interconnected with an output layer through one hidden layer. The hidden layer consists of multiple simple processing units called nodes, or neurons, as seen in figure 2.1. These artificial neurons conduct a dot product between trainable weights of the neuron and given input values. After this dot product, a bias is added, and the result is passed to an activation function (see section 2.2.1) which adds some non-linearity to the system [20, p.728]. With a given input $(x_1, x_2, ..., x_n)$, weights $(w_1, w_2, ..., w_n)$, bias b, and a non-linear activation function ϕ, the output of a neuron in a hidden layer is

$$\phi\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{2.1}$$

The output layer also consists of neurons, but most commonly they do not have an activation function. This is because the last output layer is usually used for presenting the class scores. These scores are often just some arbitrary real numbers used to predict class belongings, or more targeted values to represent probabilities [22].

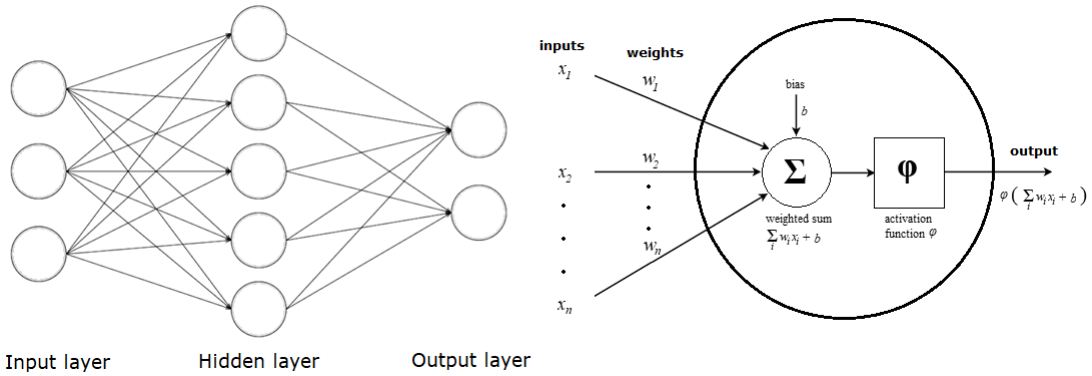

FIGURE 2.1: *To the left:* Visualization of a simple shallow neural network with one hidden layer. *To the right:* Visualization of a node/neuron in the hidden layer.

The network is trained by adjusting the weights and biases of each node, which are known as trainable parameters, to find a constellation of parameters that approximates a function which maps input data to some desirable output data (see section 2.3). Depending on the problem complexity and format, the network can be designed with a different number of neurons in the hidden layer, and one of various available activation functions [20, p.728-732].

2.2.1 Activation Functions

Activation functions take a single number and performs a specific fixed mathematical operation on it. They are used in ANNs to limit the output of individual nodes, and due to their non-linearity, they also allow different variations of input data to be mapped to the same output values separately. An example of this would be when an

image classification ANN classifies different breeds of cats to the same class. There are various activation functions used in ANNs, among which the following are the most commonly used [23].

$$\phi(x) = \begin{cases} sigmoid(x) = \frac{1}{1+e^{-x}} \\ tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} \\ ReLU(x) = max(0, x) \\ softplus(x) = ln(1+e^{x}) \end{cases} \tag{2.2}$$

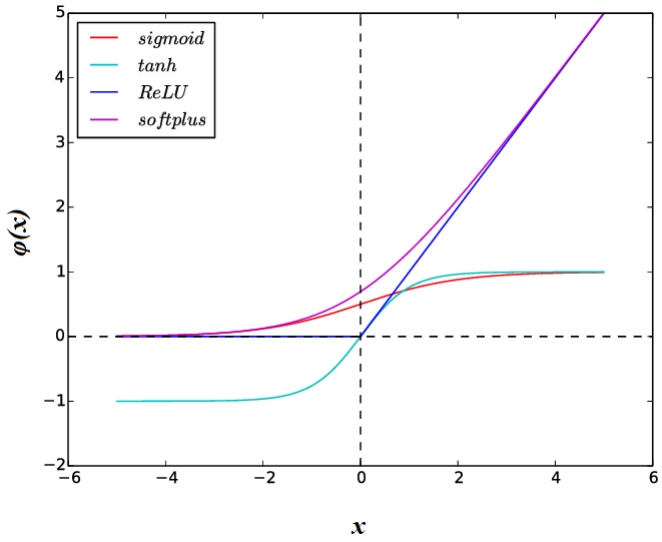

FIGURE 2.2: The most common activation functions used in ANNs.
Figure from: [23]

- **Sigmoid -** The *sigmoid* (or *logistic*) function takes a real-valued number and maps it to a value between 0 and 1. The most significant effect of the function is that for large positive numbers the output is 1, and for large negative numbers the output is 0. This effect leads to the very undesirable property of saturating the output to either 0 or 1. Moreover, *sigmoid* outputs are not zero-centered, which has implications on the dynamics during gradient descent (see sections 2.3.3 and 2.3.8). Historically, the *sigmoid* function has been the most widely used activation function. However, due to the saturating effect, and the output not being zero-centered, in practice, the *sigmoid* function is rarely ever used in modern neural networks [22].

- **Hyperbolic Tangent -** The *tanh* function takes a real-valued number and maps it to a value between -1 and 1. Although its activations also saturates, unlike *sigmoid*, *tanh*'s output is zero-centered. Therefore, when designing a ANN, the *tanh* function is always preferred to the *sigmoid* function [22].

- **ReLU -** The *Rectified Linear Unit* function gives an output x if x is positive and 0 otherwise. Compared to the *tanh* and *sigmoid* functions that involve expensive operations, *ReLU* can be implemented by simply limiting a matrix of activations at zero. It also experiences fewer problems with vanishing gradients compared to *sigmoid* or *tanh* that saturate in both directions. Some downsides with *ReLU* are that it is not zero-centered, nor is it differentiable at zero which

can create problems with learning, as numerical gradients calculated near zero can be incorrect [24].

- **Softplus -** The *softplus* (or *analytic function*) is a differentiable approximation of *ReLU*. It has similar properties to *ReLU*, but comes with a higher computational cost during training [24].

2.3 Training Artificial Neural Networks

In the training phase, the correct output values for corresponding input values are known beforehand. These known values are often called *ground truth*. An ANN is trained by first making a forward pass of training data through the network, followed by an adjustment of the trainable parameters of each neuron by evaluating how close the output of the network is to the ground truth [22]. To properly make these adjustments, a gradient vector is computed for each trainable parameter, which indicates by what amount the error would decrease (or increase) if the parameter were slightly changed [25]. To be able to evaluate how close the output of the network is to the ground truth, a loss function (sometimes referred to as a cost or objective function) is used.

2.3.1 Loss Functions

The loss function of an ANN is a differentiable function used to measure the inconsistency between predicted output values and ground truth values. It outputs a non-negative real number where decreasing values represents increasing correctness of the network. It can be looked at as a representation of a hilly landscape in the high-dimensional space of trainable parameters of the network (see figure 2.3 for a simplified three-dimensional representation). The gradient vector computed for each trainable parameter indicates in what direction the next step in this landscape should be taken to get nearer to a minimum, where the output of the loss function is low on average across the entire training dataset.

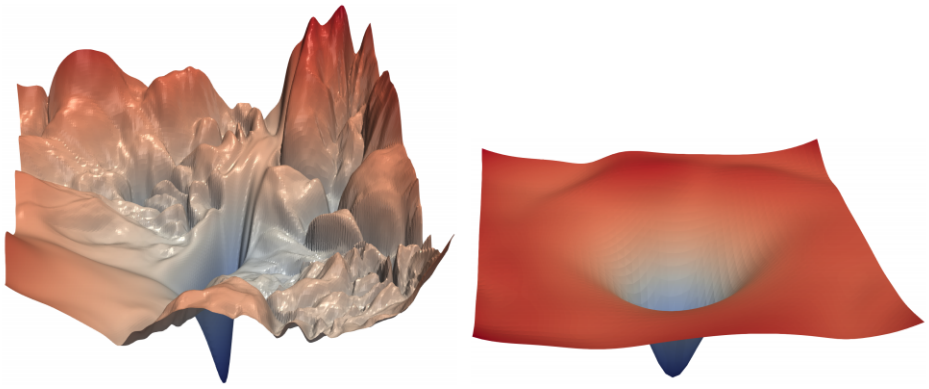

FIGURE 2.3: Three-dimensional visualization of two neural network loss surfaces. *Figure from:* [26]

With input training data $\hat{X} = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_m)$ and accompanying ground truth data $\hat{Y} = (\hat{y}_1, \hat{y}_2, ..., \hat{y}_m)$, the loss function $L_{\hat{\theta}}$, given a certain set of trainable parameters $\hat{\theta} = (\theta_1, \theta_2, ..., \theta_n)$, is

$$L_{\hat{\theta}} = k \sum_{i=1}^{m} \ell \left(f(\hat{x}_i), \hat{y}_i \right), \tag{2.3}$$

where m is the number of training samples, k is a constant usually set to $1/m$ or 1, and $\ell(f(\hat{x}_i), \hat{y}_i)$ estimates how well the network predicts an output of a training sample $f(\hat{x}_i)$ compared to the corresponding ground truth $\hat{y}_i$ [26].

Which loss function to use when designing an ANN depends on the problem to be solved, and is determined by a problem specific metric (see section 3.1) that should be maximized or minimized. There are various loss functions used in ANNs today, among which the following are some of the most prevalent [26, 27, 28].

- **Mean Squared Error -** $L_{\hat{\theta}}^{MSE} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - f(\hat{x}_i))^2$
- **Mean Absolute Error -** $L_{\hat{\theta}}^{MAE} = \frac{1}{m} \sum_{i=1}^{m} |\hat{y}_i - f(\hat{x}_i)|$
- **L1 -** $L_{\hat{\theta}}^{L1} = \sum_{i=1}^{m} |\hat{y}_i - f(\hat{x}_i)|$
- **L2 -** $L_{\hat{\theta}}^{L2} = \sum_{i=1}^{m} (\hat{y}_i - f(\hat{x}_i))^2$
- **Cross Entropy -** $L_{\hat{\theta}}^{CE} = -\frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}_i \cdot \log(\hat{x}_i) + (1 - \hat{y}_i) \cdot \log(1 - \hat{x}_i) \right)$
- **Hinge -** $L_{\hat{\theta}}^{H} = \frac{1}{m} \sum_{i=1}^{m} max(0, 1 - \hat{y}_i \cdot \hat{x}_i)$

2.3.2 Backpropagation

Backpropagation is used to compute the partial derivative $\partial L / \partial \hat{\theta}$ of the loss function L with respect to any trainable parameter θ in the network. The partial derivatives are found by propagating the error gradient of the loss function with respect to the network output backward through the network layers using the chain rule. Parameters which have larger effect on the final output will, therefore, share more of the error gradient. The error derivative can then be used in gradient descent (see section 2.3.3) to improve upon the parameters iteratively [29].

2.3.3 Optimization Algorithms

There exist multiple optimization algorithms for minimizing the loss function used in neural networks. Most commonly these algorithms are different versions of gradient descent optimization. Gradient descent updates the network parameters $\hat{\theta} = (\theta_1, \theta_2, ..., \theta_n)$ in the opposite direction of the gradient of the loss function $\partial L / \partial \hat{\theta}$ for each trainable parameter as follows [30]:

$$\Delta \theta_j = -\eta \frac{\partial L}{\partial \theta_j} \tag{2.4}$$

Here, η is the learning rate which determines the size of the steps taken when searching for a minimum and $\Delta \theta_j$ is the update change of a trainable parameter θ_j. The learning rate is commonly initialized by experimental trial and error and is often gradually decreased over iterations.

Vanilla gradient descent, or *batch gradient descent*, updates the network parameters for the entire training dataset at the same time. Since the gradients for the complete dataset have to be computed to perform just one update, batch gradient

descent is both slow and incapable of updating the network parameters online, or in other words, not capable of including new training samples on-the-fly [31]. Contrary to batch gradient descent, *Stochastic gradient descent* (SGD) performs a parameter update for each training sample $\hat{x}_i$ and accompanying ground truth data $\hat{y}_i$. It is therefore much faster for each parameter update and usually also for convergence, but not epoch[1] wise due to vector computations. SGD is also capable of learning online. However, due to the frequent parameter updates, SGD often causes the loss function to fluctuate during the learning phase which may cause it to overshoot and miss local or global minimums [31]. *Mini-batch gradient descent* calculates the loss gradient with respect to the model parameters for each sample in a subset, or mini-batch[2], of the training data. Using mini-batches reduces the variance of the parameter updates and therefore often lead to more stable convergence [31].

Around local minima, the loss surface often curves more steeply in a particular dimension. These steep loss "cliffs" may cause oscillations during the gradient descent. To dampen oscillations of this kind and also help accelerate the descent, a momentum term is often used in the parameter updates. Momentum adds a fraction α of the past time step update, $\Delta\theta_j^{t-1}$, to the current one, $\Delta\theta_j^t$, which causes smoother parameter changes over time [30]:

$$\Delta\theta_j^t = -\eta \frac{\partial L}{\partial \theta_j^t} + \alpha \Delta\theta_j^{t-1} \tag{2.5}$$

Different versions of mini-batch gradient descent with momentum is most often used when training neural networks, including Adaptive Moment Estimation (Adam), RMSprop and Adagrad [5, 6, 32, 33, 34, 35].

2.3.4 Overfitting

The primary goal when training an ANN is to generate a network that generalizes well beyond the training dataset. In other words, the network should be able to estimate the actual distribution from which the data is sampled from. An ANN with a large number of parameters may be able to memorize the complete training dataset if that dataset consists of too few data-samples. This behavior of memorization is generally known as overfitting. A common way of detecting if a network is overfitting during training is to use a small subset of the training data as a validation set. This new small subset is used to provide an unbiased evaluation of the networks fit on the training dataset (see left side of figure 2.4). Like all complex statistical models, ANNs with high complexity and too many trainable parameters are prone to overfitting [20, p.736] (see right side of figure 2.4).

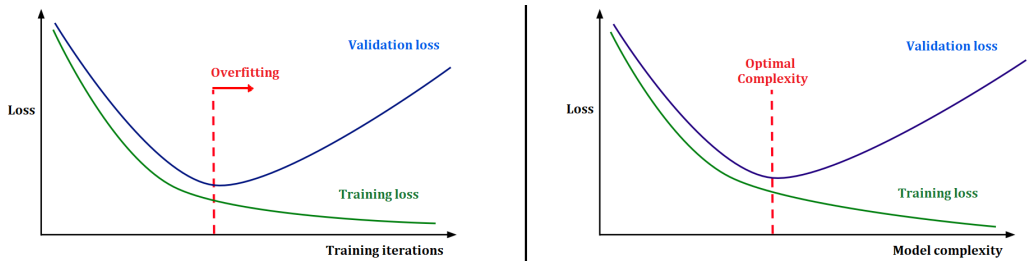

FIGURE 2.4: *To the left:* Loss as a function of training iterations. *To the right:* Loss as a function of model complexity.

[1] One epoch is usually defined as going through the complete dataset once.
[2] Using a mini-batch value of one will consequently result in the standard SGD optimizer.

The most straightforward way to fight overfitting during training is to use more training data, since a larger training dataset reduces the networks memorization capability. Obviously, in practice, supplemental data will not always be available. In those cases different regularization strategies can be adopted if a network is showing signs of overfitting.

2.3.5 Regularization

In most practical cases, the amount of available data is limited. One popular regularization strategy used to get around this problem is to transform existing data into new additional augmented data. This technique is commonly called *data augmentation* and is particularly effective when working with image datasets [36, p.236-237]. Some common image augmentation operations used are rotating, cropping, scaling, filtering and transforming pixels of images in the dataset. Another data altering regularization technique is *Noise injection* which is utilized by adding noise to the data. This injection can also be seen as a kind of data augmentation but is most often conducted when the data is fed to the input of the network and does therefore not increase the number of training data samples [36, p.237-238].

Another way to counteract overfitting is to reduce the complexity of the network by adding a term to the parameter update rule or the loss function. This strategy limits the magnitude of the free parameters of the network and is typically called L_2 *regularization* if the term is included in the loss function, or, as shown here, *weight decay* if the term is added to the update rule:

$$\Delta\theta_j = -\eta \left(\lambda\theta_j + \frac{\partial L}{\partial \theta_j} \right) \tag{2.6}$$

Here, λ is the regularization parameter which causes the parameter to decay in proportion to its size. In practice, this penalizes large parameters and effectively limits the freedom of the network [36, p.227-230].

Dropout provides an inexpensive approximation of many different ANNs making predictions at test time, without requiring excessive computational costs or memory requirements. The technique works by randomly deactivated, or drop, neurons from the ANN during training with a deactivation probability p. Dropout prevents neurons from co-adapting too much and trains the collection of all subsets of networks that can be formed by removing non-output neurons from the original ANN. Typically, for an input neuron, p is set to 0.2, and for a hidden neuron, p is usually set to 0.5. When training the network, the forward pass, back-propagation, and learning updates are conducted as usual. At test time, the average of the evaluation of the subset networks is approximated by simply using the thinned out version of the original ANN [36, p.255-265].

Early stopping is, due to both its effectiveness and simplicity, one of the most prevalent forms of regularization used when training ANNs [36, p.243]. The key idea of early stopping is to stop training just before the network begins to overfit, or memorize, the training data, which would consequently make the performance on the test data to start to decrease. To be able to find the point when overfitting begins, the error on the validation set is used as an approximation for when the network starts to memorize the training data (see figure 2.4) [36, p.241-243].

2.3.6 Hyperparameters

A hyperparameter is a parameter that is not learned by the network and whose value is initialized before the training takes place. Setting the hyperparameters of an ANN can be seen as model selection, where the model is chosen from the set of possible models generated by the hyperparameters themselves. The most common hyperparameters in the context of ANNs include the learning rate, number of hidden neurons, weight decay, loss function, batch size, and momentum. Specialized ANNs may have many other hyperparameters, specifying the structure of the network itself and those which determine how the network is trained.

2.3.7 Data Preprocessing

There are numerous forms of data prepossessing that can affect the training performance of an ANN. In the process of training the network, the initial inputs are going to be multiplied with weights and cause activations that then can be used to train the network with backpropagation and gradient descent. Therefore it is desirable that each feature of the dataset have similar properties, but without any biases like all positive values, to keep the gradients under control [22].

Zero-centering the data by *mean subtraction* is the most common form of data preprocessing and has the geometric interpretation of "moving" the dataset along every dimension in a way that the data gets centered around the origin (see middle of figure 2.5). When working with images, a convenience strategy is to subtract single values from all pixels or to perform the subtraction separately across the three color channels [22].

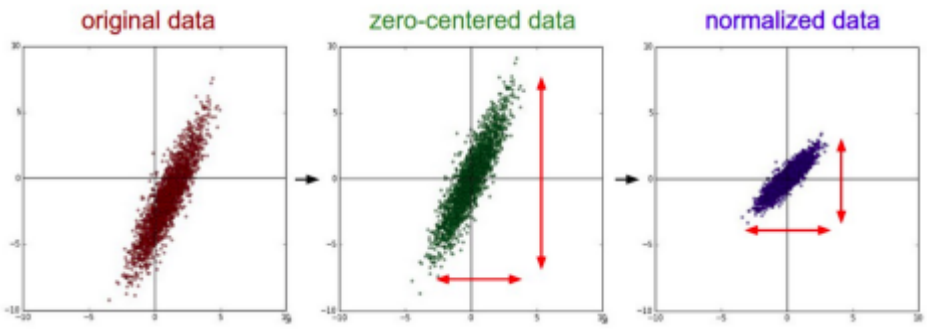

FIGURE 2.5: Common data preprocessing techniques. *Left*: 2D representation of high-dimensional original data. *Middle*: Zero-centered data obtained by subtracting the mean in each dimension. *Right*: The zero-centered data is normalized by scaling by its standard deviation. Red lines indicate the extent of the data. *Figure from:* [22]

In many cases, the features in the input data may have different scales or units, but should nevertheless affect the training of the network by a relatively equal amount. To achieve this effect *normalization* can be used to normalize the dimensions of the data in a way so that they have approximately the same scale (see right side of figure 2.5). Image pixels are ranged from 0 to 255 and are priory already on a comparative scale. Therefore, normalization on image datasets will often not contribute with any significant improvements when training an ANN [22].

2.3.8 Parameter Initialization

The initial values of the trainable parameters in an ANN (*e.g.*, weights and biases) play an essential role when training the network. For example, using too low initial values may introduce minimal gradients and therefore prohibit quick learning. Contrarily, too large initial values may cause radically large gradients and unstable training. Bringing the example of low initial values to its extreme — if all weights are set to be zero, the derivative with respect to the loss function will be identical for all weights. In other words, there will be nothing to break the symmetry between neurons because all weights will be updated equally and have identical values in the subsequent iteration. Therefore, if all weights are initialized to zero, the complexity of the network would be the same as that of a single neuron. However, the biases of the network can be initialized to zero as non zero weights will break the symmetry. A better solution than zero initialization is to instead initialize all weights to small random numbers sampled from a multi-dimensional Gaussian with zero mean. Neurons initialized this way will produce distinct gradients and become varied fragments of the full network [22].

Although this solution is better than zero initialization, random Gaussian weights present new weaknesses to deeper networks with multiple hidden layers (see section 2.4). These weaknesses mainly consist of uncalibrated neuron output variances. A deep neural network initialized with random Gaussian weights will have neurons that output distributions with a variance that grows with the number of neuron inputs. A way to address the uncalibrated variances problem is to normalize the variance of each neuron's output to 1. This normalization is done by again initialize each neuron's weight w_i from a standard normal distribution with zero mean, but also scale with the square root of the number of neuron inputs n [22]:

$$w_i \sim \frac{\mathcal{N}(0,1)}{\sqrt{n}} \tag{2.7}$$

This initialization ensures neurons with approximately the same output distribution which improves the rate of training convergence [22]. The derivation can be shown by examine the variance of a simplified neuron output $s = \sum_{i=1}^{n} w_i x_i$, without bias or activation function, where $(w_1, w_2, ..., w_n)$ is the weights of the neuron with input $(x_1, x_2, ..., x_n)$:

$$Var(s) = Var\left(\sum_{i=1}^{n} w_i x_i\right) \tag{2.8}$$

$$= \sum_{i=1}^{n} Var(w_i x_i) \tag{2.9}$$

$$= \sum_{i=1}^{n} \left([E(w_i)]^2 Var(x_i) + [E(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i)\right) \tag{2.10}$$

$$= \sum_{i=1}^{n} Var(w_i)Var(x_i) \tag{2.11}$$

$$= nVar(w_i)Var(x_i) \tag{2.12}$$

$$= Var(\sqrt{n}w_i)Var(x_i) \tag{2.13}$$

Here, equations 2.9, 2.10 and 2.13 are found with the properties of variance. Equation 2.11 holds for zero mean inputs which have expected values $E(w_i) = E(x_i) = 0$, and equation 2.12 assumes identical distributions for all w_i and x_i. From this derivation its clear that if s is to have the same variance as all of its inputs x_i, the weights should be draw from a unit Gaussian and scaled by the square root of the number of neuron inputs as seen in equation 2.7.

Xavier initialization [37] is another variance calibration method which normalizes the variance of each neuron's weight as follows:

$$Var(w_i) = \frac{2}{n_{in} + n_{out}} \quad , \forall i \tag{2.14}$$

Here n_{in}, n_{out} are the number of neurons in the previous and next layer respectively. This normalization can be done by initializing neuron weights with something the authors of the paper call the *normalized initialization* which sample random numbers from a uniform distribution between minus one and one, scaled by a normalization factor of $\sqrt{6}/\sqrt{n_{in} + n_{out}}$:

$$w_i \sim U\left[\frac{-\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right] \tag{2.15}$$

Random Gaussian weight initialization is often good enough for shallow ANNs. However, when designing more intricate deeper networks, other methods like Xavier or similar initialization techniques are commonly used to address the uncalibrated variance problem [22].

2.4 Deep Neural Networks

A Deep Neural Network (DNN) is essentially an ANN with two or more hidden layers. Since all trainable parameters of the network are learned with backpropagation, each hidden layer can be trained to automatically learn representations of input data with different levels of abstraction. With the act of adding more layers, deeper networks can represent functions of increasing complexity. In most practical deep learning scenarios the best generalizing models are most often big models (with many layers and a large number of trainable parameters) that has been regularized appropriately [36, p.225].

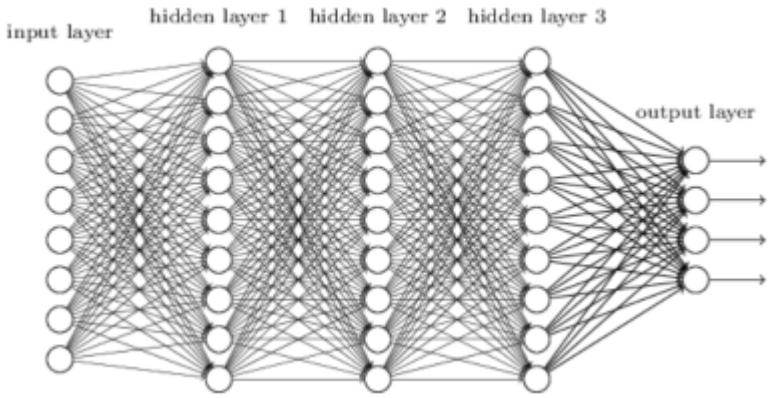

FIGURE 2.6: Visualization of a simple deep neural network with three hidden layers. *Figure from:* [38]

If a problem consists of mapping an input vector to an output vector, it can be solved with deep learning, given an adequately deep network and a sufficiently large dataset of labeled training examples [36, p.163]. A typical modern DNN may be composed of hundreds of millions of trainable weights, and therefore require hundreds of millions of labeled training examples [36, p.194].

Two significant drawbacks with more extensive deep networks are that they take longer to train, and they need more training data to converge, compared to shallow and less complex networks. An additional challenge with training deep neural networks is that as a network grows, earlier layers have a lower effect on the output. Therefore, the gradient which is backpropagated and used to update the weights of these early layers will get decreasingly small. Moreover, DNNs are prone to overfitting and often require extensive regularization methods to work properly [39].

However, in the last few years, there has been a tremendous growth in deep learning, mainly driven by faster more powerful computers, the availability of more massive datasets, and better techniques to train deeper networks. As a result, DNNs are beginning to be used within a wide range of scientific fields, not least in the field of computer vision [16].

2.5 Computer Vision

Computer vision (CV) is a branch of computer science that has seen great progress thanks to the growth of deep learning. CV involves processing and analyzing real-world digital images and videos, allowing computers to extract information from the physical world. Although perception appears as an easy task for humans, it has shown to be a much harder problem for computers, involving a great deal of sophisticated computation [13].

Previously, if presented with a task such as image classification, the usual technique was to perform a step called feature extraction. This step consisted of manually defining keypoint features described by the presence of edges or different pixel patterns surrounding point locations in an image. These features were then used to identify the different objects, or classes. An image was classified as a specific object if a satisfying number of features from one class were located in the image [40, p. 207-234]. One apparent difficulty with this approach is that the features have to be manually created and specified. This becomes inconvenient and almost impossible when the number of classes and features becomes very large.

Since digital images can be represented as vectors, matrices or tensors, images can easily be used as input to DNNs. By using the output in the form of a vector of scores, one for each category, image classification problems have been successively solved using supervised deep learning [32, 33, 41]. Since layers in DNNs are trained to learn representations of input data automatically, these layers can replace painstakingly manually defined keypoint features. Moreover, as mentioned earlier, neural networks with deeper architectures have the capacity to learn more complex features than shallow networks.

A particular kind of DNN, called Convolutional Neural Network (CNN), has shown to be exceptionally suitable for a variety of problems in CV and has therefore lately been widely adopted by the CV community [16, 42]. In 2012, a CNN called AlexNet [32] was the first DNN to win the classification challenge ImageNet Large Scale Visual Recognition Competition (ILSVRC) [13]. The success of AlexNet launched an explosion of interest in CNNs and deep learning in general. Large numbers of variations and advances on AlexNet such as VGGNet [33], InceptionNet

[43], and ResNet [41] have reach so good performance at image classification that the problem is often considered a solved problem [1]. CNNs also play a central roll in most modern object detection DNN architectures and are therefore more thoroughly explained here.

2.6 Convolutional Neural Networks

A CNN is a specialized type of DNN mostly used when the input data has a grid-like structure, such as time-series data or image data [36, p.330]. Although a CNN is capable of processing different kinds of input data, in the context of this thesis report inputs are assumed to consist of images represented by matrices with pixels values (or tensors for color images).

The structure of a CNN is similar to any regular DNN, with trainable weights and biases, weighted sums over neuron inputs with computed outputs trough activation functions, and a problem specific loss function. The distinctive difference between regular DNNs and CNNs lays in two CNN-specific layers called convolution layers and pooling layers.

2.6.1 Convolution Layers

Just like regular fully connected layers in DNNs, convolutional layers in a CNN can be seen as automatic feature extractors. The input of the first layer will be a complete matrix of image pixels, which in theory could be fed to a fully connected layer like in a regular DNN. However, due to the high dimensionality of an image, this would be very computationally expensive and unsustainable if the number of used hidden layers grows. Therefore, instead of processing single pixels, a convolutional layer takes in square patches of pixels and passes them through a kernel[3]. The kernel is convolved across the image and outputs a feature map (see figures 2.7 and 2.11). Mathematically, the filtering operation performed by a feature map is a discrete convolution, hence the name. Each layer can be composed of multiple kernels, each with individual trainable weights (feature extractors) [36, p.327]. If a layer consists of more than one kernel, the output of the layer will be a multidimensional tensor of feature maps (see figure 2.11).

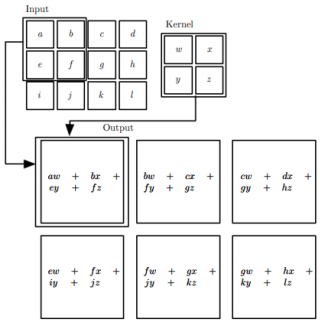

FIGURE 2.7: An example of a convolutional operation. *Figure from:* [36, p.334]

Another big difference with CNNs, compared to traditional DNNs where weights and biases of neurons are independent of each other, is that in a CNN the neurons corresponding to the same kernel in a layer share all weights and biases. The effect of this parameter sharing comes with some advantages. First, for data represented by matrices or tensors, such as images, local groups of pixel values are often highly correlated, forming distinctive local features which can easily be extracted. Second, the local statistics of images and other signals are invariant to location. For example, edges are usually detected in the early convolutional layers of a CNN (see figure 2.8). Because similar edges are found throughout the entire image, sharing parameters across different positions of the image is usually a good approach [36, p.331-335].

[3]Other commonly used names for the kernel are *convolutional filter*, *mask* and *convolution matrix*.

The computational cost, which is decreased thanks to parameter sharing used in convolutional operations, can be further reduced by increasing the step length (stride) of the kernel convolution. By utilizing a stride higher than one, the convolutional operation is not applied at each pixel location. This decrease in the number of convolutional operations results in fewer computations in the layer and also produces a smaller feature map which is used as input for the next layer [42].

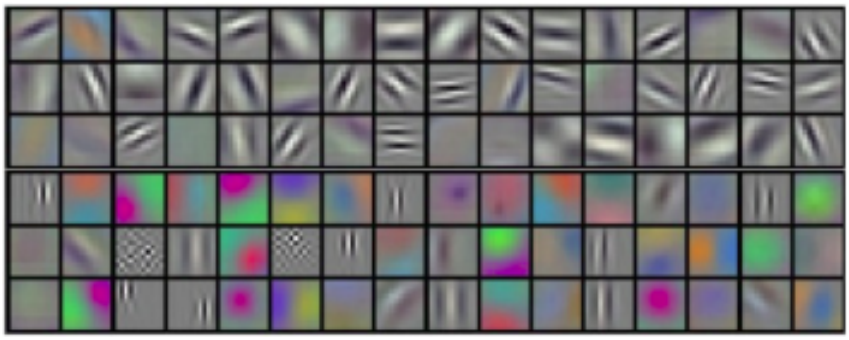

FIGURE 2.8: Visualization of features in the 96 convolutional kernels learned by the first convolutional layer in the famous AlexNet by Krizhevsky et al. *Figure from:* [32]

2.6.2 Pooling Layers

A pooling layer in a CNN can be perceived as a kind of down-sampling function. The function takes an activation map as input and outputs a summary statistic of nearby values from the input grid. Although there exist different functions to implement pooling among, the most common one is max pooling which applies a max filter to non-overlapping subregions of the initial representation (see figure 2.9) [44].

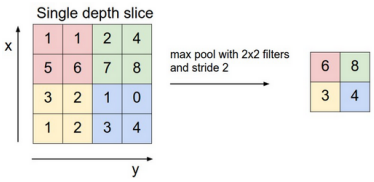

FIGURE 2.9: Max pooling using a stride of 2 and a 2×2 filter. *Figure from:* [44]

Contrary to the convolution operation, pooling uses no parameters. It slides a window over its input and selects the max value contained by the window. Similar to a convolution, the window size and stride are decided when designing the network. The most commonly used pooling layers in CNNs use filters of size 2×2 applied with a stride of 2 [44]. Pooling layers are often inserted in between convolutional layers periodically to reduce the number of parameters in the following layers. This trimming of trainable parameters decreases the computational cost, provides some invariance to rotations and translations and helps to fight overfitting. A pooling layer down-samples each feature map independently, by reducing the height and width, but keeping the depth intact [44].

2.6.3 CNN Architecture

Both pooling layers and strided convolution layers will increase the receptive field[4] of neurons in following layers. Therefore, at deeper layers, patterns processed by the kernels become more abstract and differs more from visual patterns we recognize as

[4]The receptive field of a neuron is the spatial extent of a neurons connectivity to the input image.

humans [36, p.337]. This effect can be seen in figure 2.10 where layer activations are visualized for a small image classification CNN.

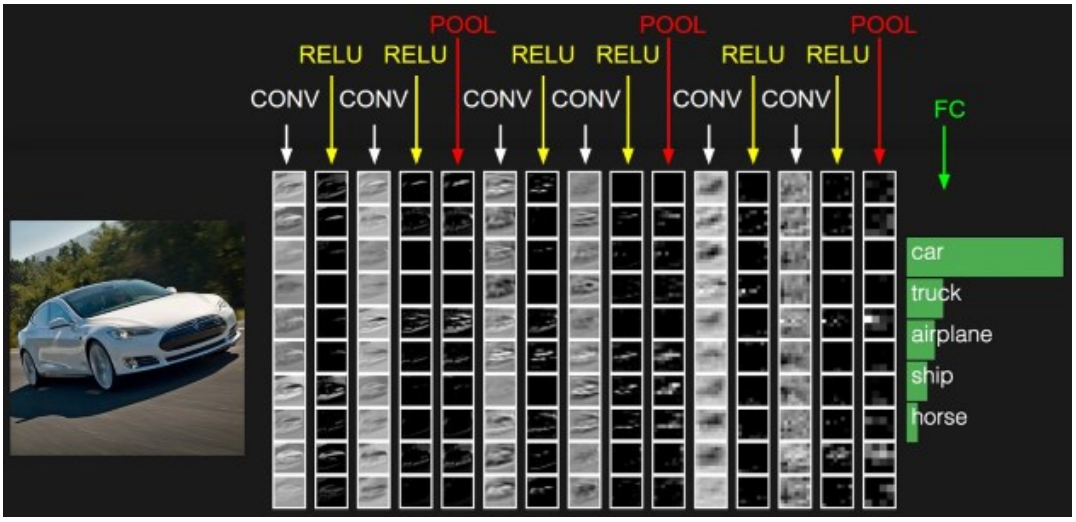

FIGURE 2.10: Layer activations visualized in a small CNN. *Figure from:* [44]

A standard image classification CNN will process each input image by passing them through a series of convolution layers with filters, pooling layers, fully connected layers, and lastly a loss function on the last fully-connected layer used for class predictions (see figure 2.11).

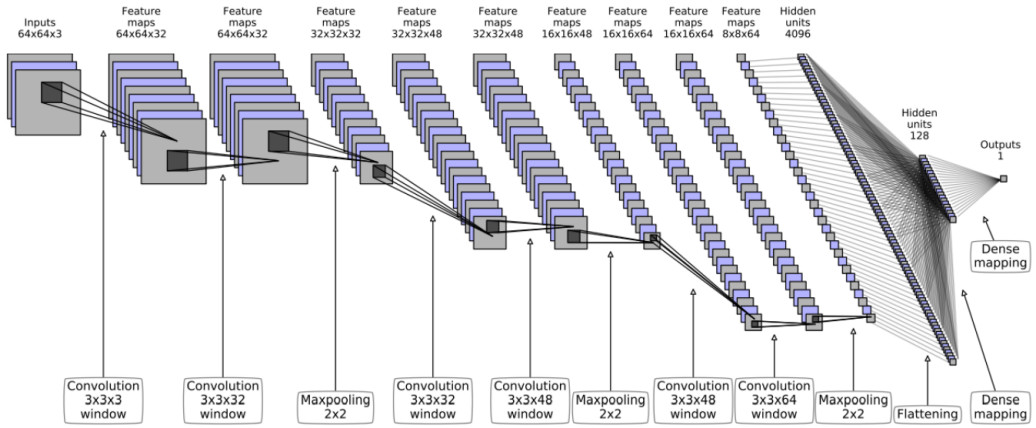

FIGURE 2.11: Visualization of the architecture of a deep convolutional neural network (CNN). *Figure from:* [45]

2.7 Object Detection

To achieve better image understanding, it is not only needed to recognize categories of object instances, as in object classification, but also spatially locate them. This task is commonly called object detection [14]. Object detection networks take images as input, and most commonly outputs bounding boxes together with class labels for all objects of interest, as shown in figure 2.12. Object detection using bounding boxes is

closely related to semantic image segmentation, which aims to assign each pixel in an image to a semantic class label.

Naturally, object detection is a much harder task than merely classifying images into distinct classes, and there will always be a trade-off between accuracy and efficiency when developing object detecting networks. It is also important to notice the increasing complexity and difficulty that comes from adding additional classes of objects to be detected.

The introduction of AlexNet did not only launch an explosion of interest in CNNs, but it also had a significant impact on the development of different kinds of object detectors that utilize the beneficial and useful characteristics of CNNs. Most DNN based object detectors use a CNN architecture as a backbone network where they utilize features from the top layer of the CNN as object representation [2].

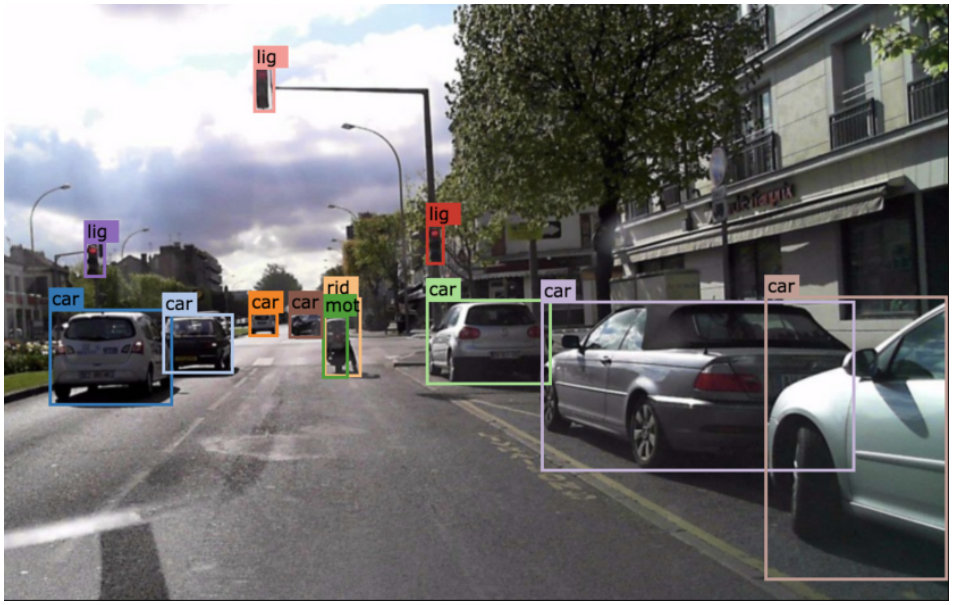

FIGURE 2.12: Visualization of ground truth bounding boxes for an example image in the BDD100K dataset. *Figure from:* [12]

However, the trade-off between accuracy and efficiency when developing these object detection frameworks has divided most of the current research into two categories. The first one uses a more traditional method which utilizes region proposals as a first step and follows by classifying each of the proposals into corresponding categories. The second method treats the problem as a regression problem using sampling over different locations, scales and aspect ratios and is all done adopting a unified network to achieve the final result. The first more traditional approach is generally more accurate but slower and is often referred to as a two-stage method, while the second approach typically results in a faster but less precise model and is commonly called a one-stage method [2]. A more detailed analysis and explanation of the one-stage object detection DNN used in this thesis project can be found in section 2.11.

2.8 Visual Perception for Autonomous Vehicles

When we drive, we consistently pay attention to our environment, look around for potential obstacles, whether they be vehicles, pedestrians, or objects on the road. Visual perception using deep object detection systems could be used by autonomous

vehicles, together with online path planning, to avoid obstacles and detect locations of free space where the car can safely drive. Although self-driving cars use additional technologies like Radar, LiDAR, and GPS, using cameras has many advantages. Some of these advantages are that cameras are cheap, they have a higher resolution than any other used sensors, and there are many more available datasets compared to other sensors [4]. However, there also exist some downsides — cameras are bad at depth estimation, and they are not reliable in extreme weather situations.

In the past, more traditional ML techniques have been utilized by autonomous vehicles when using camera imagery. In contrast to DNNs, which have only been around for a couple of years, more traditional ML techniques focus on color spaces, gradients and edges (*e.g.*, Histogram of Orientation (HOG), Scale-Invariant Feature Transform (SIFT)), with subsequent classification by Support Vector Machines (SVM) [40, p.658-685]. However, these techniques require hand-crafted features that are difficult and inconvenient to define. With the rise of deep learning, CNNs have automated most of these tasks while substantially boosting performance, and are therefore increasingly being used by self-driving systems [3].

Something that is important to have in mind when designing object detecting DNNs to be used by visual perception systems in autonomous vehicles is the frame-rate of the detector. The frame-rate, or how many frames per second (FPS) the DNN can process, is directly related to the vehicle's stopping distance, and other maneuvering actions. Therefore, to achieve feasible vehicle control, real-time frame-rates are desirable.

2.9 Real-time Object Detection for Autonomous Vehicles

As mentioned before, there are two main DNN architectural types for object detection: region-based networks (two-stage), and regression-based networks (one-stage). In two-stage frameworks, the first step consists of category-independent region proposals, followed by CNN feature extraction from these regions. In the second step category-specific classifiers are used to determine the category labels of the proposals. Most two-stage networks produce thousands of region proposals at test time, which comes with a high computational cost. The fastest two-stage object detector networks today are Faster R-CNN and R-FCN, which are capable of processing images at approximately 5-6 FPS [35, 46].

In contrast to two-stage object detectors, one-stage networks directly predict class probabilities and bounding box offsets from full images with a single feed-forward CNN network. This simpler and more elegant approach eliminates region proposal generation and subsequent feature resampling stages and enables the network to be optimized end-to-end directly on detection performance. Although this detection performance optimization comes with a slight decrease in accuracy compared to two-stage networks, one-stage networks often claim to have real-time object detection capabilities [5, 6, 8, 9, 10, 11].

One important consideration to have is that the term *real-time* do not come with a formally defined time limit or frame-rate. Instead, a real-time computing system is often described as a system that guarantees a response before a previously set *deadline* [47, p.4]. For a real-time visual perception system used by autonomous vehicles, this deadline could, for instance, be defined based on the frame rate of the camera, or the distance which detected objects have time to travel between two frames. For example, 10 FPS is often sufficient for tracking pedestrians on a sidewalk, but probably not for tracking cars going 100 kilometers per hour on a highway, or detecting a

child suddenly appearing in front of the vehicle. Thus, using current two-stage networks is not an option for the majority of scenarios which self-driving vehicles are exposed for. In the context of this thesis report, a real-time object detection system should be able to operate at frame rates above around 30 FPS, which is a common lowest frame rate for modern digital cameras.

The main two-stage based frameworks today includes Faster R-CNN [35], R-FCN [46], FPN [48] and Mask R-CNN [49]. DNN models in the forefront of object detection using one-stage methods mainly includes different versions of YOLO [5] and SSD [6]. These regression-based networks all have similar architectural types and comparable performances. Therefore only one of these models, the original Single Shot Multibox Detector (SSD), will be implemented, tested, and thoroughly reviewed in this thesis report. More information about other one-stage networks can be found in YOLO [5], YOLO9000 [8], YOLOv3 [9], RefineDet [10] and RetinaNet [11].

2.10 Virtual Environments

With the rise of deep learning, data has proven to be both the limiting and driving factor when training DNNs, particularly within CV [13, 14, 15, 16]. Creating diverse real-life datasets for driving scenarios is both time-consuming and expensive. The data has to be collected in different large-scale complex traffic systems, varying weather conditions, and numerous different geographic areas. The collected data then also has to be manually labeled, which is very labor-intensive. Moreover, manually annotating images is error-prone and highly subjective. Therefore, the resulting labels most likely will differ to some degree from the ground truth, and may affect the performance of neural networks trained on this manually labeled data [18].

FIGURE 2.13: Example of a real-world image and a corresponding virtual image from the Virtual KITTI dataset [17], with detected objects (cars) marked with bounding boxes.

One alternative to manually collected and labeled data for self-driving scenarios is to use photorealistic virtual environments to construct synthetic datasets (see figure 2.13). These datasets could be automatically assembled and annotated with much higher speed and accuracy, and thereby possibly help to solve the big deficit in labeled data for self-driving systems. Virtual environments could also add more

variance and uncertainty to a dataset to increase the responsiveness of the trained object detectors. It is easy to produce a variety of scenes that would be harder to test in real-life like severe weather, extreme traffic environments, and various high-risk scenarios.

At this moment a number of different increasingly realistic virtual worlds are being built (*e.g.*, Carla, Deepdrive , Virtual KITTI, AirSim, Truevision, ParallelEye, SynCity, Parallel Domain). This means that, in the near future, an enormous amount of automatically annotated data could be made available from such virtual worlds. Thus, if real-time object detectors trained on virtual images is shown to have comparable performance to real-time object detectors trained on real-life collected data, or if synthetic data can be used as complementary data during training, virtual environments could play a big part in the future development of autonomous driving.

2.11 Single Shot MultiBox Detector

Single Shot MultiBox Detector (SSD) is the cutting-edge real-time object detection deep neural network which is implemented and trained on driving-scene data in this thesis project. SSD is a one-stage object detection network which, provided an input image, uses a feed-forward CNN approach to produce a predefined number of bounding boxes and confidence scores for objects in the given image. The early layers in SSD are based on a pretrained classification network (VGG-16 in the paper, but any high-quality image classifier could be used), called the base network (see figure 2.14), truncated prior to the layers used for classification.

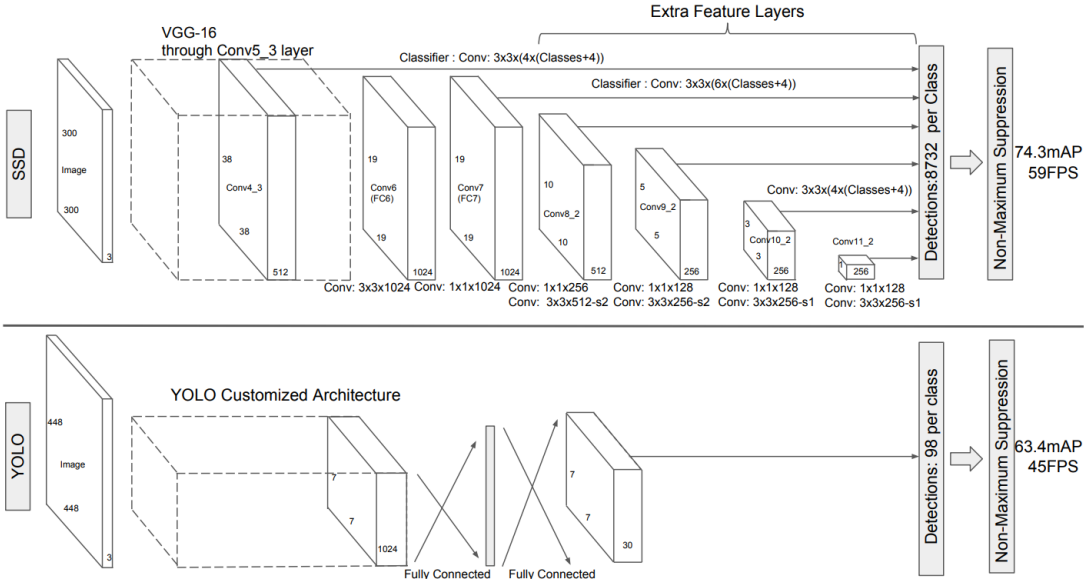

FIGURE 2.14: A comparison between the two main one-stage object detection networks: SSD and YOLO. SSD applies anchor boxes in multiple layers, including the end of the base network, instead of just applying them in the last layer like YOLO and other object detection networks. *Figure from:* [6]

To predict object bounding boxes and scores for the presence of object class instances in those boxes, SSD uses something the authors of the paper call *default*

boxes[5]. These prior bounding boxes are set at a number of predefined locations, or cells, in the image (see figure 2.15). The shapes of the anchor boxes should be based on the distribution of the ground truth object boxes in the dataset as a way to use prior knowledge of the problem when designing the network. For each cell, the network makes a small set of predictions composed of a bounding box together with confidence scores for each class (one extra class is added for *no object*). For each bounding box, the class with the highest predicted score is kept as the class for the bounded object. Lastly, the predicted boxes are matched to ground truth boxes, and the ones with an IoU overlap (see section 3.1) higher than a set threshold (0.5 in the paper) are kept. This approach simplifies the learning procedure and allows the network to predict high scores for multiple overlapping boxes, instead of being restricted to only choose one winning bounding box decided by the highest IoU.

The technique of using anchors is commonly used by many other object detection networks with slight variations. The aspect that distinguishes SSD from the rest is that the network utilizes this approach in multiple layers as shown in figure 2.14, instead of just applying it on the last layer. A typical CNN architecture uses decreasingly sized feature maps as the network gets deeper while the number of feature maps used at each layer increases (see figure 2.11). As described earlier in section 2.6.3, this layer depiction results in shallow layers with smaller receptive fields and deeper layers having larger receptive fields with more abstract representation. Therefore, SSD can use shallow layers to predict smaller objects and more deep layers to predict larger objects.

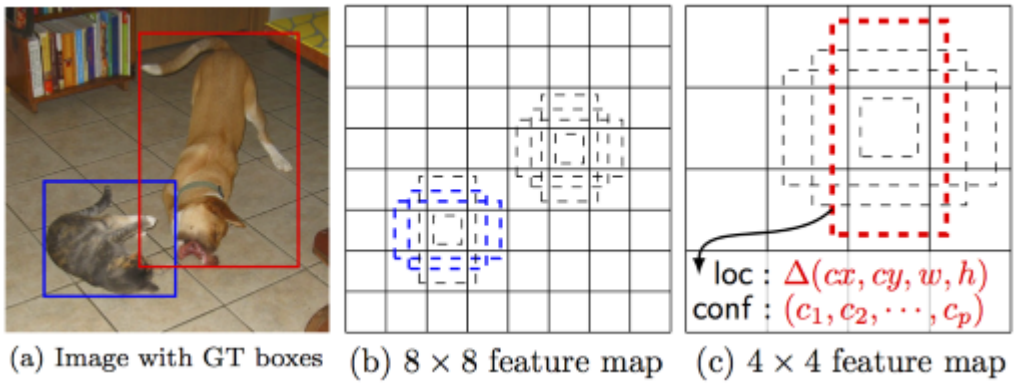

FIGURE 2.15: SSD framework. (a) SSD uses training data consisting of input images with corresponding ground truth boxes for each object. A small set of anchor boxes with varying aspect ratios are evaluated at each location in several layers in the network. This approach utilizes SSD to find objects of different sizes due to the decrease in feature map scales in deeper layers (*e.g.*, 8×8 and 4×4 in (b) and (c) respectively). In this example, two anchor boxes are matched with the cat and one with the dog. These matches are treated as positive matches and the rest as negative. *Figure from:* [6]

For example, in the original SSD architecture, the first set of prediction feature maps is taken from layer 23 in the VGG base network and consists of 512 feature maps with size 38×38 (see figure 2.14). Each point in all 38×38 feature maps covers a part of the image, and the 512 channels represent different features at every position. Using these features, classification to predict class labels and regression to

[5]Faster R-CNN and YOLO calls default boxes for *anchors* or *anchor boxes*, which are the terms that are going to be used in this thesis.

predict bounding box coordinates are conducted at each point in the 38×38 grid of the input image. The second prediction layer consists of 1,024 feature maps with size 19×19. This layer is capable of predicting slightly larger objects, as the points of the features cover bigger receptive fields. As can be seen in figure 2.14, and visualized in 2.16, this shrinking of feature map sizes continues down to the last layer consisting of only one point feature maps, which is ideal for detecting bigger objects.

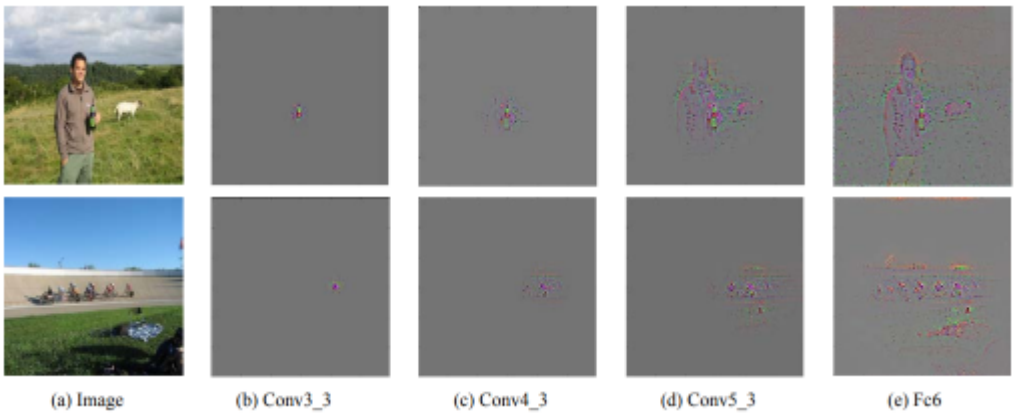

FIGURE 2.16: A visualization of the effective receptive fields in different layers of the SSD architecture. *Figure from:* [50]

Even if applying predictions with anchor boxes in shallow layers helps to predict smaller objects, SSD still suffer from the common one-stage object detection problem of detecting very small objects [5, 6, 10, 11]. To help improving the performance on datasets with small objects, SSD implements random cropping and a special data augmentation technique the authors of the paper calls a 'zoom-out' operation. This operation is conducted by randomly placing an image on a canvas of 16 times the size of the original image, which is filled out with mean values, before doing any random crop operation. They call it a 'zoom-out' operation because random cropping can be seen as a kind of 'zoom-in' operation. These extra augmentation operations increases the performance of SSD with 2%-3% according to the paper.

The loss function used by SSD is a weighted sum between localization loss (regression) and confidence loss (classification):

$$L(x, c, l, g) = \frac{1}{N}\left(L_{conf}(x, c) + L_{loc}(x, l, g)\right) \tag{2.16}$$

Here, N is the number of matched anchor boxes, x is 1 if the anchor box is matched to a determined ground truth box and 0 otherwise, l is the predicted bounding box parameters, g the ground truth bounding box parameters, and c is the class confidence.

The localization loss L_{loc} in 2.16 is given by:

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^{p} L1^{smooth}(l_i^m - \hat{g}_j^m), \tag{2.17}$$

in which

$$L1^{smooth}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

$$\hat{g}_j^{cx} = \frac{g_j^{cx} - d_i^{cx}}{d_i^w}, \quad \hat{g}_j^{cy} = \frac{g_j^{cy} - d_i^{cy}}{d_i^h}, \tag{2.18}$$
$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right), \quad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right).$$

Here, x_{ij}^p indicates if the i-th anchor box matched the j-th ground truth box of class p. Offsets to anchor box d are given by (cx,cy), w, and h, denoting distance to center, width offset, and height offset, respectively.

The confidence loss L_{conf} in 2.16 is given by:

$$L_{conf}(x, c) = -\sum_{i \in Pos}^{N} x_{ij}^p \log(\hat{c}_i) - \sum_{i \in Neg} \log(\hat{c}_i^0), \tag{2.19}$$

where

$$\hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}. \tag{2.20}$$

SSD predictions are set as being positive matches or negative matches depending on the set IoU threshold (see section 3.1). If the corresponding anchor box and ground truth box has an IoU greater than the threshold, the match is positive. Otherwise, it is negative. During training, as a majority of the predicted bounding boxes will have low IoU values, there will be a unbalance between positive and negative predictions. Although negative samples are needed to make the network learn what constitutes incorrect predictions, the ratio between positive and negative examples are set to be approximately 1:3.

FIGURE 2.17: Non-maximum suppression first discards boxes with low confidence scores and then disposes boxes among the remaining ones with high IoU overlaps. *Figure from:* [51]

The SSD approach described above produces 8,732 predicted bounding boxes for each class, where a significant fraction of those refer to the same objects (see figure

2.17 for an example). Although this is fine during training, at test time these duplicate predictions pointing to the same objects need to be pruned to one per predicted object. This pruning is done using Non-Maximum Suppression (NMS) which first discards boxes with low confidence scores (0.01 in SSD paper) and then disposes boxes among the remaining ones with high IoU overlaps (0.45 in SSD paper). Using NMS guarantees that high probability predictions are retained by the network, while the noisier ones are removed.

One of the main features of SSD is that it utilizes the varying receptive field sizes in different feature maps within the network for detecting objects of different sizes. Shallow layers in the network with smaller receptive fields detect smaller objects, and deeper layers with larger receptive fields detect larger objects. The anchor boxes used by SSD are predefined with aspect ratios $a_r \in \{0.5, \frac{1}{3}, 1, 2, 3\}$ and scaled for each feature map to relate to the corresponding receptive fields. In other words, each specific feature map will be responsive to particular scales of the objects in the dataset. Using m feature maps responsible for making predictions, the anchor box scales for each feature map layer k is given by:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), \quad k \in [1, m] \tag{2.21}$$

where s_{min}, s_{max}, are the scales used in the first and last layers respectively, and in-between layer scales are spaced regularly (these scales are set to $s_{min} = 0.2$ and $s_{max} = 0.9$ in the paper[6]). The width w_k and height h_k of the anchor boxes in a specific layer k is then given by:

$$w_k^a = s_k\sqrt{a_r}, \quad h_k^a = \frac{s_k}{\sqrt{a_r}} \tag{2.22}$$

For the aspect ratio of 1, an extra anchor box is added whose scale is $s'_k = \sqrt{s_k s_{k+1}}$, which results in 6 anchor boxes per prediction layer. The center of each anchor box is set to $(\frac{i+0.5}{|f_k|}, \frac{j+0.5}{|f_k|})$, where $i, j \in [0, |f_k|)$, and $|f_k|$ is the size of the k-th square feature map.

In the paper, aspect ratios, a_r, and scales, s_{min}, s_{max}, are set to specific values ($a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$ and $s_{min} = 0.2$, $s_{max} = 0.9$), but in practice, these can be chosen to fit the distribution of ground truth bounding boxes in the training dataset.

[6]Scaling factors $s_{min} = 0.2$ and $s_{max} = 0.9$ were used for the main training on the PASCAL dataset. SSD was also trained on the COCO dataset [15] in the paper using other dataset specific hyperparameter settings for the scaling factors.

Chapter 3

Problem Formulation

In this thesis, the problem of object detection in digital images using deep neural networks will be studied. Specifically, the focus will be on real-time capable deep neural networks and their viability for self-driving systems. Self-driving systems are commonly categorized into three subsystems: perception, planning, and control [4]. The perception system is responsible for translating raw sensor data into a model of the surrounding environment, the planning system makes purposeful decisions based on the environment model, and the control system executes planned actions. The objective of this thesis is to study the perception problem in the contexts of object detection for autonomous vehicles using a representative cutting-edge real-time object detection deep neural network architecture called Single Shot MultiBox Detector (SSD).

The goal for this thesis is to answer the following research questions:

- How well does a state-of-the-art real-time capable object detecting deep neural network perform when trained on a challenging and diverse driving-scene dataset like Berkeley Deep Drive 100k (BDD100K)?
- Is the performance of real-time capable object detecting deep neural networks good enough to be able to be utilized in self-driving systems?
- How big is the performance gap between real-time capable object detecting deep neural networks and the top performing (but slower) networks trained on the BDD100K dataset?
- Is the performance of a real-time capable object detecting deep neural network trained on synthetic driving-scene data comparable with the same network trained on real driving-scene data? If not, what is the gap between available real and synthetic driving scene data, from the perspective of object detection architectures?

To be able to answer these questions, the SSD network will be implemented as it is presented in the SSD paper [6], using a neural networks library named Keras in the programming language Python. The deployed network will be trained and evaluated on different driving-scene datasets (see section 4.3), where the evaluation is performed using the performance metrics described in section 3.1. Additionally, the thesis will also result in a review on the background and theory of object detection networks and computer vision for self-driving systems, including a specifically systematic review given for the SSD network.

3.1 Performance Evaluation and Documentation

Evaluating an object detecting system is a non-trivial task since there are two distinct functions to measure: determining whether an object is present in the image (classification), and detecting the location of the object (regression).

The two most common metrics used for bounding box localization and classification tasks when working with object detection are Intersection over Union (IoU) area and Average Precision (AP), respectively [13, 14, 15].

IoU corresponds to the fraction of the intersection between a predicted box B_{Pred} and the corresponding ground truth box B_{GT}:

$$\text{IoU} = \frac{|B_{Pred} \cap B_{GT}|}{|B_{Pred} \cup B_{GT}|} = \frac{|B_{Pred} \cap B_{GT}|}{|B_{Pred}| + |B_{GT}| - |B_{Pred} \cap B_{GT}|}, \tag{3.1}$$

or in other words, the area of overlap between the two bounding boxes (see figure 3.1).

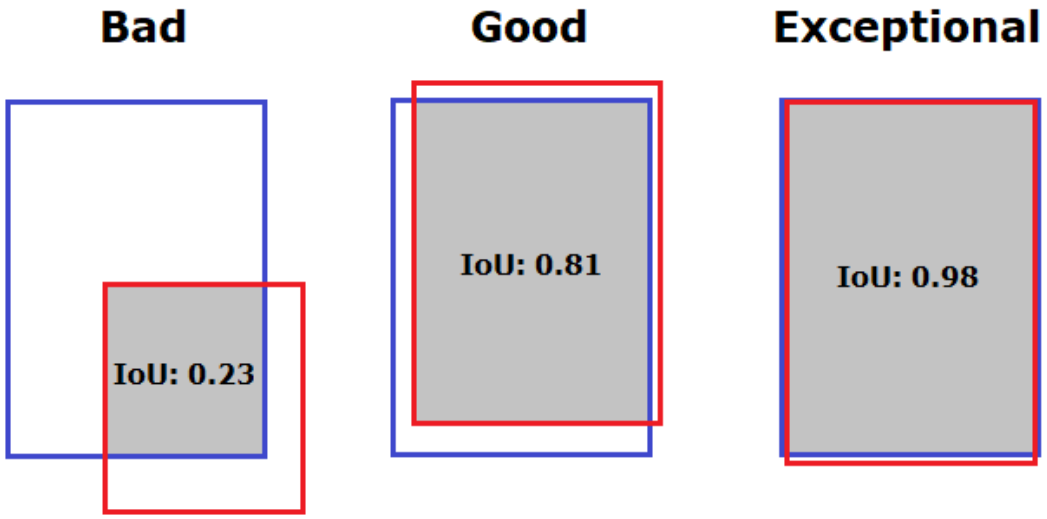

FIGURE 3.1: IoU examples for three bounding boxes.

AP is an average value score of maximum precisions at different recall values in a *precision-recall* curve. *Precision*, P, is a measurement of the prediction accuracy, i.e., the percentage of correct positive predictions, and is defined as the number of true positives (correct predictions), T_P, over the sum between the number of false positives (incorrect predictions), F_P, and the number of true positives:

$$P = \frac{T_P}{F_P + T_P} \tag{3.2}$$

A precision score close to 100% indicates a high likelihood that positive detections, in fact, are accurate predictions. *Recall*, R, determines the prediction quality, or the fraction of true object detections compared to the total number of objects in the dataset. Recall is defined as the number of true positives over the sum between the number of false negatives (missed predictions), F_N, and the number of true positives:

$$R = \frac{T_P}{F_N + T_P} \tag{3.3}$$

A high recall score suggests that most objects in the dataset were positively detected. However, a system that have high recall together with low precision, will return

many detections, but most of those detections will be incorrectly labeled. If the situation is the opposite with high precision but low recall, the system will return few results, but most of the predicted labels will be correct. The ideal network has high precision and high recall which will result in many predictions, most of which are labeled correctly.

The precision-recall curve can be computed by varying the confidence score threshold that determines what is counted as a predicted positive detection of the class. Figure 3.2 shows two examples of precision-recall curves of different prominence for the SSD network trained on the PASCAL dataset (see section 4.3.4).

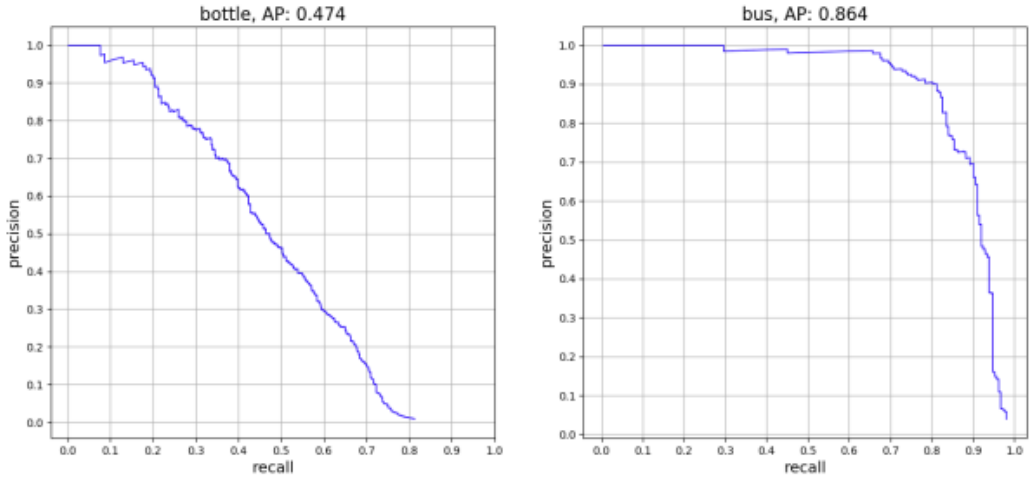

FIGURE 3.2: Two examples of precision-recall curves for the SSD network trained on the PASCAL dataset (see section 5.4.1).

There are different ways of computing AP using the precision-recall curve. One common way, which will be implemented in this thesis report, is the k-point sampling Pascal-VOC-style AP [14]. Here, AP is defined as the mean value of the precision at a set of eleven equally spaced recall levels, $R_i = [0.0, 0.1, 0.2, ..., 1.0]$. Thus,

$$\text{AP} = \frac{1}{11} \sum_{R_i} P(R_i), \tag{3.4}$$

where the precision at recall i is taken to be the maximum precision measured at a recall exceeding R_i. Looking at equation 3.4 and figure 3.2, it is clear that AP essentially is an approximation of the area under the precision-recall curve.

Since object detection tasks consist of both bounding box localization and classification, the metric commonly used for object detection is a combination of IoU and AP called mean Average Precision (mAP) [5, 6, 8, 9, 10, 11, 35, 46]. Mean average precision is simply the mean of the APs computed over all the classes of the task, for a fixed IoU threshold. Naturally, the mAP score will depend on the chosen IoU threshold. Therefore, the performance can be evaluated at different IoU values. Object detection networks are often evaluated using mAP with a IoU threshold of 0.5 [5, 6, 14, 35, 52], therefore most evaluations in this thesis report also will be conducted using a IoU threshold of 0.5. Precision-recall curves will be presented for each trained network to give a more refined and detailed illustration of network performances.

Chapter 4

Experimental Work – Preparations

In this chapter, everything regarding the preparations for the experiments is presented. These preparations consist of determining the most suitable platform for building and training the SSD network along with which deep learning framework and software libraries to use. A presentation of all datasets used in the experiments is also given together with an explanation of the data analysis and prepossessing process. The last section will be dedicated to explaining the implementation of a CNN using the chosen frameworks and software libraries.

The SSD network and its related functions and utilities are implemented as a combination of preexisting and self-programmed python code. All code used for prepossessing and data analyzing is original code for this thesis project. The SSD network is implemented using Keras (see section 4.2.2) with both self-written code and code that is inspired by the Caffe implementation of SSD made by the authors of the SSD paper together with various open sourced implementations. All code for the project is available in an online repository[1] and contain documentation about how the content was created.

4.1 Virtual Computing Platform

Training a deep neural network that involves compute-intensive tasks on extensive datasets can take weeks or even months using a single processor. However, if those tasks are instead run on a GPU, the training time can be reduced to hours or days instead of weeks. Using cloud computing for this task is a convenient way to manage large datasets, and it allows deep learning models to scale efficiently and at lower costs using GPU processing power. Currently, there are several options to chose from when it comes to picking a cloud provider, with the three biggest ones being Google, Amazon, and Microsoft. All three of these provide effective and versatile cloud platforms and infrastructures that can be utilized for deep learning tasks. The choice for this thesis project fell on Google's platform since they offer a trial period of one year including $300 worth of computing credits.

Google Cloud Platform (GCP) is a collection of computing services offered by Google which run on the same base framework as Google's end-user products such as Google Search and YouTube. The platform provides more than ninety products including computing, storage, analytics and ready to use machine learning solutions [53]. Although GCP offers practical and straightforward machine learning services, this thesis project requires a more specific solution for deploying and training deep neural networks on a high performing system. Therefore, the implementation and training of the network are performed on Google's virtual machine (VM) solution called Google Compute Engine (GCE). GCE allows users to set up and launch VM

[1] `https://gitlab.com/roger_cybercom/ssd-master-thesis`

instances on demand which can be accessed via a browser-based console, secure-shell (SSH), or a command-line interface. One big advantage with using cloud-based VMs like GCE is that they can be customized to users needs, including the choice of both operating system and hardware. A GCE instance can be launched with varying hardware specifications for memory, disc space, CPUs and GPUs.

4.2 Tools

Implementing a DNN or a deep learning framework from scratch using a programming language was never considered for this thesis project. This work would require such a significant amount of effort it puts it beyond the scope of the project. Thankfully, with the growing popularity in deep learning, many deep learning frameworks have recently been developed. These frameworks include some popular libraries such as Caffe, Theano, TensorFlow, PyTorch, and Keras. The fact that most of the available deep learning frameworks have APIs in python indicates that Python is one of the more adopted languages when working with machine learning. Therefore, the implementation of SSD will be utilized in Python, using Keras, with TensorFlow as a backend for handling low-level operations.

4.2.1 TensorFlow

TensorFlow is an open source software library used for tensor manipulation in machine learning, deep learning, and other scientific domains requiring computationally extensive numerical calculations. It was initially developed to be used by Google researchers and engineers but was later released as an open source software. TensorFlow is made to be flexible and easily deployable on different devices, using single or multiple CPUs, GPUs or TPUs. The library can be accessed using APIs in Python, C, and C++ [54]. Although TensorFlow can be used directly to build and train deep neural networks, it is sometimes more convenient to use a more high-level neural networks library built on top of TensorFlow.

4.2.2 Keras

Keras is a software written in Python used for building and training neural networks with a more high-level approach. All computationally extensive low-level operations are conducted by an underlying well-optimized tensor manipulation library, serving as the 'backend engine'. Keras is capable of using different backend implementations including TensorFlow, Theano, and Microsoft Cognitive Toolkit. Keras provides a way to build a neural network as a sequence, or a graph, of standalone, fully-configurable modules representing network layers, activations functions, loss functions, and different schemes of regularization and initialization [55].

4.3 Datasets

With the rise of deep learning, data has proven to be both the limiting and driving factor when training DNNs, particularly within CV where large visual datasets (e.g., ImageNet [13] and COCO [15]) have been the driving force behind recent advances with DNNs used for CV. These deep networks used for visual perception can require training datasets with millions of annotated images to achieve state-of-the-art performance. This need for massive amounts of data to be able to exploit

the potential of deep learning has shown to be an obstacle when developing DNNs for autonomous vehicle perception systems [12] due to the lack of diverse, large-scale annotated driving scene datasets. Most available driving scene datasets for autonomous driving today are not only limited in size, but also typically in scene variation, annotation richness, geographic locations, weather conditions, and daily temporal variations. Therefore, networks trained on existing driving scene datasets tend to overfit specific domain characteristics [12] and are not fit for general purpose self-driving systems. To address this problem Berkeley Artificial Intelligence Research (BAIR) recently (May 30, 2018) released the largest and most diverse driving scene dataset to this date named Berkeley Deep Drive 100k (BDD100K) [12].

4.3.1 BDD100K

BDD100K consists of hundred thousand images annotated with image level tagging, object bounding boxes, drivable areas, lane markings, and full-frame instance segmentation. However, for this thesis project only the bounding box annotated images are relevant (see figure 4.1).

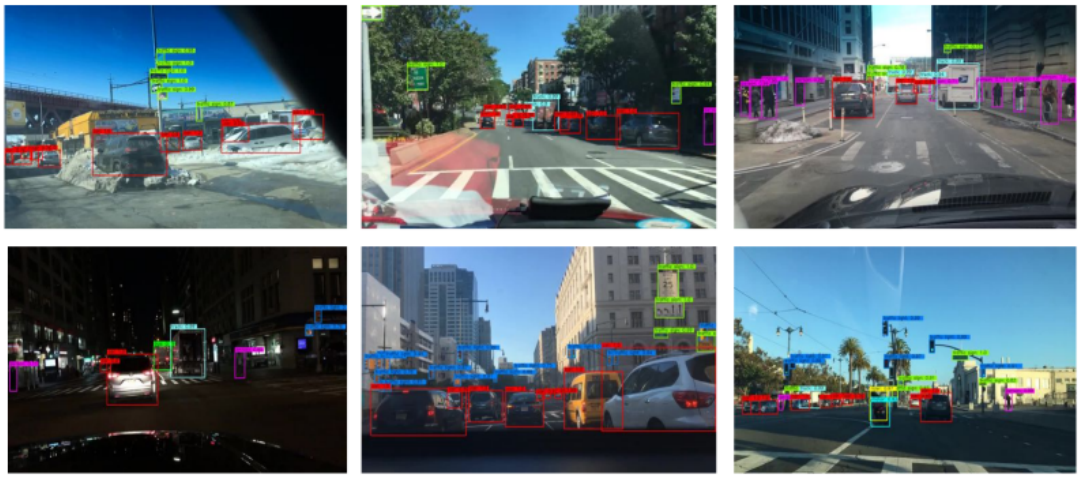

FIGURE 4.1: Example images from the BDD100K datasets [12] showing object bounding box annotations.

BDD100K is a challenging dataset, created to test the limit of object detectors, by reflecting a variety of driving scene conditions that a self-driving car could encounter today. The dataset was collected in New York, Berkeley, and San Francisco, across varying driving scenarios and covers different weather conditions, including sunny, overcast, and rainy, as well as images collected at both daytime and nighttime as shown in figure 4.2. This variety entitles BDD100K to be a good representation of the physical world a self-driving system would encounter. BDD100K has over 1.8 million bounding box annotated distinct objects with appearances and contexts from ten different classes (bus, traffic light, traffic sign, person, bike, truck, motorcycle, car, train, rider). Object count statistics for all classes in the dataset is shown in figure 4.3, and the distribution of bounding box sizes in figure 4.4.

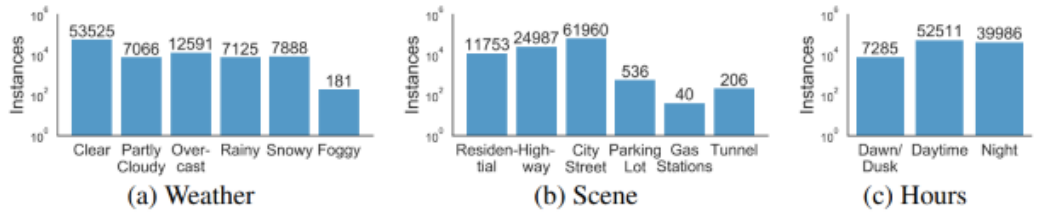

FIGURE 4.2: Distribution of image scenarios in BDD100K, showing distributions of images with different (a) weather, (b) scene, and (c) times of day. *Figure from:* [12]

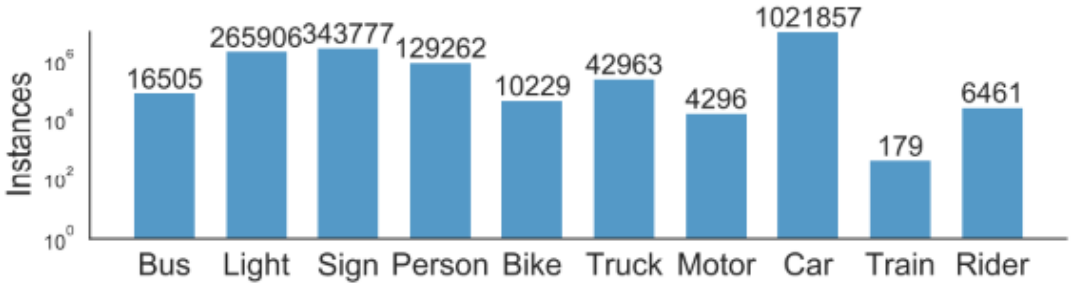

FIGURE 4.3: Object counts for the ten classes in all 100,000 images in BDD100K. *Figure from:* [12]

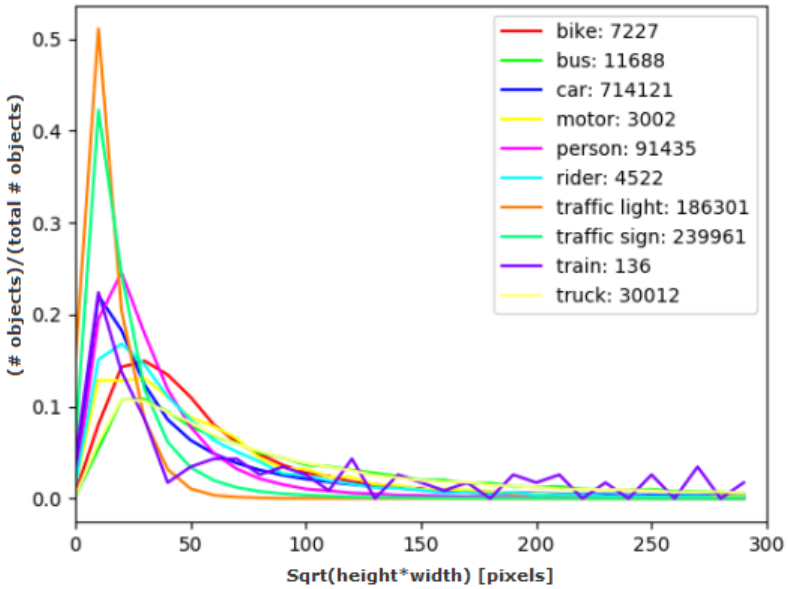

FIGURE 4.4: Distribution of object sizes for each class in the 80,000 images in BDD100K with released annotations. *Figure from:* [56]

Although the dataset consists of 100,000 images, 20,000 of these do not come with annotations. Instead, the annotations for these unlabeled images are kept by BAIR for evaluation of submitted object detection systems. In this thesis project, to be able to run multiple evaluations, 10,000 of the 80,000 labeled images are kept isolated as test images, 5,000 are used for validation, and the remaining 65,000 are used for training.

4.3.2 FCAV

Creating diverse real-life driving scene datasets like BDD100K is both time-consuming and expensive. The data has to be collected in different large-scale complex traffic systems, varying weather conditions, and numerous different geographic areas. The collected data then also has to be manually labeled, which is labor-intensive. Moreover, manually annotating images is error-prone and highly subjective.

One alternative to manually collected and labeled data for self-driving scenarios is to use photorealistic virtual environments to construct synthetic datasets. These datasets can be automatically assembled and annotated with much higher speed and accuracy, and also be produced with a variety of scenes that would be harder to do in real-life, like severe weather, extreme traffic environments, and various high-risk scenarios.

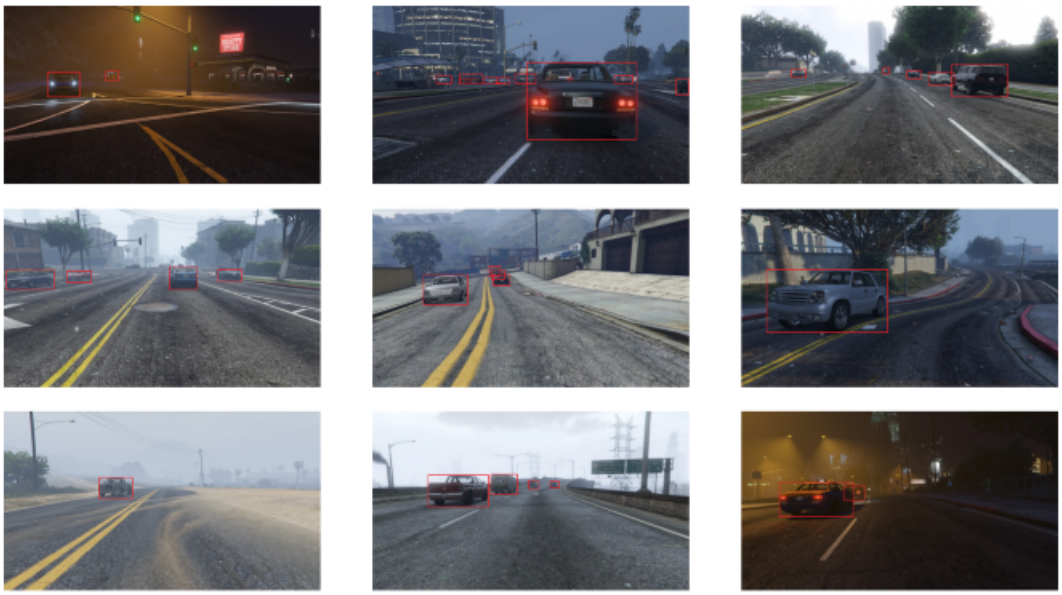

FIGURE 4.5: Example images from the FCAV synthetic driving scene dataset [19] showing bounding box annotations for cars.

FCAV [19] is an synthetic dataset collected in the game *Grand Theft Auto V* provided by a collaboration between the University of Michigan and Ford Center for Autonomous Vehicles. The dataset consists of 200,000 images collected in the game with bounding box annotations for all car instances (see Figure 4.5). Different weather scenarios are simulated, including sun, fog, rain and haze, as well as images simulated at day, night, morning and dusk.

FCAV will be used in this thesis project to investigate how large the gap is between real and virtual data. This comparison will be done by comparing the performance of a network trained on FCAV with a network trained on a subset of BDD100K including only images with car annotations. To make this comparison as equitable as possible, a subset of FCAV consisting of an equal number of images as BDD100K is constructed by randomly selecting 80,000 images from the complete set of images in FCAV. This subset is then split into a training set consisting of 65,000 images, a validation set consisting of 5,000 images, and a test set consisting of 10,000 images, in the same way as was done with the BDD100K dataset.

4.3.3 KITTI

To be able to quantitatively analyze the performance and make a fair comparison between the two networks trained on real and synthetic data, a third distinct test dataset called KITTI will be used for evaluation. KITTI is a driving scene dataset used for object detection, tracking, semantic segmentation, and optimal flow purposes [57]. It was collected around the city of Karlsruhe, Germany, by the Karlsruhe Institute of Technology (KIT) and Toyota Technological Institute of Chicago (TTI-C), and is composed of 7,481 training images and 7,518 testing images. However, only the 7481 training images will be used for comparing the performances of the two networks trained on BDD100K and FCAV since ground truth annotations for the KITTI testing set are not released to the public. Annotations for cars, pedestrians, and cyclists are provided in the dataset, but seeing that FCAV only have annotations for cars, the networks will only be evaluated on the car annotated images of KITTI (see figure 4.6).

FIGURE 4.6: Example images from the driving scene dataset KITTI [57] showing bounding box annotations for cars.

Using a completely new dataset for comparing the performance of the networks trained on real and synthetic data will help prevent any biases towards any of the networks. One could use only a isolated subset of BDD100K for testing which would only consist of images not used for training. However, this dataset would still have similarities with the training data in BDD100K (for example, similarities in lightning conditions, image resolution, object sizes, bounding box aspect ratios, geographical similarities, car models, etc.), and the evaluation would therefore most likely favour the network trained on BDD100K compared to the one trained on FCAV.

4.3.4 PASCAL

The PASCAL Visual Object Classification (PASCAL VOC) datasets were provided as part of the PASCAL Visual Object Classes challenge from 2005 to 2012. The PASCAL VOC2007 and VOC2012 datasets with bounding box annotations were used to train and evaluate the original SSD network. Therefore, these datasets will be used for ensuring that the implementation of SSD in Keras in this thesis project is an accurate representation of the original SSD network. Together, PASCAL VOC2007 and VOC2012 consist of 32,494 images with annotations for twenty different classes (airplane, bicycle, bird, boat, bottle, bus, car, cat, chair, cow, dining table, dog, horse,

motorbike, person, potted plant, sheep, sofa, train, TV monitor). This combined dataset was split into a training set consisting of 25,494 images, a validation set consisting of 2,000 images, and a test set consisting of 5,000 images.

FIGURE 4.7: Example images from the PASCAL datasets [14, 52] showing object bounding box annotations. *Figure from:* [58]

4.3.5 Annotations

Datasets that use bounding boxes for object detection consist of image files together with corresponding bounding box annotation files. Unfortunately, there is no convention on how to store annotations for bounding box coordinates. Therefore, object detection datasets come with a variety of annotation file formats. For example, the KITTI dataset (see section 4.3.3) uses plain text annotation files, where the annotation files have identical file names as the corresponding image files. Each line in such a text file consist of data for one object in the corresponding image separated with white spaces. KITTI annotation files do not contain any information about the data. Instead, to be able to use the information correctly, one has to refer to the documentation on their website. In contrast, the BDD100K dataset uses one big JSON (JavaScript Object Notation) file to store all annotation data where each object in the file correspond to a specific image. However, in this thesis project, the PASCAL VOC format will be used, and therefore, all datasets will be converted into this format. In PASCAL VOC format each image comes with a corresponding xml file containing all annotation data which makes it more convenient since all the information can be, and is, labeled within the xml file (this is also true when using JSON, but PASCAL VOC format gives more flexibility seeing that the annotation data is split into individual files).

4.4 Data Analysis and Preprocessing

The power of deep learning networks comes from their ability to learn patterns from large amounts of data. Thus, understanding the data used to train the SSD network implemented in this thesis project is critical to achieve as good performance as possible. The focus for the data analysis in this project will be on distributions, sizes, and aspect ratios for bounding boxes in the datasets. This is due to, as explained

in section 2.11, the importance of correct anchor box hyperparameter settings when building the SSD network.

Anchor boxes in SSD are defined by parameters for scaling factors (see equations 2.21 and 2.22) and specific aspect ratios in each of the six prediction layers. Finding the best values for these parameters would require one to train multiple versions of the network and then comparing the resulting performances. Seeing that training SSD takes several days, this is not an option for this thesis project. Instead, these parameter values have to be decided by analyzing the bounding box distributions, sizes, and aspect ratios in the datasets.

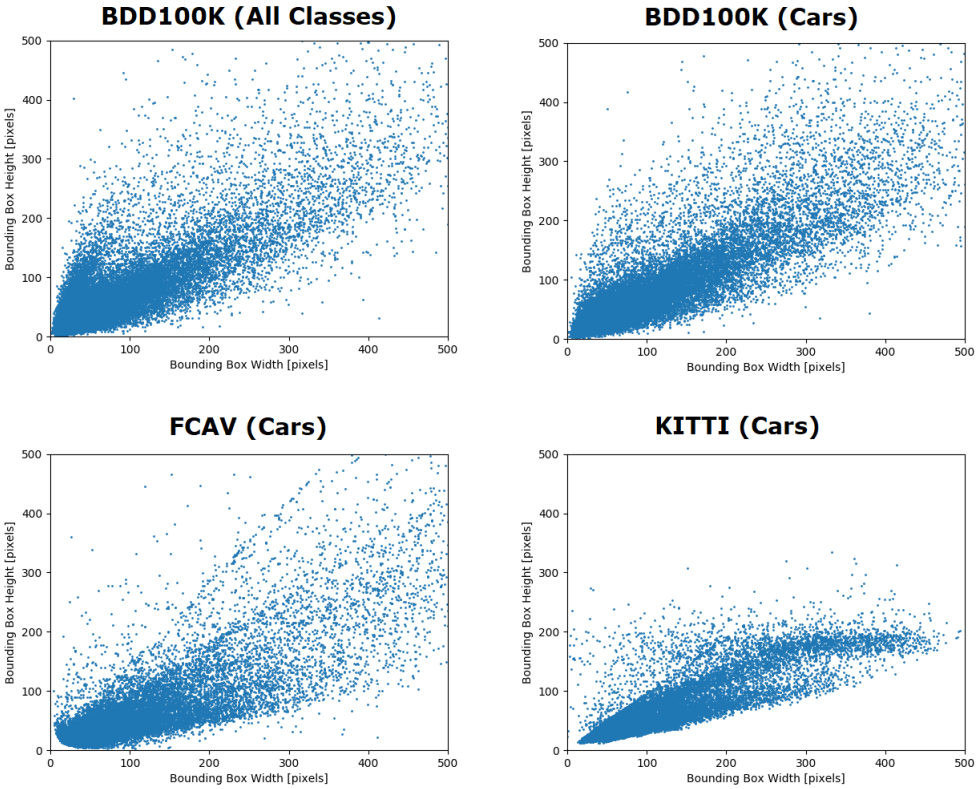

FIGURE 4.8: Scatter plots showing bounding box widths and heights in 5,000 random images from each dataset.

It is clear by studying figure 4.8 that the datasets consisting of only cars are over-represented by bounding boxes with aspect ratios over one[2]. It is only in the complete BDD100K dataset including all classes one can see a collection of bounding boxes with aspect ratios under one. This notion is something that gets even more clear when looking at figure 4.9 showing histograms for different bounding box aspect ratios for the different datasets. Note the spikes at various aspect ratios in figure 4.9. These spikes could possible be caused by automated annotation tools set to default aspect ratios when the labeling is performed. The SSD network uses five default aspect ratios for its anchor boxes. These five values are chosen as hyperparameters when building the network and should be carefully matched to bounding box aspect ratios in the training dataset.

The other hyperparameter corresponding to anchor boxes in SSD is a set of scaling factors which define all anchor box sizes. All prediction layers have a corresponding scaling factor parameter which is based on the input size of the network.

[2]A diagonal line from the origin with slope one would represent aspect ratios of one.

Each prediction layer has five anchor box aspect ratios which are used together with the scaling factor to define the anchor boxes of that layer (see section 2.11). For example, the original SSD network uses images with 300×300 pixels as input. If a specific anchor box has an aspect ratio of 2, and scaling factor of 0.1, the width, w, of the anchor box will be $w = 300 \cdot 0.1 \cdot \sqrt{2} = 42$ pixels, and the height, h, will be $h = 300 \cdot 0.1 / \sqrt{2} = 21$ pixels. The scaling factors for each prediction layer should be chosen based on the bounding box sizes in the training dataset (see figure 4.10).

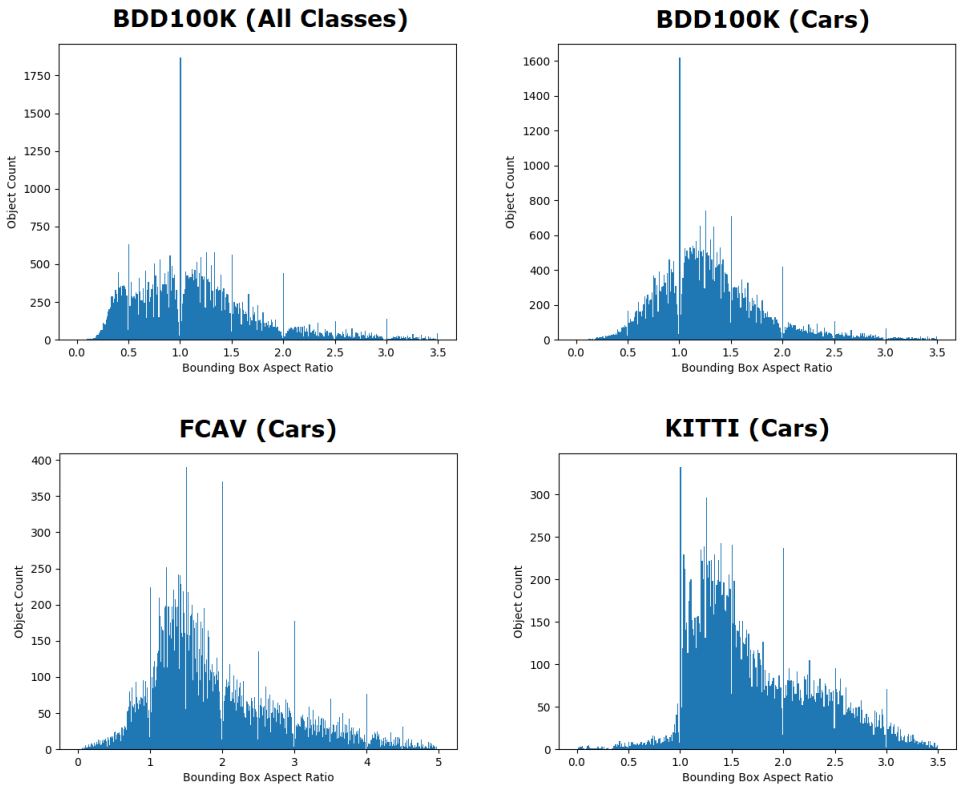

FIGURE 4.9: Histograms showing object counts for different bounding box aspect ratios in 5,000 random images from each dataset.

Another aspect that may differ between datasets is the positioning of bounding boxes throughout the dataset. This can be seen in figure 4.11 where bounding box centers for object instances are shown as image heatmaps for each dataset. This is something that could be taken advantage of by adjusting a hyperparameter in SSD which specifies the spacing between adjacent anchor box center points for each prediction layer together with another hyperparameter controlling the offsets of the first anchor box center points from the top and left borders of the image. However, this would be an rather inexact way of adjusting anchor box positions, and is therefore not used. Instead, the heatmaps in figure 4.11 can be used to get a clearer perception of the differences between the datasets.

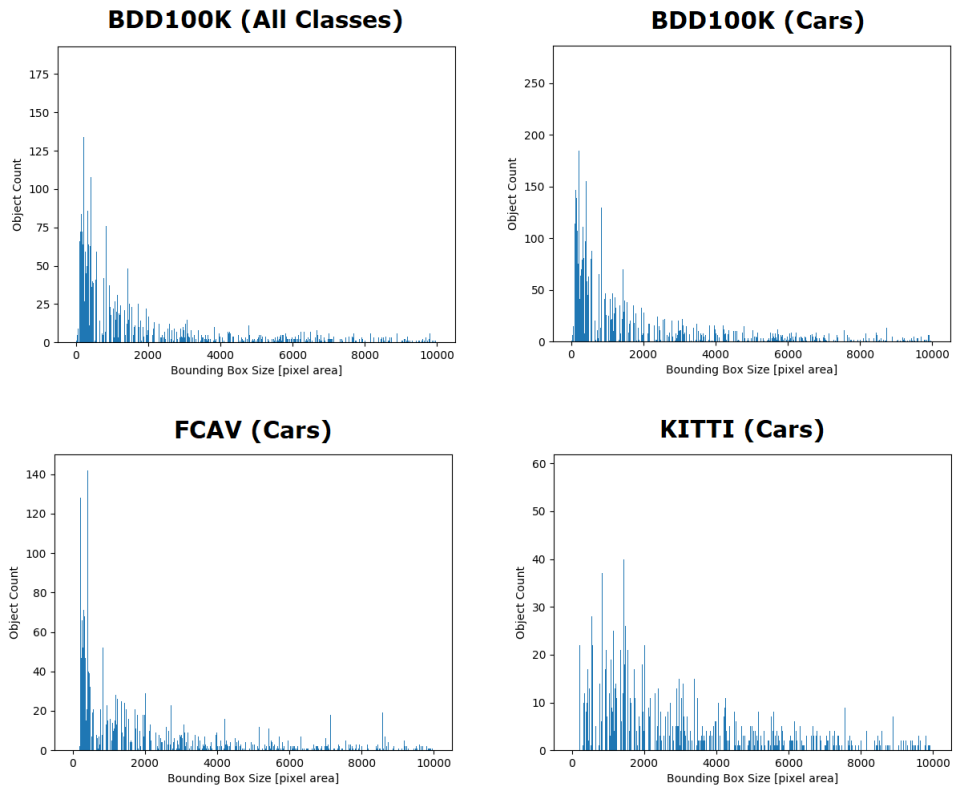

FIGURE 4.10: Histograms showing object counts for different bounding box sizes in 5,000 random images from each dataset.

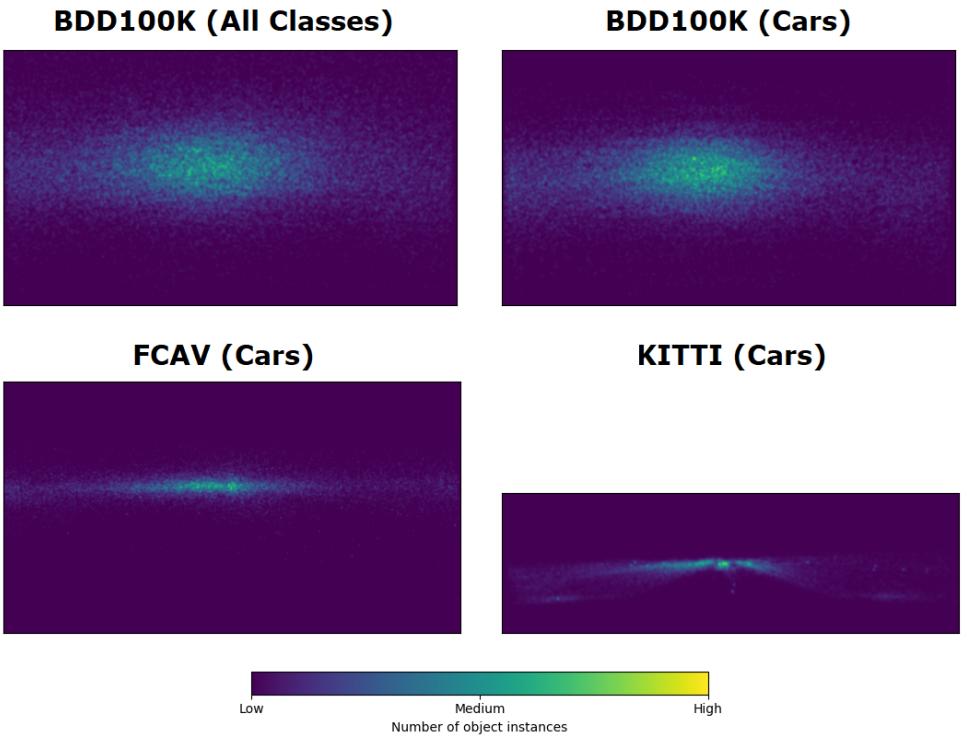

FIGURE 4.11: Image heatmaps showing bounding box centers for object instances in 5,000 random images from each dataset. Note the wider aspect ratios of images in the KITTI dataset.

The only preprocessing that will be performed on the data is *mean subtraction* to zero center the datasets. This centering is conducted by subtracting mean pixel values separately across the three color channels (see section 2.3.7) which can be seen in figures 4.13, 4.14 and 4.15. In figure 4.12 the collected per channel mean values are visualized for each dataset as a scatter plot in a RGB color space where each color channel is represented in Cartesian coordinates. This visualization also gives additional information about the differences in the datasets.

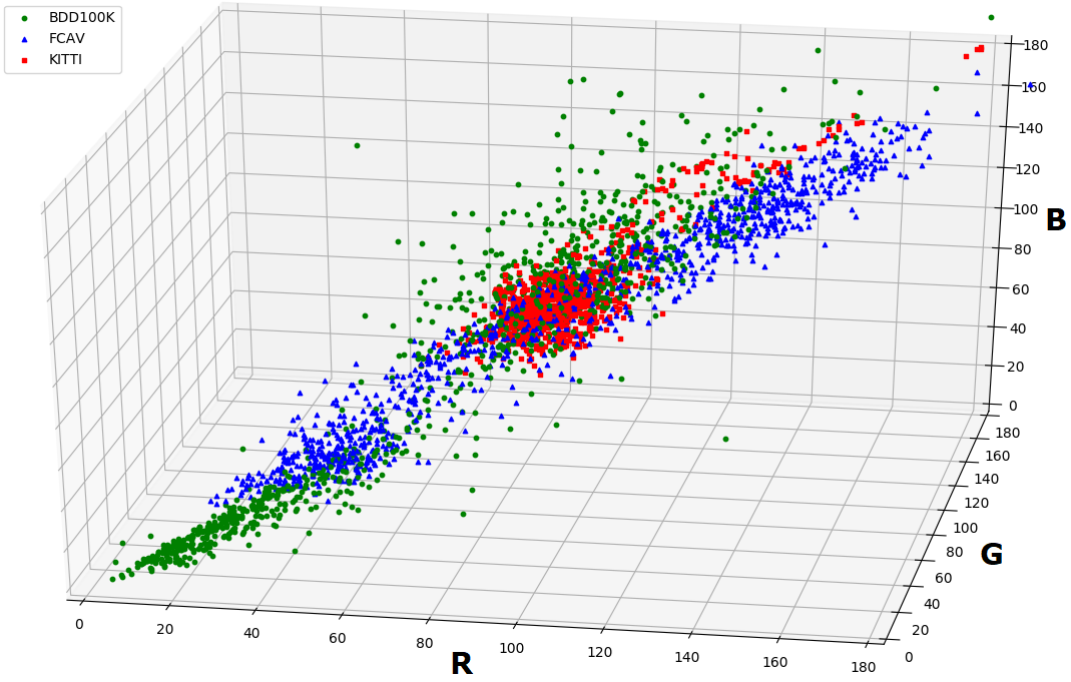

FIGURE 4.12: 3D scatter plot of mean RGB values for 1,000 random images from BDD100K, FCAV, and KITTI.

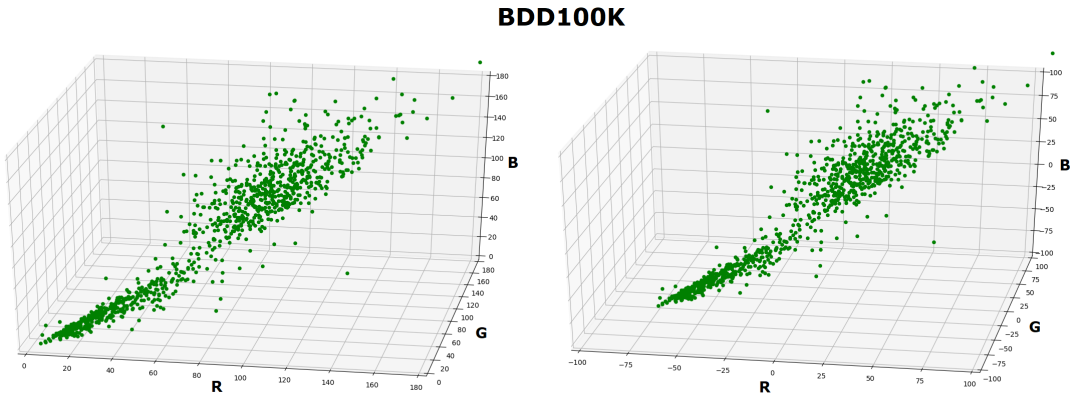

FIGURE 4.13: Mean RGB values for 1,000 random images from BDD100K before and after mean subtraction.

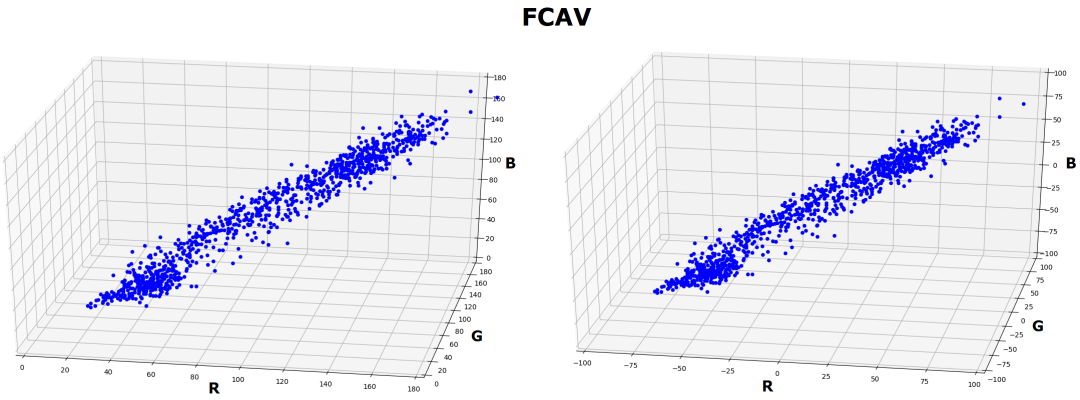

FIGURE 4.14: Mean RGB values for 1,000 random images from FCAV before and after mean subtraction.

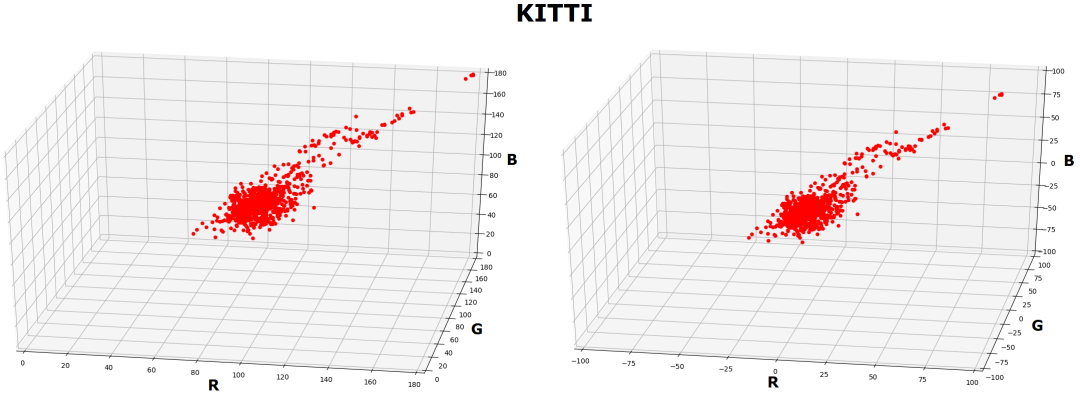

FIGURE 4.15: Mean RGB values for 1,000 random images from KITTI before and after mean subtraction.

4.5 Implementation Example of Building a CNN in Keras

Implementing a CNN in Keras is done by stacking up different CNN layers, together with specifying activation functions, padding for pooling, dropout rates, etc., as shown in this simple example:

```
'''
Builds and returns a simple CNN with 8 layers (6 conv, 2 fully connected).
Images are resized to INPUT_SIZExINPUT_SIZE
'''
def createModel():

    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
                     input_shape=(INPUT_SIZE,INPUT_SIZE,3)))
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
17      model.add(Dropout(0.25))
18
19      model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
20      model.add(Conv2D(64, (3, 3), activation='relu'))
21      model.add(MaxPooling2D(pool_size=(2, 2)))
22      model.add(Dropout(0.25))
23
24      model.add(Flatten())
25      model.add(Dense(512, activation='relu'))
26      model.add(Dropout(0.5))
27      model.add(Dense(10, activation='softmax'))
28
29      return model
```

This example code shows a python function called *createModel* which returns a simple CNN consisting of 8 layers (6 convolutional and 2 fully connected). In line 7 the model is declared as a sequential model which allows one to easily stack sequential layers to the network.

The first layer added in line 8 is a 2D convolutional layer which will be the first layer to process input images. The first argument of the *Conv2D* function is the number of output channels, followed by a tuple declaring the kernel size. Using 'same' for padding will make Keras to try to pad with zeros evenly where the kernel filter reaches outside image boundaries. Next, the activation function is set as a rectified linear unit (ReLU), and finally the model has to be initiated with the size of the input to the layer (image resolution and color channels). The input shape only has to be declared in the first layer, in the following parts of the network Keras will work out the size of the tensors automatically. Keras will also take care of the declarations of all weights and bias variables throughout the network.

In the following convolutional layer, added in line 10, only two arguments are declared; the number of output channels, and the activation function. The rest of the arguments are set to default values specified in the Keras documentation [55]. In line 11 a max pooling layer is added with a 2×2 filter, and in line 12 dropout is applied to the preceding layer with a drop out rate of 0.25. This procedure is performed three times (line 8-22) resulting in six convolutional layers with max pooling and dropout.

Finally, the soft-max classification, or output layer, is added in line 27 including a specification on how many output nodes the network will have (10 in this example). To connect the last convolutional layer to the soft-max layer the output of the last convolutional layer is flattened in line 24. This flattened layer is followed by declaring a fully connected layer in line 26 using the *Dense* layer in Keras. On the last line the built model is returned to the caller of the function.

The next step in the process is to compile the model and start training on a set of training images. In Keras, the compiling of the model is done with one command:

```
1  model.compile(loss=keras.losses.categorical_crossentropy,
2                optimizer=keras.optimizers.SGD(lr=0.01),
3                metrics=['accuracy'])
```

In this example the loss function is set to Keras standard cross entropy for categorical classification, but it can also be set to any custom self defined loss function such as, for example, the SSD loss function described in equation 2.16. When compiling the model the optimizer also has to be defined. In this example SGD is used with a

learning rate of 0.01. Finally, a metric that will be computed when evaluating the model is specified which also can be custom designed when needed.

Training a model in Keras can once again be executed by running a single command:

```
1 model.fit(training_data,
2           target_data,
3           batch_size=32,
4           epochs=100,
5           verbose=1,
6           validation_data=(x_val, y_val),
7           callbacks=[history])
```

Here, the first arguments are numpy arrays with training data and ground truth target data. The next argument is the batch size witch is set to 32 for this example and on line 4 the number of epochs to train is set to 100. The verbose flag, set to 1 on line 5, specifies if detailed information should be printed out in the console describing the progress of the training. On line 6 the validation data is passed to the fit function for Keras to test the metric against when evaluating. The last argument sets the Keras callbacks which are used to track different variables during training and create checkpoints to save the model at multiple stages in the training process.

When working with big image-based datasets, like the ones in this thesis project, the complete dataset will most likely not fit into memory using simple numpy arrays. Instead, a common technique is to use data-generators to creates data in batches of input and corresponding output data on the fly during the training process like in this example:

```
1 train_gen = ImageDataGenerator().
2             flow_from_directory(TRAINING_DATA_PATH,
3                                 target_size=(INPUT_SIZE,INPUT_SIZE),
4                                 batch_size=32)
5 valid_gen = ImageDataGenerator().
6             flow_from_directory(VALIDATION_DATA_PATH,
7                                 target_size=(INPUT_SIZE,INPUT_SIZE),
8                                 classes=CLASSES,
9                                 batch_size=32)
```

Here the Keras data-generator *ImageDataGenerator* is used together with one of its built in functions for reading images directly from a given directory. When using data-generators to load datasets, the *fit* function used earlier for training the model has to be changed to the corresponding *fit_generator* function:

```
1 model.fit_generator(train_gen,
2                     batch_size=32,
3                     epochs=100,
4                     verbose=1,
5                     validation_data=valid_gen,
6                     callbacks=[history])
```

All this builds, compiles and trains simple CNN capable of classifying images of size INPUT_SIZE$\times$INPUT_SIZE with three color channels into ten classes.

Chapter 5

Implementation of the Complete SSD Network in Keras

By using the architectural design described in the SSD paper and shown in figure 5.1, together with the Caffe implementation released by the authors, the SSD network is stacked up, compiled, and trained in Keras the same way as shown in the simple example in section 4.5. As explained in section 2.11, any high-quality image classifier network can be used as the pretrained base classification network used as the first section of SSD. However, for this thesis project, the VGG-16 [33] network, which is pretrained on the ILSVRC CLS-LOC dataset [13], is used just as in the original SSD implementation. Parameters for all convolutional layers not part of the pretrained VGG-16 network are initialized with the Xavier method [37] explained in section 2.3.8. Even if the base network is pretrained, its parameters are updated when training the SSD network, making the numbers of trainable parameters in the final Keras implementation of SSD, to sum up to 24,948,866. The complete Keras architecture of the SSD network implemented for this project can be found in Appendix A.

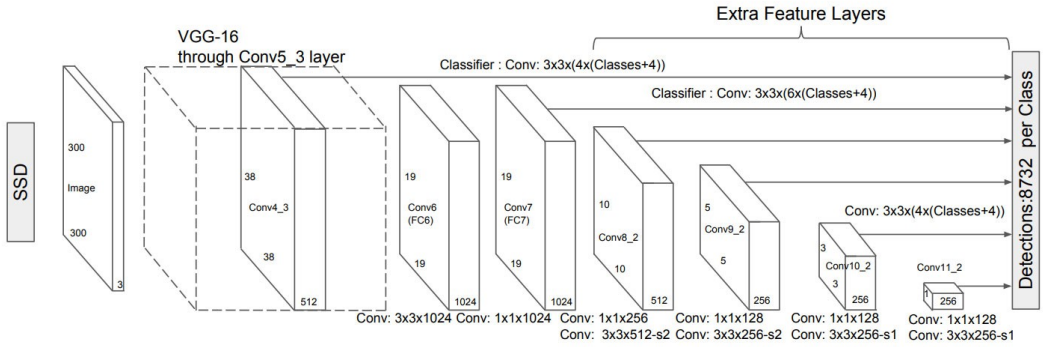

FIGURE 5.1: Architectural design of SSD showing all prediction layers. *Figure from:* [6]

As explained in section 2.11, and shown in figure 5.1, the final SSD network produces more than 8,000 bounding box predictions per class, where a significant fraction of those boxes refer to the same objects. Although this is fine during training, at test time these duplicate predictions pointing to the same objects needs to be pruned using NMS. Another detail that has to be addressed during test time is that the network will output bounding box coordinates relative to the anchor boxes defined when building the network. Thus, the resulting coordinates for bounding box predictions have to be decoded into absolute coordinates to match input images. Additionally, images have to be resized to fit the input layer of the network, which in this case is a tensor consisting of three pixel-value matrices (one for each color) with size 300×300.

5.1 Train-, Validation-, and Test-data

The ultimate goal when training SSD is to create a network able to generalize well to input samples that it has never seen before. As mentioned earlier, SSD consist of 24,948,666 trainable parameters and may, therefore, be able to memorize a complete training dataset if that dataset consists of too few data-samples. This behavior of memorization is generally known as overfitting (see section 2.3.4). To be able to track the performance during training and to detect if SSD is overfitting, a small subset of the training data is used as a validation set. This new small subset is used to provide an unbiased evaluation of the networks fit on the training dataset periodically during training (see figures 5.2-5.5). To be able to reliably evaluate the final trained SSD network, the training data is split into a third distinct test dataset. This test data is not exposed to the network until the training phase is over. Train, validation, and test split ratios for each dataset are presented in section 4.3.

5.2 Training

Many hyperparameter settings used when training the SSD implementation in this thesis project will be identical to the hyperparameters used in the SSD paper. Some examples of these parameters are: spacing between anchor box center points for each prediction layer, batch size, kernel (convolutional filter) sizes and dimensions, pooling filter sizes, and all loss function parameters. Thresholds values for confident scores and IoU overlaps used for NMS, together with IoU thresholds for positive matches will also be implemented identically as in the original SSD implementation. If any hyperparameters are changed compared to the original SSD implementation, these changes will be presented in the report. For all other hyperparameter values, the reader is referred to the SSD paper [6].

Data augmentation techniques used when training will be the same as the ones implemented in the SSD paper. These techniques are random image expansion, random cropping, random mirroring and photometric distortions (changing the brightness, contrast, hue, and saturation) as described in [59].

The original SSD network was trained on the PASCAL VOC2007 [14] and VOC2012 [52] datasets using mini-batch gradient descent (see section 2.3.3) with a batch size of 32 and a learning rate of 10^{-3} for the 120,000 first iterations. The initial learning rate was then lowered to 10^{-4} for the last 40,000 iterations. The strategy used in this thesis project will be to use an initial learning rate of 10^{-3}, which will be lowered when the network seems to begin to converge.

While training, a Keras callback for checkpoints will be used for saving all weights and bias values periodically to disk. These checkpoints are used to prevent losing previous training progress during training sessions which will take days to bring to completion, despite being conducted on a GPU (NVidia Tesla K80).

5.2.1 PASCAL

To ensure that the implementation of SSD in Keras in this thesis project is an accurate representation of the original network, the Keras implementation was trained and evaluated on the PASCAL datasets presented in section 4.3.4, just as the original implementation of SSD. The complete architecture and all settings were implemented as similar as possible to the original SSD implementation when building and training the Keras version of the network. To replicate the training as accurate as possible the network was trained, just as in the SSD paper, using mini-batch gradient descent

with a batch size of 32, and an initial learning rate of 10^{-3}, which was then lowered to 10^{-4} when the network began to converge. Some problems did arise with loss values going to infinity in the early training iterations were exponential growth for the loss led to floating point overflows (giving Python NaN values), which was most likely due to *exploding gradients*[1]. This problem with growing loss values could have been resolved by using another optimizer than SGD [60], but seeing that the purpose of this experiment is to replicate the results from the SSD paper as accurately as possible, the aspiration was to use the same optimizer as was used in the original SSD implementation. Therefore, instead of changing optimizer, the training was restarted from an untrained state if the loss value went to infinity. On the third attempt the problem did not appear, and the loss value continued to decrease (on average) throughout the complete training session as expected. At 58,000 iterations the network seemed to start to converge (see figure 5.2), and the learning rate was therefore lowered to 10^{-4}, followed by 22,000 additional iterations. This training session lasted almost 80 hours, or slightly over three days.

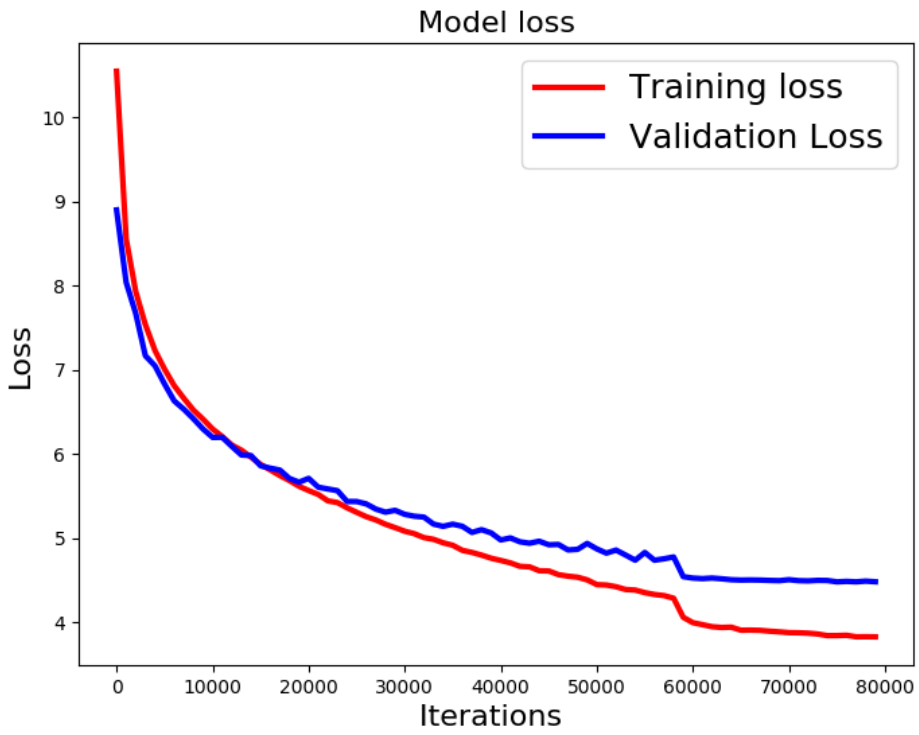

FIGURE 5.2: Training and validation losses for 80,000 training iterations on PASCAL.

5.2.2 BDD100K

In this experiment, the SSD network was trained on the full BDD100K dataset consisting of ten classes (bus, traffic light, traffic sign, person, bike, truck, motorcycle, car, train, rider). Before images were fed to the network they were resized to 300×300 pixels, and the per channel color mean values were subtracted (see figure

[1]Exploding gradients is a phenomenon that causes gradients that are backpropagated through the network to grow exponentially from layer to layer. This exponential growth is caused by the explosion of the long term components, which can grow exponentially more than short term ones [60].

4.13). The per channel mean values for BDD100K ($R = 70$, $G = 74$, $B = 70$) were found by computing the mean RGB value from 10,000 random images in the dataset.

Looking at figures 4.8 and 4.10 one can see that the dataset has a big fraction of small bounding boxes, where the majority of objects have bounding box widths and heights smaller than 50 pixels. Therefore the scaling factors s_{min} and s_{max} explained in section 2.11 were set to $s_{min} = 0.1$ and $s_{max} = 0.86$ (the original SSD network trained on PASCAL used $s_{min} = 0.2$ and $s_{max} = 0.9$). Seeing that there are six prediction layers (five, excluding the Conv4_3 layer inside the base network), equation 2.21 gives the hyperparameters for each prediction layer: $s_1 = 0.1$, $s_2 = 0.29$, $s_3 = 0.48$, $s_4 = 0.67$, $s_5 = 0.86$. An extra scaling factor, s_0, for the Conv4_3 layer inside the base network (see figure 5.1) used for detecting extra small objects was set to $s_0 = 0.05$.

When SSD was trained on PASCAL in the original implementation, five different aspect ratios where used to define the shape of all anchor boxes $a_r^{pascal} \in \{0.5, \frac{1}{3}, 1, 2, 3\}$. Using figure 4.9, new aspect ratios values were chosen and set to $a_r^{BDD100K} \in \{0.5, 1, 1.5, 2, 2.5\}$, based on the scales of objects in BDD100K.

Just as when training on PASCAL, mini-batch gradient descent was chosen as the optimizer, with a batch size of 32 (larger batch sizes were tested, but this resulted in issues with GPU memory). As expected, the same problem with exploding gradients in early training iterations as in the PASCAL experiment appeared when training on BDD100K. This time, the experiment had to be restarted six times until the loss value ceases to snowball during early training iterations.

For this training session, an initial learning rate of 10^{-3} was used, which was lowered to 10^{-4} when the network seemed to begin to converge at around 64,000 iterations, as can be seen in figure 5.3. Training the SSD network on the complete BDD100K dataset took almost 150 hours, or slightly over six days.

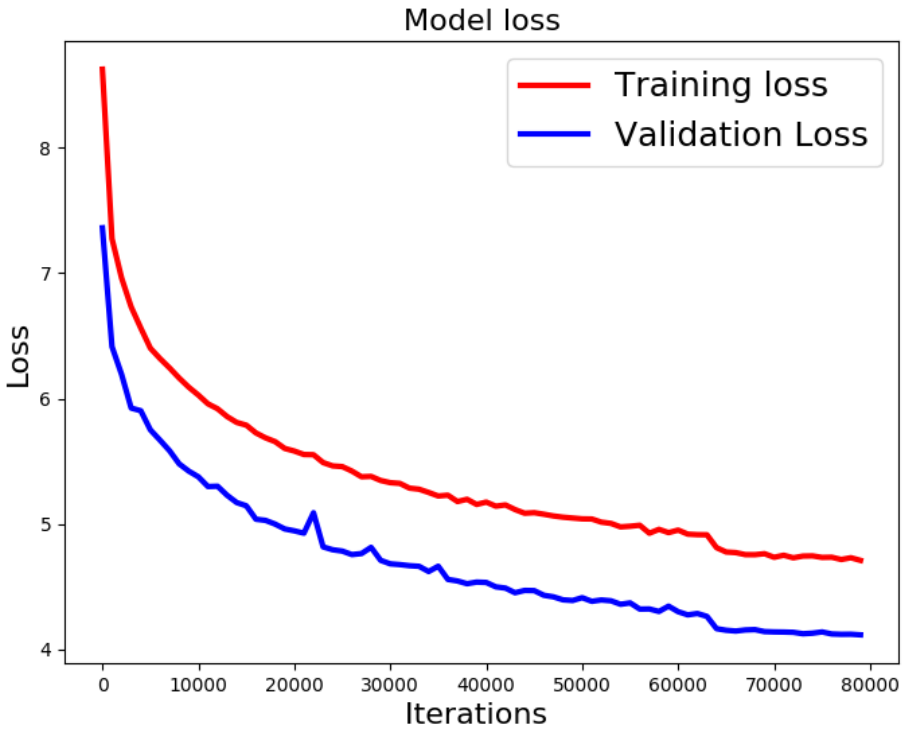

FIGURE 5.3: Training and validation losses for BDD100K.

5.2.3 Virtual and Real Data

In this experiment, two versions of the network were trained, one version using the virtual dataset FCAV, and one version using a subset of BDD100K (containing only images and annotations for cars). Due to continuous problems with growing loss values during early training iterations, other optimizers were considered for these training sessions. The choice finally fell on the Adam optimizer[2]. Adam is currently generally recommended as the default optimizer to use when training deep neural networks on the grounds that using Adam often helps speeding up the training time compared to SGD [62]. Another, most relevant advantage of using Adam, is that it reduces the risk for vanishing and exploding gradients [61]. This advantage was validated in both these training sessions since no problems with exploding gradients where encountered when using the Adam optimizer.

By studying figures 4.9 and 4.10 one can see that both FCAV and the subset of BDD100K have similar distributions in bounding box sizes and aspect ratios. Therefore, to further aid a fair comparison, both networks were trained using the same settings for bounding box scales and aspect ratios as was used in the SSD paper when training on the COCO[3] dataset. The COCO scaling factors, $s_{min} = 0.15$, $s_{max} = 0.87$, and $s_0 = 0.07$, were used since COCO contain objects with smaller sizes than PASCAL. Aspect ratios for COCO used in the paper were identical to the aspect ratios used when training on PASCAL (see section 5.2.1).

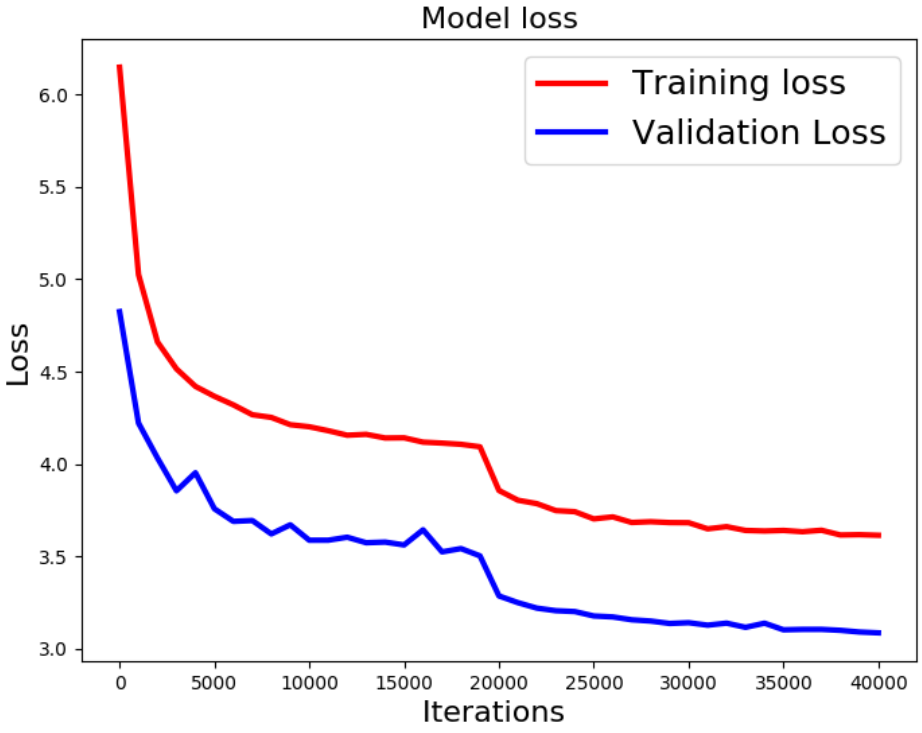

FIGURE 5.4: Training and validation losses for subset of BDD100K including only cars.

[2]The Adam optimizer is an extension to SGD which uses an adaptive learning rate found by running an average of the first and second order moments of the gradients [61].

[3]Microsoft Common Objects in Context (COCO) [15] is a dataset for bounding box object detection, object segmentation and scene understanding with 2.5 million labeled instances in 328,000 images.

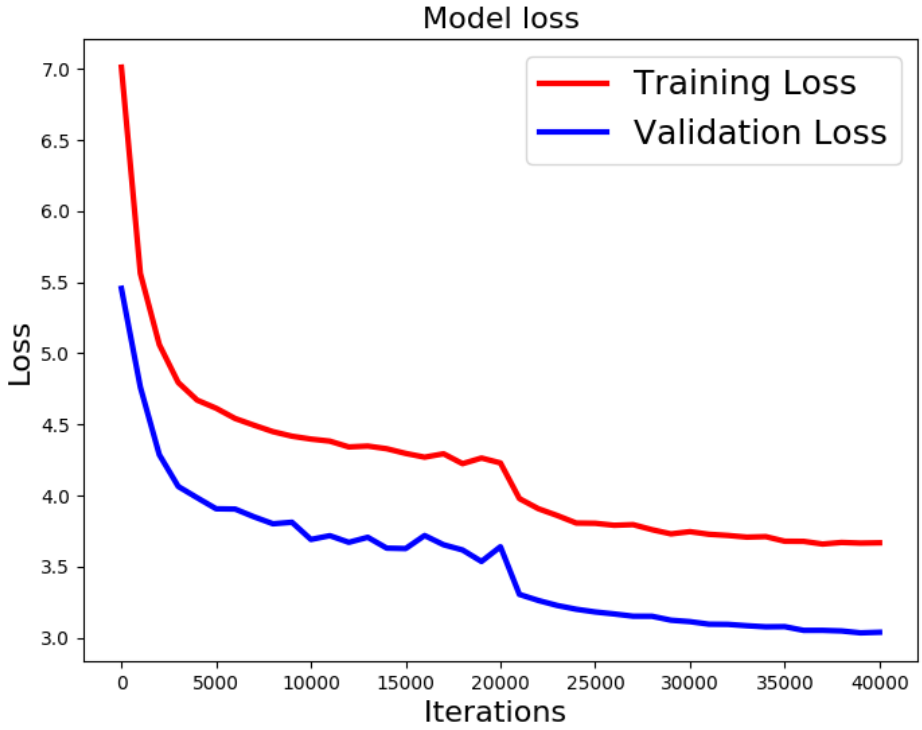

FIGURE 5.5: Training and validation losses for FCAV.

Using the Adam optimizer eliminated all problems with exploding gradients in both training sessions. It also seemed to speed up the training time since both training sessions converged at around 20,000 iterations where the learning rate was lowered and then followed by 20,000 additional training iterations for both networks (changing optimizer did not result in any noticeable change in training time per iteration). Training the SSD network on the subset of BDD100K took little over 70 hours, or around three days, and the FCAV training session lasted more than twice as long with a duration of almost 150 hours, or slightly over six days.

5.3 Testing

All testing was conducted on distinct and isolated test sets of the different datasets which were split as described in section 4.3 (except when comparing networks trained on synthetic and real data where an additional dataset named KITTI was used to support an unbiased evaluation further).

To compute mAP values for a trained network on a certain dataset, the predictions on the test set of that dataset was evaluated with the official Pascal VOCdevkit 2007 development kit [63] evaluation code. This evaluation code performs the following steps: 1) Runs predictions over the entire given test set. 2) Matches the predictions to annotated ground truth boxes. 3) Computes the precision-recall curves by varying the confidence score threshold that determines what is counted as a predicted positive detection of the class. 4) Samples eleven equally spaced recall levels from these precision-recall curves to compute APs for each class (which essentially is the area under the precision-recall curve). 5) Computes the mAPs over all classes.

Additionally, precision-recall curves (see section 3.1) are also presented for all networks to give a more refined picture of their performances.

5.4 Results, Analysis and Evaluation

In this section, all results are presented, analyzed, and evaluated for each individually trained SSD network. As described in section 3.1, if not otherwise stated, all presented AP and mAP values are evaluated using an IoU threshold of 0.5.

5.4.1 PASCAL

Results for the Keras implementation of SSD trained on the PASCAL dataset will be compared with results for two versions of the original SSD network, one version with the 'zoom-out' operation explained in section 2.11, and one version without. The reason for this is that the authors of the paper did not present per class scores for the version trained with the 'zoom-out' operation.

TABLE 5.1: Average precisions for all predicted classes on the test data in PASCAL 2007 as they were reported in the SSD paper (without the 'zoom-out' operation explained in section 2.11), and the implemented Keras version of SSD in this thesis evaluated using the official Pascal VOCdevkit 2007 evaluation code.

Class	**Original SSD [AP]**	**Keras SSD [AP]**
Airplane	0.856	0.784
Bicycle	0.801	0.829
Bird	0.705	0.749
Boat	0.576	0.668
Bottle	0.462	0.474
Bus	0.794	0.864
Car	0.761	0.852
Cat	0.892	0.875
Chair	0.530	0.582
Cow	0.770	0.793
Dining table	0.608	0.762
Dog	0.870	0.844
Horse	0.831	0.869
Motor Bike	0.823	0.844
Person	0.794	0.765
Potted Plant	0.459	0.505
Sheep	0.759	0.774
Sofa	0.695	0.790
Train	0.819	0.864
TV monitor	0.675	0.764
mAP	**0.724**	**0.763**

TABLE 5.2: Mean average precisions on the test data in PASCAL 2007 as they were reported in the SSD paper (with and without the 'zoom-out' operation explained in section 2.11), and the implemented Keras version of SSD in this thesis evaluated using the official Pascal VOCdevkit 2007 evaluation code.

	Original SSD without 'zoom-out'	**Original SSD with 'zoom-out'**	**Keras SSD**
mAP	**0.724**	**0.772**	**0.763**

In table 5.1 all per class scores are presented for both the Keras implementation and the original SSD network trained without the 'zoom-out' operation, and table 5.2 shows the mAP scores for the Keras implementation compared to the two different versions of the original SSD network. Although the Keras implemented SSD network in this thesis project uses the 'zoom-out' operation and therefore should decisively be evaluated against the original SSD version also trained using this operation, the comparison in table 5.1 gives a more detailed picture of the performances than merely comparing final mAP scores.

The PASCAL precision-recall curves for the SSD implementation shown in figure 5.6 do not give any additional relevant information about the correctness of the SSD implementation. Instead, this figure is presented as a comparison for the precision-recall curves for the versions of the SSD network trained on other datasets discussed in the following sections.

The drop that can be seen just before 60,000 iterations in figure 5.2, showing the training loss curves for the Keras implementation trained on PASCAL, is due to the reducing of the learning rate from 10^{-3} to 10^{-4}. From looking at the training and validation loss curves, one can conclude that waiting for a few thousand iterations more before lowering the learning rate could have improved the final performance to some degree. If that is the case, it could also explain why the Keras implementation performed slightly worse than the original SSD version (see table 5.2). Another explanation for this small performance difference could be that the networks were trained with a different number of training iterations. Using these insights, the conclusion is that the implementation of SSD in Keras in this thesis project is an accurate representation of the original network.

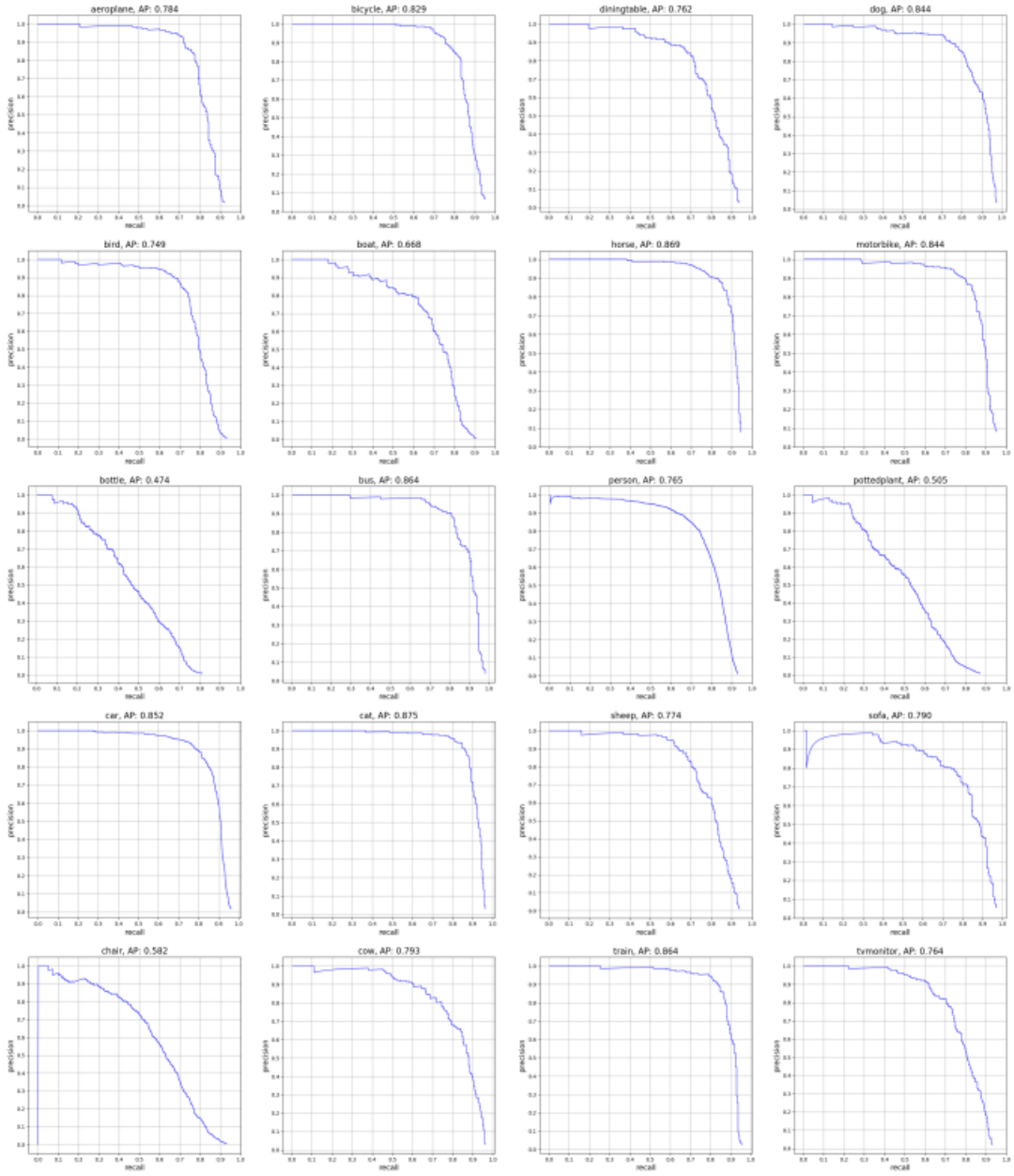

FIGURE 5.6: Precision-recall curves for all 20 classes in PASCAL for the SSD network trained on PASCAL

5.4.2 BDD100K

As a benchmark for evaluating the performance of SSD trained on BDD100K, the network will be compared against the results of the runner up in the road object detection competition presented during the 2018 Workshop of Autonomous Driving (WAD), which was based on the BDD100K dataset. The reason the second place taker in the competition is used is that the winners did not publish a paper presenting their results. For this competition, an IoU threshold of 0.7 was used with the winner and runner up scoring mean average precisions of 0.331 and 0.297 on the BDD100K dataset, respectively. The second place taker in the competition presented an object detection network called CFENet in their paper named *CFENet: An Accurate and Efficient Single-Shot Object Detector for Autonomous Driving* [56]. The network is implemented as a single-shot (one-stage) network. However, by using a input size of 800×800 pixels, the fastest version (CFENet800) process images at 21 FPS [56], which is less than half the speed of SSD and not fast enough to be considered a real-time system in the context of this thesis report (as discussed in section 2.9).

In table 5.3 the average precisions for all predicted classes on the test data in BDD100K are presented as they were reported in the CFENet paper, compared with the results of the implemented Keras version of SSD in this thesis. It should be noted that CFENet was trained using all 80,000 training images in BDD100K, and evaluated by submitting predictions for the 20,000 test images (which does not have publicly released annotations) to BAIR for evaluation. In contrast, the SSD implementation in this thesis report used the 80,000 training images for training, validation, and testing, as described in sections 4.3.1 and 5.1.

The first thing to notice when looking at table 5.3 is that all three networks have somewhat poor results with low mAP scores, at least in the context of being used in self-driving systems. Even the best performing network of the three, CFENet800-MS (without reported FSP in the paper, but presumably lower than the reported 21 FPS of CFENet800), had a relatively low mAP score of 0.297, which loosely can be interpreted as correctly detecting[4] around thirty percent of all objects in the test data. The first and probably most significant reason behind these low AP scores is the BDD100K dataset itself, which was used for training all three networks. BDD100K is a challenging dataset, created to test the limit of object detectors, by reflecting a variety of driving scene conditions that a self-driving car could encounter today. The effect on the performance of SSD by using BDD100K for training is discussed further in section 7.1. One also has to take into consideration that the evaluation was done using a IoU threshold of 0.7, which is relatively strict and therefore naturally decreases the resulting AP scores. This effect can be seen when the SSD network is evaluated using a more widely used IoU of 0.5, which results in higher AP scores for all classes.

[4] A correct detection for CFENet800-MS would require both a correct classification and a predicted bounding box with an IoU higher than 0.7.

TABLE 5.3: Average precisions for all predicted classes on the test data in BDD100K as they were reported in the CFENet paper (with an IoU threshold set at 0.7), and the implemented Keras version of SSD in this thesis evaluated using the official Pascal VOCdevkit 2007 evaluation code (showing results with IoU thresholds set at both 0.7 and 0.5).

Class	**CFENet800 [AP]**	**CFENet800-MS [AP]**	**SSD (IoU=0.7) [AP]**	**SSD (IoU=0.5) [AP]**
Bike	0.146	0.205	0.057	0.087
Bus	0.396	0.504	0.111	0.150
Car	0.452	0.513	0.273	0.473
Motorcycle	0.115	0.167	0.091	0.091
Person	0.176	0.291	0.015	0.138
Rider	0.127	0.239	0	0.041
Traffic light	0.087	0.153	0.020	0.176
Traffic sign	0.283	0.375	0.111	0.278
Train	0	0	0	0
Truck	0.453	0.522	0.116	0.234
mAP	**0.223**	**0.297**	**0.079**	**0.167**

The second thing to notice is that SSD has significantly worse performance than the two CFENet networks, with lower AP scores for all classes. This difference is most likely caused by the repression of both trainable parameters and input size to make SSD operate at real-time frame rates. Moreover, most one-stage detectors struggle with detecting smaller objects, and as discussed earlier, BDD100K consist of a large number of small sized objects (see figure 4.4).

Vehicles like cars, busses and trucks are likely easier to detect due to a combination of enough training samples and these objects being larger on average compared to the other objects in the dataset. Another indication of the importance of having many training samples is that cars, with more than half of all objects in BDD100K (1,021,857 instances), gets the highest AP scores for all networks shown in table 5.3. In contrast, there are only 179 trains in the dataset, which results in zero detections for all tested networks.

In figure 5.7 the precision-recall curves for all ten classes in BDD100K are presented for the SSD network evaluated with an IoU threshold of 0.5. Here, the two curves belonging to the classes with most training samples in the training data (traffic signs and cars), are the only ones resembling the curves from the PASCAL evaluation (see figure 5.6). The rest of the classes show low precision scores even when confidence thresholds for correct detections are high, at recall values close to zero. Or in other words, the network still makes a fair amount of mistakes, even if only detections that the network is sure of (high confidence scores) are counted as positive predictions (see section 3.1).

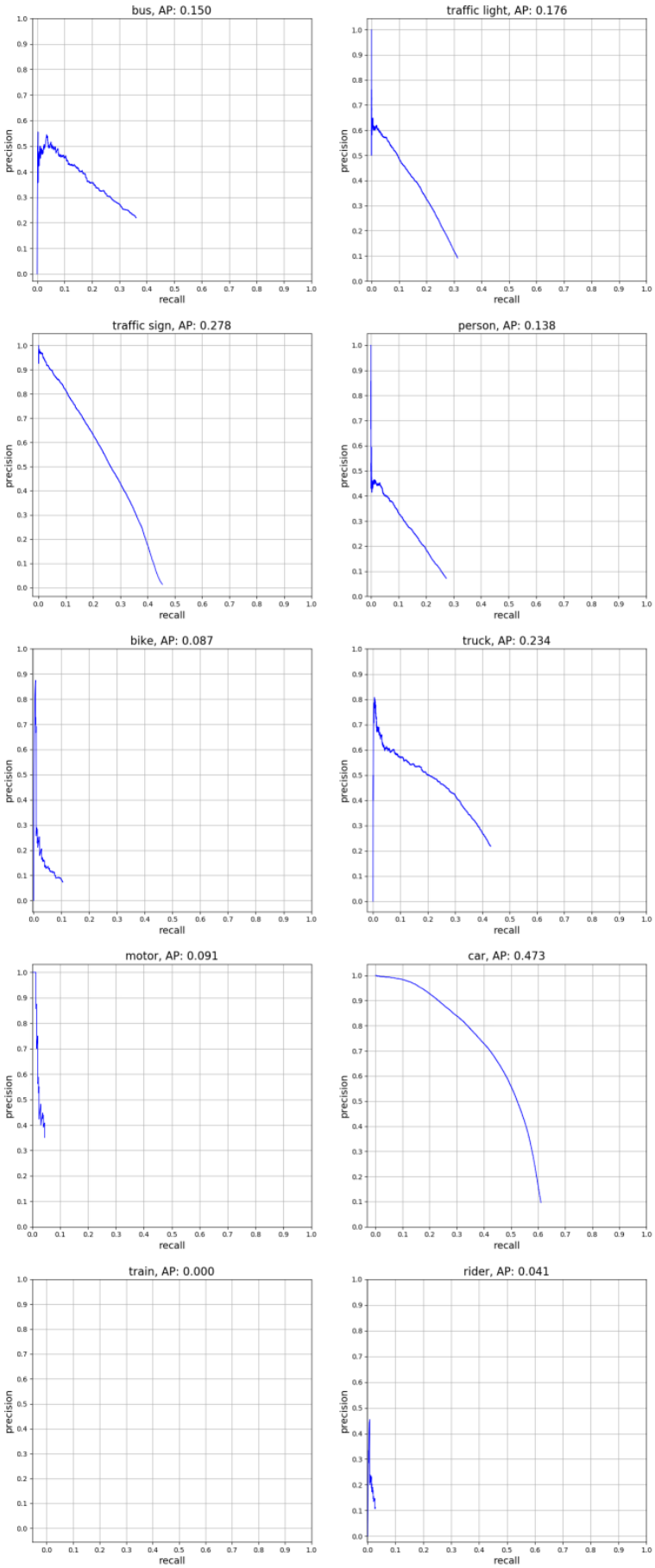

FIGURE 5.7: Precision-recall curves for the SSD network trained on BDD100K (evaluated with an IoU threshold of 0.5).

Figure 5.8 shows examples of detections (and missed detections) in eight sample images from the 10,000 test images in BDD100K made by the SSD network trained on the complete BDD100K dataset. In these examples, one can observe that SSD seems to struggle with smaller objects such as traffic signs, distant cars, and persons. Further qualitative results, in the form of video visualization of detections made by SSD outside the office of Cybercom in Kista, Stockholm, can be found online[5].

FIGURE 5.8: Eight images from the test set of BDD100K [12], showing ground truth bounding boxes (in red) and detections made by the trained SSD network.

[5]Video visualization of real-life detections made by SSD can be found at `https://youtu.be/oDp9ozP_q5E`.

5.4.3 Virtual and Real Data

Here, a comparison between two versions of SSD is conducted. The first version is trained on the virtual dataset FCAV, and the other version is trained on a subset of BDD100K (containing only images and annotations for cars). To be able to quantitatively analyze the performance and make a fair comparison between the two networks, all 7,481 training images from the KITTI dataset were used for evaluation in addition to evaluating the networks on the test sets of both FCAV and the subset of BDD100K. These cross-evaluations are presented in tables 5.4 and 5.5, with corresponding precision-recall curves in figures 5.9 and 5.10.

TABLE 5.4: Average precisions for SSD, trained on the subset of BDD100K (containing only cars), evaluated on the complete KITTI dataset and the test sets of both FCAV and the subset of BDD100K.

BDD100K evaluated on:	**BDD100K**	**KITTI**	**FCAV**
AP	**0.461**	**0.585**	**0.348**

TABLE 5.5: Average precisions for SSD, trained on FCAV, evaluated on the complete KITTI dataset and the test sets of both FCAV and the subset of BDD100K (containing only cars).

FCAV evaluated on:	**FCAV**	**KITTI**	**BDD100K**
AP	**0.507**	**0.319**	**0.283**

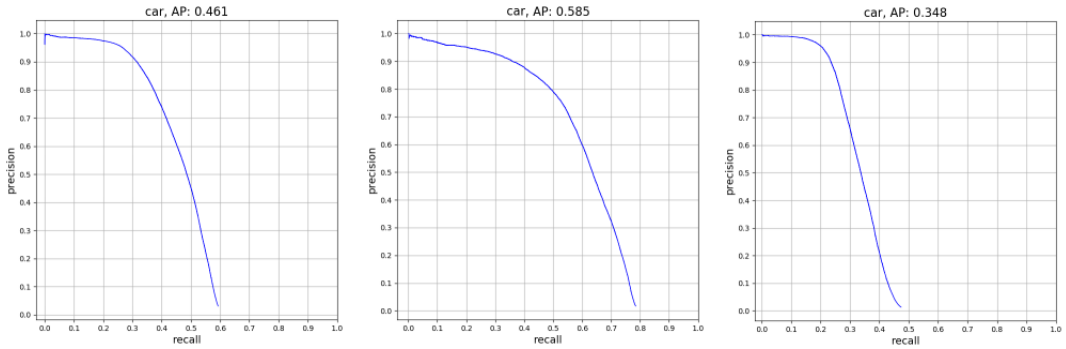

FIGURE 5.9: Precision-recall curves for the SSD network trained on the subset of BDD100K (containing only cars). The three curves shows evaluations on *left:* BDD100K, *middle:* KITTI, and *right:* FCAV.

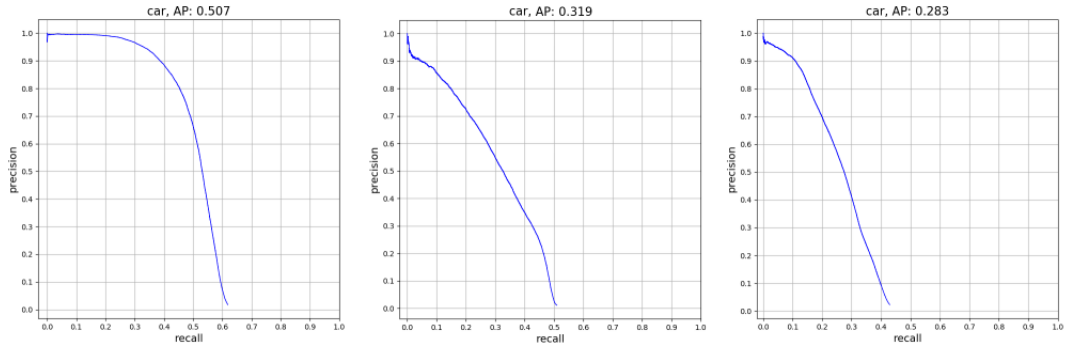

FIGURE 5.10: Precision-recall curves for the SSD network trained on FCAV. The three curves shows evaluations on *left:* FCAV, *middle:* KITTI, and *right:* BDD100K.

Looking at the AP scores for the two networks when evaluated on KITTI, it is clear that the BDD100K trained SSD network outperforms the version trained on FCAV. However, with an AP score of 0.319, this experiment shows that it is possible to train an object detecting deep neural network to recognize real-life objects using only synthetic computer generated images.

By comparing AP scores for when the networks were evaluated on their own test sets, one can see that the virtual dataset FCAV almost is as challenging as BDD100K scoring an AP of 0.507, compared to an AP of 0.461 for BDD100K. This is somewhat expected since images in FCAV were simulated during similar conditions as the real conditions during the collection of BDD100K.

Seeing that lighting, color, and texture variation in the real world is greater than that of the synthetic images, one possibility is that it would require many more simulated images to achieve similar performance as with real-world images.

Chapter 6

Related Work

There exist two established types of DNNs capable of conducting object detection in digital images. The first one uses a more traditional method which utilizes two networks working together. The first network is used to find regions in the image with potential objects, which is followed by a second classifying network applied on each of the proposals. The second type treats the problem as a regression problem using sampling over different locations, scales and aspect ratios and is all done adopting a unified network to achieve the final result. The first more traditional approach is generally more accurate but slower and is often referred to as a two-stage method, while the second approach typically results in a faster but less precise model and is commonly called a one-stage method [2].

6.1 One-stage networks

DNN models in the forefront of object detection using one-stage methods mainly includes different versions of YOLO [5] and SSD [6]. These regression-based networks all have similar architectural types and comparable performances. The original Single Shot Multibox Detector (SSD), was implemented, tested, and thoroughly reviewed in this thesis report. Therefore, the only other one-stage network discussed in this section will be the original YOLO architecture.

6.1.1 YOLO

You only look once (YOLO) is a real-time object detection system which applies a single neural network to the full input image, hence the name *You only look once*. YOLO divides the input image into an $S \times S$ sized grid where each cell of the grid is responsible for predicting B number of bounding boxes with associated confidence scores, and C conditional class probabilities representing the probability for each class given a certain predicted object. The confidence score of a bounding box is computed by multiplying the class probability by the corresponding IoU between a predicted box and a ground truth box (see figure 6.1).

The final layer of the network outputs an $S \times S \times (C + B \times 5)$ tensor representing the predictions for each of the cells in the grid. The five in the equation corresponds to the five predictions: x-coordinate for bounding box center, y-coordinate for bounding box center, bounding box width, bounding box height, and prediction confidence score. When YOLO was trained on the PASCAL dataset the authors of the paper used $S = 7$, $B = 2$ and $C = 20$ (PASCAL has 20 classes), resulting in a final prediction tensor of size $7 \times 7 \times 30$.

Just as SSD, YOLO predicts a big number of bounding boxes, where a significant fraction of those refer to the same objects. Although this is fine during training, at test time these duplicate predictions pointing to the same objects need to be pruned

to one per predicted object. This pruning is done using Non-Maximum Suppression (NMS) (see section 2.11), which removes boxes with low class probabilities and merges highly-overlapping bounding boxes of same objects into single boxes as can be seen in figure 6.1.

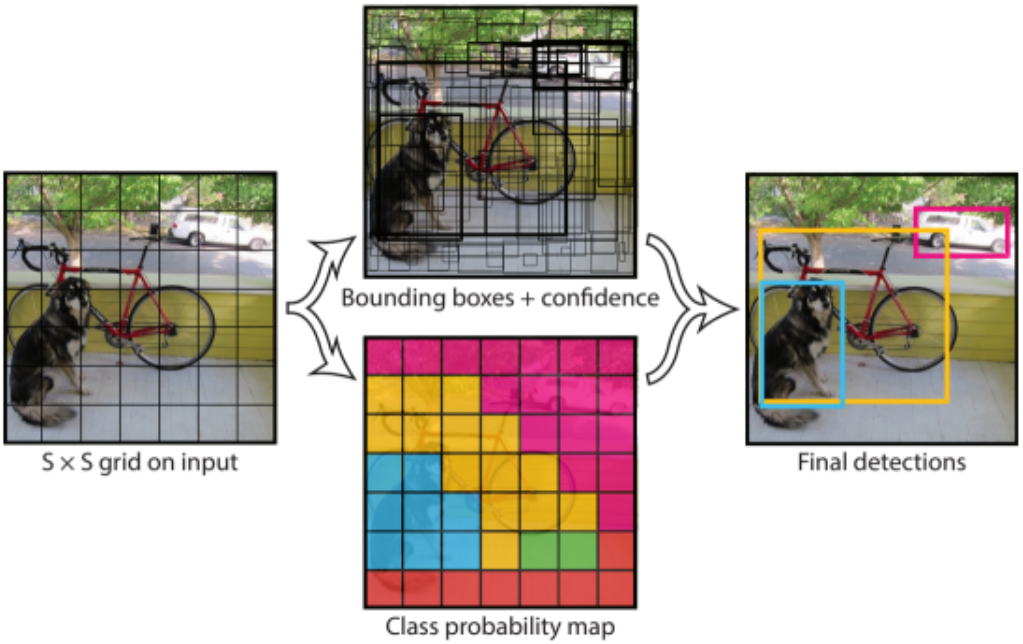

FIGURE 6.1: The YOLO detection model divides the input image into an $S \times S$ grid, and for each grid cell it predicts B bounding boxes, confidence for those boxes, and C class probabilities. *Figure from:* [64]

6.2 Two-stage networks

In two-stage networks, the first step consists of category-independent region proposals, followed by CNN feature extraction from these regions. In the second step category-specific classifiers are used to determine the category labels of the proposals. Most two-stage networks produce thousands of region proposals at test time, which comes with a high computational cost. The two main two-stage based networks for object detection using bounding boxes, which have influenced the development of sereral similar networks, are Faster R-CNN [35] and R-FCN [46].

6.2.1 Faster R-CNN

The Faster R-CNN network is a developed version of the earlier networks Fast R-CNN [34] and R-CNN [64], with both faster image processing and better accuracy than its precursors. In all three R-CNN architectures, detection happens in two stages. The first stage uses something called a region proposal network (RPN) which is used to predict class indeterminate bounding box proposals.

These region proposals are found by sliding a window over a convolutional feature map output from a chosen shared convolutional layer inside the network at different positions in an $n \times n$ grid. At each sliding-window location, multiple bounding box proposals are generated using k number of *anchor boxes*[1].

In the second stage, these box proposals are used to crop features in the same shared convolutional layer as the sliding window was applied on. The cropped features are then fed to the remainder of the feature extractors in order to make classifications and bounding box revisions for each of the proposals (see figure 6.2).

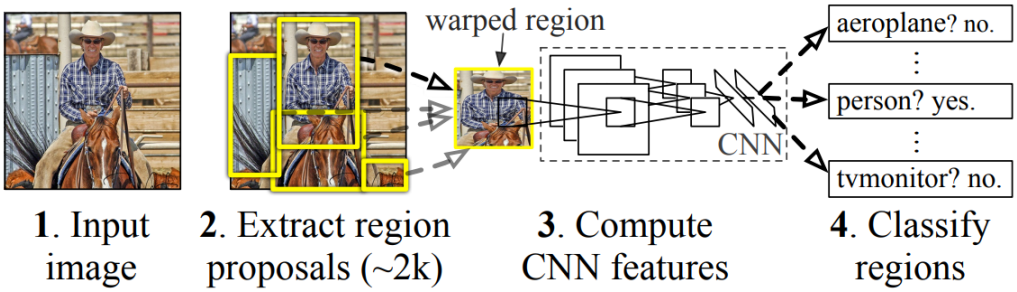

FIGURE 6.2: Overview of the object detection system used by all R-CNN type networks. *Figure from:* [64]

The R-CNN networks have had a big influence (particularly Faster R-CNN) on the development of new object detection networks and has led to a number networks based on the R-CNN systems (including SSD, YOLO and R-FCN).

6.2.2 R-FCN

Although Faster R-CNN is capable of processing images at much higher speeds than Fast R-CNN, the category-specific classifiers used to determine the category in the second step still has to be applied for each region proposal. To address this problem, Dai et al. proposed a new network called R-FCN (Region-based Fully Convolutional Network) [46] which has a similar architecture as Faster R-CNN, but instead uses a slightly different approach when cropping features in the second stage. In contrast to Faster R-CNN where box proposals are used to crop features in the same shared convolutional layer as the sliding window was applied on, R-FCN takes crops from the last layer of features prior to prediction (see figure 6.3).

Moving the cropping to the later parts of the network lowers the amount of computation that has to be done per region proposal, making the network capable of making predictions with comparable accuracy to Faster R-CNN at slightly faster running times.

[1]The technique of using anchor boxes (also called *default boxes* or *prior boxes*) was later picked up by many other object detection networks, like for example YOLO and SSD, and are more thoroughly explained in section 2.11.

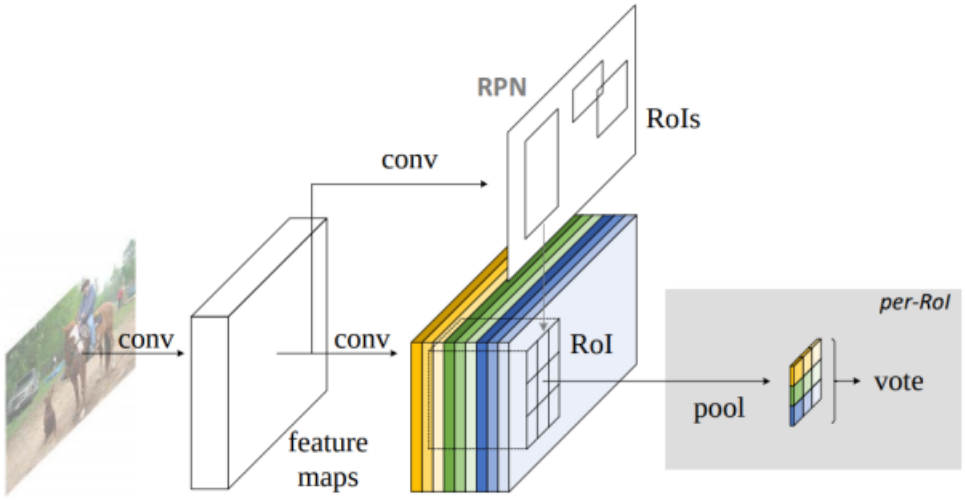

FIGURE 6.3: Overview of the object detection system used by the R-FCN network. *Figure from:* [46]

Chapter 7

Discussion and Future Work

In this chapter, the results outlined in section 5.4 are interpreted and discussed. The work conducted in this thesis underline several interesting and important research questions concerning the structure of object detection networks, driving-scene datasets, and the task of visual perception for autonomous vehicles, which are discussed here. Additionally, possible areas of future work are presented in the last section of the chapter.

7.1 Discussion

The first experiment was conducted to ensure that the implementation of SSD in Keras in this thesis project can be considered to be an accurate representation of the original network. This securing of sameness was conducted by training and evaluating the implemented SSD network on the same dataset (PASCAL) as the original SSD network presented in the paper by Liu et al. The complete architecture and all settings when training the Keras version of the network was implemented as similarly as possible to the original SSD implementation (see section 5.2.1 for details). The most notable difference between the two networks is that the original SSD was trained for twice as many training iterations as the Keras version in this thesis. This difference was an active choice made to save both time and GCP computation credits, reducing the training duration from about six days down to three. As can be seen in figure 5.4.1, the training loss curves where very close to converging already at 80,000 iterations, which presumably means that it is unlikely that any larger performance gains would have been achieved by continuing to train for 80,000 additional iterations. However, this difference in training iterations is most likely the cause of the small difference in mAP scores between the networks presented in figure 5.2.

The second conducted experiment is the main experiment of the thesis. Here, SSD is trained on the large and diverse driving-scene dataset BDD100K to study how the network performs when trained on a challenging and general dataset collected on real-life roads. As the results in section 5.4.2 show, the performance of SSD is not good enough to be applied in self-driving systems. This unsuitability for autonomous driving is established even more when looking at figure 7.1, which shows examples of detections (and missed detections) in real-world images made by the SSD network trained on the complete BDD100K dataset. The big performance difference when SSD is evaluated at IoU thresholds 0.7 and 0.5 (see table 5.3) indicates that SSD has problems with localization precision for predicted bounding boxes. This conclusion is drawn since the network makes a significant performance jump when the condition for bounding box localization precision is loosen. Another problem, which is a well known problem for one-stage detectors, is that SSD struggles with small sized objects which is made clear when looking at figures 5.8 and

7.1. The ability to detect small objects is very important for the application of autonomous driving and this problem is another argument for SSD not being suitable as a general object detector for self-driving systems. However, it should be noted that the network has a significantly higher AP score for the object class car than any other of the classes in the dataset. Seeing that more than half the objects in BDD100K are cars, the low AP scores for the other classes could partly be explained by them not having enough of samples in the dataset.

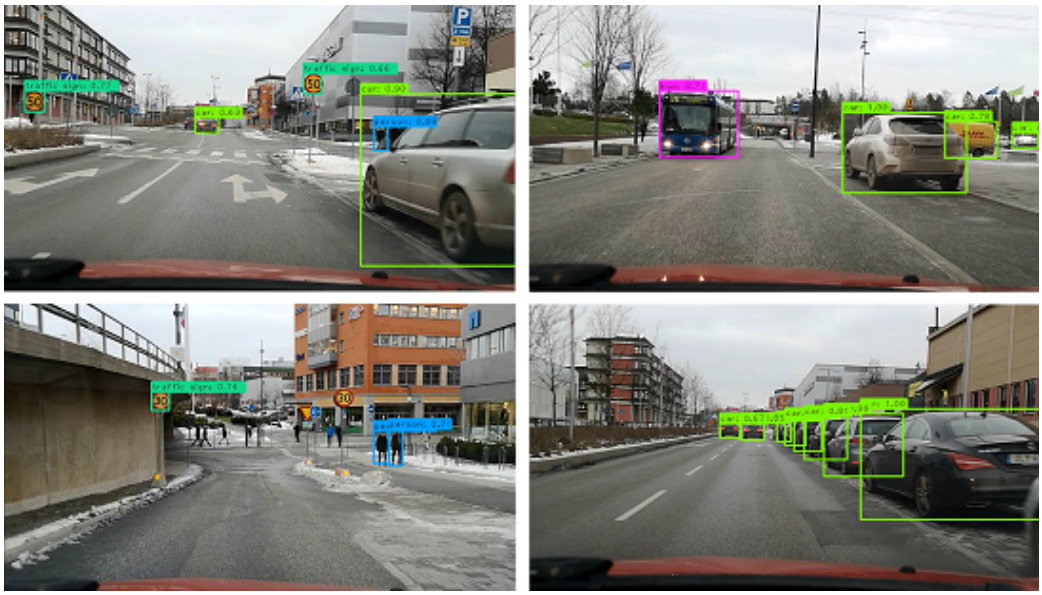

FIGURE 7.1: Four frames sampled from a video which was recorded outside the office of Cybercom in Kista, Stockholm, showing detections made by the trained SSD network.

Although the network has a reasonable detection rate for cars in BDD100K, the results of the experiment still show that real-time object detectors today most likely are not good enough to be used as general object detecting systems in self-driving cars. A general object detector needs to have high accuracy for all objects it has been train to detect, and work adequately in all weather conditions during all hours of the day, which is not the case for SSD when trained on BDD100K. One alternative could be to train a set of networks to be used in different conditions (sub-domains of the dataset like images collected during daytime, night, rain, snow, etc.) which could give a substantial boost in performance. However, this alternative is left for future work.

The third and last experiment consisted of training and comparing two versions of SSD, one version trained with synthetic data, and one version trained on a subset of BDD100K. One concern with using synthetic data is that it is dependent on the data generation model, and therefore may introduce bias and not generalize well enough to the physical world. This kind of bias could partly explain the performance different seen in the cross evaluation on the KITTI dataset seen in table 5.4 and 5.5. One possibility is that it would require many more simulated images to achieve similar performance as with real-world images. This requirement is however not a major problem since the main purpose with using synthetic data is that it is easy to automatically produce and annotate massive amounts of virtual images.

7.1.1 Research Questions

Here, answers to the research questions stated in chapter 3 are given using the results and inferences from the experiments conducted and explained in section 5.4.

- *How well does a state-of-the-art real-time capable object detecting deep neural network perform when trained on a challenging and diverse driving-scene dataset like Berkeley Deep Drive 100k (BDD100K)?*

  BDD100K is a large dataset consisting of 1.8 million bounding box annotated distinct objects with appearances and contexts from ten different classes. It can be seen as a representation of the physical world a self-driving system would encounter, with images collected during a variety of driving conditions. The results of the experiments in this thesis show that SSD, which is a cutting-edge object detecting network, does not perform adequately on the a majority of the classes in the dataset. The network produce low AP scores for all classes except the class *car*, with an overall mAP of 0.167. One of the major causes of this low detection performance is probably the suppression of both trainable parameters and input size of the network to make SSD operate at real-time frame rates. Moreover, most one-stage detectors struggle with detecting smaller objects, and as discussed earlier, BDD100K consist of a large number of small sized objects (see figure 4.4).

- *Is the performance of real-time capable object detecting deep neural networks good enough to be able to be utilized in self-driving systems?*

  The results of the experiments in this thesis (conducted on the real-time capable object detecting deep neural network SSD) show that SSD most likely is not good enough to be used as a general object detecting system in self-driving cars. A general object detector utilized in autonomous vehicles needs to have high accuracy for all objects it has been train on in all weather conditions and during all hours of the day, which is not the case for SSD when trained on BDD100K. However, for some applications the network could have an acceptable detection rate for cars, with an AP score of 0.473 evaluated at an IoU of 0.5.

- *How big is the performance gap between real-time capable object detecting deep neural networks and the top performing (but slower) networks trained on the BDD100K dataset?*

  Modern real-time capable object detection networks, such as the SSD network, achieve their fast performance at the expense of detection rate and accuracy. This is made clear in table 5.3 where the average precisions for all predicted classes on the test data in BDD100K are presented for the runner up in the 2018 WAD road object detection competition, compared with the results of the implemented Keras version of SSD in this thesis. Here, the second place network in the competition called CFENet800-MS scored a mAP of 0.297, evaluated at an IoU of 0.7, which is more than three times higher than the SSD mAP score of 0.079. This performance difference show that the gap between real-time capable object detecting deep neural networks and networks designed only for detection performance, is considerably large.

- *Is the performance of a real-time capable object detecting deep neural network trained on synthetic driving-scene data comparable with the same network trained on real driving-scene data? If not, what is the gap between available real and synthetic driving scene data, from the perspective of object detection architectures?*

  When two versions of SSD (one version trained on the virtual dataset FCAV, and the other version is trained on a subset of BDD100K) were evaluated on their own test sets, the virtual dataset FCAV scored an AP of 0.507, compared to an AP of 0.461 for BDD100K. This shows that SSD is capable of learning to detect objects in both real-life images as well as in images from virtual environments. Looking at the AP scores for the two networks when evaluated on KITTI, it is clear that the BDD100K trained SSD network outperforms the version trained on FCAV, when evaluating on images collected in the physical world. However, with an AP score of 0.319 for FCAV and 0.585 for BDD100K, this experiment shows that it is possible to train an object detecting deep neural network to recognize real-life objects using only synthetic computer generated images.

7.2 Future Work

One suitable task for future work would be to investigate how using different base networks (see section 2.11) affects the speed of SSD. The original SSD implementation uses VGG16 [33] as a base network. However, seeing that this SSD implementation spend about 80% of the forward time on the base network, and that faster and better CNNs are being developed continuously, using another base network could further improve the speed of SSD. If this can be achieved, SSD could still be capable of real-time speeds even with a larger input layer (taking images with higher resolution as input), and therefore presumably operate with higher accuracy than the original implementation.

Considering that BDD100K is a diverse and general driving-scene dataset which proved to be challenging for SSD, another possible approach for future work could be to investigate how well SSD perform on different sub-domains of BDD100K (*e.g*, daytime, night, rain, snow, etc.). Changing the trained parameters of the network is an easy task which can be performed quickly if a car detects that new conditions have occurred. If one can achieve a significant performance boost by training multiple versions of SSD on different sub-domains there could therefore still be areas of use for SSD in a self-driving system.

Training SSD using synthetic datasets from other increasingly realistic virtual worlds is something that also could be studied in future work. Since the virtual dataset FCAV used in this thesis was simulated and collected in the video game Grand Theft Auto V, which was released more than five years ago[1], using newer more realistic synthetic driving-scene images would presumably result in better accuracy when the network is tested on real-life data.

Another exciting aspect of virtual worlds which could be built upon this project is to use virtual environments, not only to collect synthetic data, but also for the development and validation of highly automated end-to-end image based driving systems. One of the most complex challenges encountered when building complete autonomous driving systems is the variability of the environment and its impact

[1]Grand Theft Auto V was released in September 2013 for PlayStation 3 and Xbox 360, in November 2014 for PlayStation 4 and Xbox One, and in April 2015 for Microsoft Windows.

on features. Therefore, it is critical that an end-to-end system and its features are validated against maximum variability. Building and testing autonomous vehicles in photorealistic simulations present a near-infinite variety of conditions and events — before the vehicles even reach real-world scenarios. Thus, virtual worlds could be a safer, more scalable, and more cost-effective way to bring self-driving cars to our roads.

Chapter 8

Conclusion

In this thesis, the objective has been to study the perception problem in the contexts of real-time object detection for autonomous vehicles. Self-driving systems are commonly categorized into three subsystems; perception, planning, and control, where the perception system is responsible for translating raw sensor data into a model of the surrounding environment. To study this problem, a cutting-edge real-time object detection deep neural network called SSD was trained and evaluated on both real and virtual driving-scene data.

The results in this thesis show that modern real-time capable object detection networks, such as the SSD network, achieve their fast performance at the expense of detection rate and accuracy. Further development in both hardware and software technologies will presumably result in a better trade-off between the run time and detection rate. However, as the technologies stand today, general real-time object detection networks do not seem to be suitable for high precision tasks, such as visual perception for autonomous vehicles. Although the results show that SSD should not be used within a self-driving system, the network could possible be used as assistants for other tasks, like driver warning alerts using detections with high probabilities or augmented reality techniques visualized with onboard screens.

Additionally, the results show that synthetic driving scene data possibly could be used to train object detection networks used by self-driving systems, at least in the future. The open sourced virtual dataset, FCAV, used for this project, is simulated and collected in a more than five year old video game. Today, a number of different increasingly realistic virtual worlds are being built (*e.g.*, Carla, Deepdrive , Virtual KITTI, AirSim, Truevision, ParallelEye, SynCity, Parallel Domain). This means that, in the near future, an enormous amount of automatically annotated (and presumably high quality) data could be made available from such virtual worlds. This new higher quality synthetic data could in the near future be used to train deep object detecting networks utilized by self-driving systems.

Bibliography

[1] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE International Conference on Computer Vision*. Vol. 2015 Inter. 2015, pp. 1026–1034. ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.123. arXiv: 1502.01852.

[2] Zhong-Qiu Zhao et al. *Object Detection with Deep Learning: A Review*. Tech. rep. 2018. arXiv: 1807.05511v1.

[3] Joel Janai et al. "Computer Vision for Autonomous Vehicles: Problems, Datasets and State-of-the-Art". In: (2017). ISSN: 10495258. DOI: 10.1007/1-4020-3858-8_15. arXiv: 1704.05519.

[4] Scott Pendleton et al. "Perception, Planning, Control, and Coordination for Autonomous Vehicles". In: *Machines* 5.1 (2017), p. 6. ISSN: 2075-1702. DOI: 10.3390/machines5010006. URL: http://www.mdpi.com/2075-1702/5/1/6.

[5] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. Tech. rep. 2016. arXiv: 1506.02640v5. URL: https://pjreddie.com/darknet/yolov1.

[6] Wei Liu et al. "SSD: Single shot multibox detector". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9905 LNCS (2016), pp. 21–37. ISSN: 16113349. DOI: 10.1007/978-3-319-46448-0_2. arXiv: 1512.02325.

[7] Juergen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 08936080. DOI: 10.1016/j.neunet.2014.09.003. arXiv: 1404.7828v4.

[8] Joseph Redmon and Ali Farhadi. *YOLO9000: Better, Faster, Stronger*. Tech. rep. 2016. arXiv: 1612.08242v1. URL: https://pjreddie.com/darknet/yolov2.

[9] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. Tech. rep. 2018. arXiv: 1804.02767v1. URL: https://pjreddie.com/yolo.

[10] Shifeng Zhang et al. "Single-Shot Refinement Neural Network for Object Detection". In: (2017). DOI: 10.1109/CVPR.2018.00442. arXiv: 1711.06897.

[11] Tsung Yi Lin et al. *Focal Loss for Dense Object Detection*. Tech. rep. 2017, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324. arXiv: 1708.02002.

[12] Fisher Yu et al. *BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling*. Tech. rep. 2018. arXiv: 1805.04687v1.

[13] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. ISSN: 15731405. DOI: 10.1007/s11263-015-0816-y. arXiv: 1409.0575.

[14] Mark Everingham et al. "The PASCAL Visual Object Classes (VOC) Challenge". In: (2009). URL: http://host.robots.ox.ac.uk/pascal/VOC/pubs/everingham10.pdf.

[15] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. Tech. rep. 2015. arXiv: 1405.0312v3.

[16] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations (ICRL)* (2015), pp. 1–14. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.09.005. arXiv: 1409.1556.

[17] Adrien Gaidon et al. *Virtual Worlds as Proxy for Multi-Object Tracking Analysis*. Tech. rep. 2016. arXiv: 1605.06457v1. URL: http://www.xrce.xerox.com/.

[18] Antonio M. López et al. "Training my car to see using virtual worlds". In: *Image and Vision Computing* 68 (2017), pp. 102–118. ISSN: 02628856. DOI: 10.1016/j.imavis.2017.07.007.

[19] Matthew Johnson-Roberson et al. *Driving in the Matrix: Can Virtual Worlds Replace Human-Generated Annotations for Real World Tasks?* Tech. rep. 2016. arXiv: 1610.01983.

[20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594.

[21] Dennis F. Kibler and Pat Langley. "Machine Learning As an Experimental Science". In: *Proceedings of the 3rd European Conference on European Working Session on Learning*. EWSL'88. Glasgow, UK: Pitman Publishing, Inc., 1988, pp. 81–92. ISBN: 0-273-08800-9. URL: http://dl.acm.org/citation.cfm?id=3108771.3108779.

[22] Fei-Fei LI, Justin Johnson, and Serena Yeung. *Stanford online course: Convolutional Neural Networks for Visual Recognition. Lecture 6 - Training Neural Networks I.* Stanford vision and learning lab, 2018. URL: http://cs231n.stanford.edu/syllabus.html.

[23] Hoon Chung, Sung Lee, and Jeon Park. "Deep neural network using trainable activation functions". In: (2016), pp. 348–352. DOI: 10.1109/IJCNN.2016.7727219.

[24] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: (2017). arXiv: 1710.05941v2.

[25] Youshen Xia and Jun Wang. *A General Methodology for Designing Globally Convergent Optimization Neural Networks*. Tech. rep. 6. 1998. URL: https://pdfs.semanticscholar.org/872d/12b46c39565540ff4daa74960504d0e7c9c6.pdf.

[26] Hao Li et al. "Visualizing the Loss Landscape of Neural Nets". In: (2017). ISSN: 1752-0894. DOI: 10.1038/NGEO921. arXiv: 1712.09913.

[27] Katarzyna Janocha and Wojciech Marian Czarnecki. "On Loss Functions for Deep Neural Networks in Classification". In: (2017). ISSN: 20838476. DOI: 10.4467/20838476SI.16.004.6185. arXiv: 1702.05659.

[28] Christian Hennig and Mahmut Kutlukaya. *SOME THOUGHTS ABOUT THE DESIGN OF LOSS FUNCTIONS*. Tech. rep. 1. 2007, pp. 19–39. URL: https://www.ine.pt/revstat/pdf/rs070102.pdf.

[29] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: (1986). DOI: 10.1038/323533a0.

[30] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: (2016). ISSN: 0006341X. DOI: 10.1111/j.0006-341X.1999.00591.x. arXiv: 1609.04747.

[31] Fei-Fei LI, Justin Johnson, and Serena Yeung. *Stanford online course: Convolutional Neural Networks for Visual Recognition. Lecture 3 - Loss Functions and Optimization.* Stanford vision and learning lab, 2018. URL: http://cs231n.stanford.edu/syllabus.html.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[33] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations (ICRL)* (2015), pp. 1–14. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.09.005. arXiv: 1409.1556.

[34] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083.

[35] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. Tech. rep. 2016. arXiv: 1506.01497v3.

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

[37] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[38] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[39] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[40] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 1st. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN: 1848829345, 9781848829343.

[41] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). ISSN: 1664-1078. DOI: 10.3389/fpsyg.2013.00124. arXiv: 1512.03385.

[42] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision–ECCV 2014* 8689 (2014), pp. 818–833. ISSN: 978-3-319-10589-5. DOI: 10.1007/978-3-319-10590-1_53. arXiv: 1311.2901.

[43] Christian Szegedy et al. *Going deeper with convolutions*. Tech. rep. 2014. arXiv: 1409.4842v1.

[44] Fei-Fei LI, Justin Johnson, and Serena Yeung. *Stanford online course: Convolutional Neural Networks for Visual Recognition. Lecture 5 - Convolutional Neural Networks.* Stanford vision and learning lab, 2018. URL: http://cs231n.stanford.edu/syllabus.html.

[45] Heikki Huttunen. *Convolutional Networks, Recurrent Nets*. Tampere University of Technology, 2018. URL: http://www.cs.tut.fi/courses/SGN-41007/slides/Lecture6b.pdf.

[46] Jifeng Dai et al. "R-FCN: Object Detection via Region-based Fully Convolutional Networks". In: (2016). ISSN: 15206149. DOI: 10.1109/ICASSP.2017.7952132. arXiv: 1605.06409.

[47] Rob Williams. *Real-Time Systems Development*. 1st. Elsevier Science & Technology, 2005. ISBN: 0750664711, 9780750664714.

[48] Tsung Yi Lin et al. *Feature pyramid networks for object detection*. Tech. rep. 2017, pp. 936–944. DOI: 10.1109/CVPR.2017.106. arXiv: 1612.03144.

[49] Kaiming He et al. *Mask R-CNN*. Tech. rep. 2018. arXiv: 1703.06870v3.

[50] Wenzhe Yang Quan Liao Guangming Shi Jinjian Wu Guimei Cao Xuemei Xie. "Feature-fused SSD: fast detection for small objects". In: *Proc.SPIE* 10615 (2018), pp. 10615 –10615 –8. DOI: 10.1117/12.2304811. URL: https://doi.org/10.1117/12.2304811.

[51] Jan Hendrik Hosang, Rodrigo Benenson, and Bernt Schiele. "Learning non-maximum suppression". In: (2017). arXiv: 1705.02950.

[52] Mark Everingham et al. "The PASCAL Visual Object Classes Challenge: A Retrospective". In: (2013). URL: http://host.robots.ox.ac.uk/pascal/VOC/pubs/everingham15.pdf.

[53] *The Google Cloud Adoption Framework*. Tech. rep. 2018. URL: https://services.google.com/fh/files/misc/adoption_framework_whitepaper_nov12_final.pdf.

[54] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[55] François Chollet et al. *Keras*. https://keras.io. 2015. URL: https://github.com/keras-team/keras.

[56] Qijie Zhao et al. *CFENet: An Accurate and Efficient Single-Shot Object Detector for Autonomous Driving*. 2018. arXiv: 1806.09790.

[57] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012. URL: http://www.cvlibs.net/publications/Geiger2012CVPR.pdf.

[58] Shifeng Zhang et al. "Single-Shot Refinement Neural Network for Object Detection". In: (2018). DOI: 10.1109/CVPR.2018.00442. URL: https://www.researchgate.net/publication/321180719_Single-Shot_Refinement_Neural_Network_for_Object_Detection.

[59] Andrew G. Howard. *Some Improvements on Deep Convolutional Neural Network Based Image Classification*. 2013. arXiv: 1312.5402.

[60] George Philipp, Dawn Song, and Jaime G. Carbonell. *Gradients explode - Deep Networks are shallow - ResNet explained*. 2018. URL: https://openreview.net/forum?id=HkpYwMZRb.

[61] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980.

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

[63] Mark Everingham and John Winn. *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Development Kit*. 2007. URL: http://host.robots.ox.ac.uk/pascal/VOC/voc2007/devkit_doc_07-Jun-2007.pdf.

[64] Ross B. Girshick et al. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524.

Appendix A

SSD Architecture

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 300, 300, 3)	0	
identity_layer (Lambda)	(None, 300, 300, 3)	0	input_1[0][0]
input_mean_normalization (Lambd	(None, 300, 300, 3)	0	identity_layer[0][0]
input_channel_swap (Lambda)	(None, 300, 300, 3)	0	input_mean_normalization[0][0]
conv1_1 (Conv2D)	(None, 300, 300, 64)	1792	input_channel_swap[0][0]
conv1_2 (Conv2D)	(None, 300, 300, 64)	36928	conv1_1[0][0]
pool1 (MaxPooling2D)	(None, 150, 150, 64)	0	conv1_2[0][0]
conv2_1 (Conv2D)	(None, 150, 150, 128	73856	pool1[0][0]
conv2_2 (Conv2D)	(None, 150, 150, 128	147584	conv2_1[0][0]
pool2 (MaxPooling2D)	(None, 75, 75, 128)	0	conv2_2[0][0]
conv3_1 (Conv2D)	(None, 75, 75, 256)	295168	pool2[0][0]
conv3_2 (Conv2D)	(None, 75, 75, 256)	590080	conv3_1[0][0]
conv3_3 (Conv2D)	(None, 75, 75, 256)	590080	conv3_2[0][0]
pool3 (MaxPooling2D)	(None, 38, 38, 256)	0	conv3_3[0][0]
conv4_1 (Conv2D)	(None, 38, 38, 512)	1180160	pool3[0][0]
conv4_2 (Conv2D)	(None, 38, 38, 512)	2359808	conv4_1[0][0]
conv4_3 (Conv2D)	(None, 38, 38, 512)	2359808	conv4_2[0][0]
pool4 (MaxPooling2D)	(None, 19, 19, 512)	0	conv4_3[0][0]
conv5_1 (Conv2D)	(None, 19, 19, 512)	2359808	pool4[0][0]
conv5_2 (Conv2D)	(None, 19, 19, 512)	2359808	conv5_1[0][0]
conv5_3 (Conv2D)	(None, 19, 19, 512)	2359808	conv5_2[0][0]
pool5 (MaxPooling2D)	(None, 19, 19, 512)	0	conv5_3[0][0]
fc6 (Conv2D)	(None, 19, 19, 1024)	4719616	pool5[0][0]
fc7 (Conv2D)	(None, 19, 19, 1024)	1049600	fc6[0][0]
conv6_1 (Conv2D)	(None, 19, 19, 256)	262400	fc7[0][0]
conv6_padding (ZeroPadding2D)	(None, 21, 21, 256)	0	conv6_1[0][0]
conv6_2 (Conv2D)	(None, 10, 10, 512)	1180160	conv6_padding[0][0]

```
conv7_1 (Conv2D)                (None, 10, 10, 128)  65664       conv6_2[0][0]
__________________________________________________________________________________________________
conv7_padding (ZeroPadding2D)   (None, 12, 12, 128)  0           conv7_1[0][0]
__________________________________________________________________________________________________
conv7_2 (Conv2D)                (None, 5, 5, 256)    295168      conv7_padding[0][0]
__________________________________________________________________________________________________
conv8_1 (Conv2D)                (None, 5, 5, 128)    32896       conv7_2[0][0]
__________________________________________________________________________________________________
conv8_2 (Conv2D)                (None, 3, 3, 256)    295168      conv8_1[0][0]
__________________________________________________________________________________________________
conv9_1 (Conv2D)                (None, 3, 3, 128)    32896       conv8_2[0][0]
__________________________________________________________________________________________________
conv4_3_norm (L2Normalization)  (None, 38, 38, 512)  512         conv4_3[0][0]
__________________________________________________________________________________________________
conv9_2 (Conv2D)                (None, 1, 1, 256)    295168      conv9_1[0][0]
__________________________________________________________________________________________________
conv4_3_norm_mbox_conf (Conv2D) (None, 38, 38, 8)    36872       conv4_3_norm[0][0]
__________________________________________________________________________________________________
fc7_mbox_conf (Conv2D)          (None, 19, 19, 12)   110604      fc7[0][0]
__________________________________________________________________________________________________
conv6_2_mbox_conf (Conv2D)      (None, 10, 10, 12)   55308       conv6_2[0][0]
__________________________________________________________________________________________________
conv7_2_mbox_conf (Conv2D)      (None, 5, 5, 12)     27660       conv7_2[0][0]
__________________________________________________________________________________________________
conv8_2_mbox_conf (Conv2D)      (None, 3, 3, 8)      18440       conv8_2[0][0]
__________________________________________________________________________________________________
conv9_2_mbox_conf (Conv2D)      (None, 1, 1, 8)      18440       conv9_2[0][0]
__________________________________________________________________________________________________
conv4_3_norm_mbox_loc (Conv2D)  (None, 38, 38, 16)   73744       conv4_3_norm[0][0]
__________________________________________________________________________________________________
fc7_mbox_loc (Conv2D)           (None, 19, 19, 24)   221208      fc7[0][0]
__________________________________________________________________________________________________
conv6_2_mbox_loc (Conv2D)       (None, 10, 10, 24)   110616      conv6_2[0][0]
__________________________________________________________________________________________________
conv7_2_mbox_loc (Conv2D)       (None, 5, 5, 24)     55320       conv7_2[0][0]
__________________________________________________________________________________________________
conv8_2_mbox_loc (Conv2D)       (None, 3, 3, 16)     36880       conv8_2[0][0]
__________________________________________________________________________________________________
conv9_2_mbox_loc (Conv2D)       (None, 1, 1, 16)     36880       conv9_2[0][0]
__________________________________________________________________________________________________
conv4_3_norm_mbox_conf_reshape  (None, 5776, 2)      0           conv4_3_norm_mbox_conf[0][0]
__________________________________________________________________________________________________
fc7_mbox_conf_reshape (Reshape) (None, 2166, 2)      0           fc7_mbox_conf[0][0]
__________________________________________________________________________________________________
conv6_2_mbox_conf_reshape (Resh (None, 600, 2)       0           conv6_2_mbox_conf[0][0]
__________________________________________________________________________________________________
conv7_2_mbox_conf_reshape (Resh (None, 150, 2)       0           conv7_2_mbox_conf[0][0]
__________________________________________________________________________________________________
conv8_2_mbox_conf_reshape (Resh (None, 36, 2)        0           conv8_2_mbox_conf[0][0]
__________________________________________________________________________________________________
conv9_2_mbox_conf_reshape (Resh (None, 4, 2)         0           conv9_2_mbox_conf[0][0]
__________________________________________________________________________________________________
conv4_3_norm_mbox_priorbox (Anc (None, 38, 38, 4, 8) 0           conv4_3_norm_mbox_loc[0][0]
__________________________________________________________________________________________________
fc7_mbox_priorbox (AnchorBoxes) (None, 19, 19, 6, 8) 0           fc7_mbox_loc[0][0]
__________________________________________________________________________________________________
conv6_2_mbox_priorbox (AnchorBo (None, 10, 10, 6, 8) 0           conv6_2_mbox_loc[0][0]
__________________________________________________________________________________________________
conv7_2_mbox_priorbox (AnchorBo (None, 5, 5, 6, 8)   0           conv7_2_mbox_loc[0][0]
__________________________________________________________________________________________________
conv8_2_mbox_priorbox (AnchorBo (None, 3, 3, 4, 8)   0           conv8_2_mbox_loc[0][0]
__________________________________________________________________________________________________
conv9_2_mbox_priorbox (AnchorBo (None, 1, 1, 4, 8)   0           conv9_2_mbox_loc[0][0]
__________________________________________________________________________________________________
mbox_conf (Concatenate)         (None, 8732, 2)      0           conv4_3_norm_mbox_conf_reshape[0]
                                                                 fc7_mbox_conf_reshape[0][0]
                                                                 conv6_2_mbox_conf_reshape[0][0]
                                                                 conv7_2_mbox_conf_reshape[0][0]
                                                                 conv8_2_mbox_conf_reshape[0][0]
                                                                 conv9_2_mbox_conf_reshape[0][0]
__________________________________________________________________________________________________
conv4_3_norm_mbox_loc_reshape ( (None, 5776, 4)      0           conv4_3_norm_mbox_loc[0][0]
```

```
____________________________________________________________________________________________________
fc7_mbox_loc_reshape (Reshape)  (None, 2166, 4)      0           fc7_mbox_loc[0][0]
____________________________________________________________________________________________________
conv6_2_mbox_loc_reshape (Resha (None, 600, 4)       0           conv6_2_mbox_loc[0][0]
____________________________________________________________________________________________________
conv7_2_mbox_loc_reshape (Resha (None, 150, 4)       0           conv7_2_mbox_loc[0][0]
____________________________________________________________________________________________________
conv8_2_mbox_loc_reshape (Resha (None, 36, 4)        0           conv8_2_mbox_loc[0][0]
____________________________________________________________________________________________________
conv9_2_mbox_loc_reshape (Resha (None, 4, 4)         0           conv9_2_mbox_loc[0][0]
____________________________________________________________________________________________________
conv4_3_norm_mbox_priorbox_resh (None, 5776, 8)      0           conv4_3_norm_mbox_priorbox[0][0]
____________________________________________________________________________________________________
fc7_mbox_priorbox_reshape (Resh (None, 2166, 8)      0           fc7_mbox_priorbox[0][0]
____________________________________________________________________________________________________
conv6_2_mbox_priorbox_reshape ( (None, 600, 8)       0           conv6_2_mbox_priorbox[0][0]
____________________________________________________________________________________________________
conv7_2_mbox_priorbox_reshape ( (None, 150, 8)       0           conv7_2_mbox_priorbox[0][0]
____________________________________________________________________________________________________
conv8_2_mbox_priorbox_reshape ( (None, 36, 8)        0           conv8_2_mbox_priorbox[0][0]
____________________________________________________________________________________________________
conv9_2_mbox_priorbox_reshape ( (None, 4, 8)         0           conv9_2_mbox_priorbox[0][0]
____________________________________________________________________________________________________
mbox_conf_softmax (Activation)  (None, 8732, 2)      0           mbox_conf[0][0]
____________________________________________________________________________________________________
mbox_loc (Concatenate)          (None, 8732, 4)      0           conv4_3_norm_mbox_loc_reshape[0][
                                                                 fc7_mbox_loc_reshape[0][0]
                                                                 conv6_2_mbox_loc_reshape[0][0]
                                                                 conv7_2_mbox_loc_reshape[0][0]
                                                                 conv8_2_mbox_loc_reshape[0][0]
                                                                 conv9_2_mbox_loc_reshape[0][0]
____________________________________________________________________________________________________
mbox_priorbox (Concatenate)     (None, 8732, 8)      0           conv4_3_norm_mbox_priorbox_reshap
                                                                 fc7_mbox_priorbox_reshape[0][0]
                                                                 conv6_2_mbox_priorbox_reshape[0][
                                                                 conv7_2_mbox_priorbox_reshape[0][
                                                                 conv8_2_mbox_priorbox_reshape[0][
                                                                 conv9_2_mbox_priorbox_reshape[0][
____________________________________________________________________________________________________
predictions (Concatenate)       (None, 8732, 14)     0           mbox_conf_softmax[0][0]
                                                                 mbox_loc[0][0]
                                                                 mbox_priorbox[0][0]
====================================================================================================
Total params: 23,745,908
Trainable params: 23,745,908
Non-trainable params: 0
____________________________________________________________________________________________________
```

Appendix B

Original Project Code

In this appendix all original code written for this project is presented. The complete code base for the project can be found at `https://gitlab.com/roger_cybercom/ssd-master-thesis`.

B.1 JSON to XML Annotations

```
1  '''
2  Convert JSON annotations to PASCAL xml annotations
3  Master thesis project Cybercom
4  Roger Kalliomaki
5  '''
6
7  import argparse
8  import json
9  import lxml.etree as ET
10 import os
11
12 # Set classes, xml folder path, imageset file path, JSON
13 # file path, and boolean SORT_OUT which defines if images
14 # without annotations should be included
15 CATEGORIES = ['bus', 'traffic light', 'traffic sign', 'person', 'bike', 'truck', 'motor', /
       'car', 'train', 'rider']
16 XML_FOLDER_PATH = "./Annotations/"
17 IMAGESET_FILE_PATH = "./ImageSets/Main/train.txt"
18 JSON_FILE_PATH = "./bdd100k_labels_images_train.json"
19 SORT_OUT = True
20
21 # Load JSON file
22 frames = json.load(open(JSON_FILE_PATH, 'r'))
23 imageset_file = open(IMAGESET_FILE_PATH, "w")
24
25 # Go through all annotations in JSON file
26 write_to_file = True
27 for frame in frames:
28     if SORT_OUT:
29         write_to_file = False
30
31     # Set general image information
32     annotation = ET.Element("annotation")
33     ET.SubElement(annotation, "folder").text = "BDD100K"
34     ET.SubElement(annotation, "filename").text = frame['name']
35     source = ET.SubElement(annotation, "source")
36     ET.SubElement(source, "database").text = "Berkeley Deep Drive Database"
```

```
37          ET.SubElement(source, "annotation").text = "PASCAL␣VOC2007"
38          ET.SubElement(source, "image").text = "Berkeley␣Deep␣Drive␣Database"
39          ET.SubElement(source, "flickrid").text = "Berkeley␣Deep␣Drive"
40          owner = ET.SubElement(annotation, "owner")
41          ET.SubElement(owner, "flickrid").text = "BDD100K"
42          ET.SubElement(owner, "name").text = "Berkeley␣Deep␣Drive"
43          size = ET.SubElement(annotation, "size")
44          ET.SubElement(size, "width").text = "1280"
45          ET.SubElement(size, "height").text = "720"
46          ET.SubElement(size, "depth").text = "3"
47          ET.SubElement(annotation, "segmented").text = "0"
48
49          # Go throug all objects in image annotation
50          for label in frame['labels']:
51              if 'box2d' not in label:
52                  continue
53              xy = label['box2d']
54              if xy['x1'] >= xy['x2'] or xy['y1'] >= xy['y2']:
55                  continue
56              if SORT_OUT and label['category'] in CATEGORIES:
57                  write_to_file = True
58              else:
59                  continue
60              # Bounding box coordinates
61              x1 = str(int(round(xy['x1'])))
62              x2 = str(int(round(xy['x2'])))
63              y1 = str(int(round(xy['y1'])))
64              y2 = str(int(round(xy['y2'])))
65              # General object information
66              obj = ET.SubElement(annotation, "object")
67              ET.SubElement(obj, "name").text = label['category']
68              ET.SubElement(obj, "pose").text = "Unspecified"
69              ET.SubElement(obj, "truncated").text = "1" if label['attributes']['truncated'] /
                    else "0"
70              ET.SubElement(obj, "difficult").text = "0"
71              bndbox = ET.SubElement(obj, "bndbox")
72              ET.SubElement(bndbox, "xmin").text = x1
73              ET.SubElement(bndbox, "ymin").text = y1
74              ET.SubElement(bndbox, "xmax").text = x2
75              ET.SubElement(bndbox, "ymax").text = y2
76
77          # Write to file
78          if write_to_file:
79              frame_filename = os.path.splitext(frame['name'])[0]
80              file_tree = ET.ElementTree(annotation)
81              file_tree.write(XML_FOLDER_PATH + frame_filename + ".xml", /
                    pretty_print=True)
82              imageset_file.write(frame_filename + "\n")
83
84  imageset_file.close()
```

B.2 Determine and Save Bounding Box Sizes

```
1  '''
2  Save bounding box sizes in dataset in csv file
```

```
3  Master thesis project Cybercom
4  Roger Kalliomaki
5  '''
6
7  from lxml.etree import parse
8  from os import listdir
9  from os.path import isfile, join
10 import random
11
12 # Set dataset, nr of images and the boolean USE_ALL_OBJECTS which defines if
13 # all objects should be used or only cars.
14 DATASET = 'BDD100K'
15 USE_MAX_NR_IMAGES = True
16 MAX_NR_IMAGES = 5000
17 USE_ALL_OBJECTS = True
18
19 # Set correct annotations path depending on if complete dataset should be used.
20 if USE_ALL_OBJECTS:
21     ANNOTATIONS_PATH = './Annotations_all_classes/'
22 elif DATASET=='BDD100K':
23     ANNOTATIONS_PATH = './Annotations_cars/'
24 else:
25     ANNOTATIONS_PATH = './Annotations/'
26
27 # Read xml files and shuffle them.
28 xml_files = [f for f in listdir(ANNOTATIONS_PATH) if /
       isfile(join(ANNOTATIONS_PATH, f))]
29 random.shuffle(xml_files)
30 # Set nr of files to use
31 nr_files = len(xml_files)
32 if USE_MAX_NR_IMAGES:
33     nr_files = MAX_NR_IMAGES
34
35 # Go throug files and save bounding box sizes in a csv file.
36 if USE_ALL_OBJECTS:
37     result_file = open('./'+DATASET+'_all_classes_bb_sizes.csv', 'w')
38 else:
39     result_file = open('./'+DATASET+'_cars_bb_sizes.csv', 'w')
40 counter = 0
41 for xml_file_name in xml_files:
42     xmlTree = parse(ANNOTATIONS_PATH + xml_file_name)
43     for element in xmlTree.findall('object'):
44         if USE_ALL_OBJECTS or element.find('name').text == 'car' or /
               element.find('name').text == 'Car':
45             xmin = int(element.find('bndbox').find('xmin').text)
46             xmax = int(element.find('bndbox').find('xmax').text)
47             ymin = int(element.find('bndbox').find('ymin').text)
48             ymax = int(element.find('bndbox').find('ymax').text)
49             dx = xmax-xmin
50             dy = ymax-ymin
51             csv_string = '%d,%d\n' % (dx,dy)
52             result_file.write(csv_string)
53     if USE_MAX_NR_IMAGES and counter == MAX_NR_IMAGES:
54         break
55     counter += 1
56     print('%d/%d' % (counter,nr_files))
```

B.3 Plot Bounding Box Sizes

```
'''
Plot bounding box sizes saved in given csv file
Master thesis project Cybercom
Roger Kalliomaki
'''

import matplotlib.pyplot as plt
import numpy as np

# Set dataset and the boolean USE_ALL_OBJECTS which defines if
# all objects should be used or only cars.
DATASET = 'BDD100K'
USE_ALL_OBJECTS = True

# Load csv file into numpy array
if USE_ALL_OBJECTS:
    FILE_PATH = './' + DATASET + '_all_classes_bb_sizes.csv'
else:
    FILE_PATH = './' + DATASET + '_cars_bb_sizes.csv'
bb_sizes = np.genfromtxt(FILE_PATH, delimiter=",", dtype=int)

# Save widths and heights from numpy array wiht bounding box sizes
dx = []
dy = []
for dx_dy in bb_sizes:
    dx.append(dx_dy[0])
    dy.append(dx_dy[1])

# Plot scatter plot
plt.scatter(dx,dy,s=1)
plt.xlabel('Bounding Box Width [pixels]')
plt.ylabel('Bounding Box Height [pixels]')
plt.xlim(0,500)
plt.ylim(0,500)

# Save plot
if USE_ALL_OBJECTS:
    plt.savefig('all_classes_bb_sizes_%s.png' % (DATASET))
else:
    plt.savefig('cars_bb_sizes_%s.png' % (DATASET))


plt.show()
```

B.4 Determine and Save Aspect Ratios

```
'''
Save bounding box aspect ratios in dataset in csv file
Master thesis project Cybercom
Roger Kalliomaki
'''

from lxml.etree import parse
```

```
 8  from os import listdir
 9  from os.path import isfile, join
10  import random
11
12  # Set dataset, nr of images and the boolean USE_ALL_OBJECTS which defines if
13  # all objects should be used or only cars.
14  DATASET = 'BDD100K'
15  USE_MAX_NR_IMAGES = True
16  MAX_NR_IMAGES = 5000
17  USE_ALL_OBJECTS = True
18
19  # Set correct annotations path depending on if complete dataset should be used.
20  if USE_ALL_OBJECTS:
21      ANNOTATIONS_PATH = './Annotations_all_classes/'
22  elif DATASET=='BDD100K':
23      ANNOTATIONS_PATH = './Annotations_cars/'
24  else:
25      ANNOTATIONS_PATH = './Annotations/'
26
27  # Read xml files and shuffle them.
28  xml_files = [f for f in listdir(ANNOTATIONS_PATH) if /
        isfile(join(ANNOTATIONS_PATH, f))]
29  random.shuffle(xml_files)
30  # Set nr of files to use
31  nr_files = len(xml_files)
32  if USE_MAX_NR_IMAGES:
33      nr_files = MAX_NR_IMAGES
34
35  # Go throug files and save aspect ratios in a csv file.
36  if USE_ALL_OBJECTS:
37      result_file = open('./'+DATASET+'_all_classes_bb_aspect_ratios.csv', 'w')
38  else:
39      result_file = open('./'+DATASET+'_cars_bb_aspect_ratios.csv', 'w')
40  counter = 0
41  for xml_file_name in xml_files:
42      xmlTree = parse(ANNOTATIONS_PATH + xml_file_name)
43      for element in xmlTree.findall('object'):
44          if USE_ALL_OBJECTS or element.find('name').text == 'car' or /
                element.find('name').text == 'Car':
45              xmin = int(element.find('bndbox').find('xmin').text)
46              xmax = int(element.find('bndbox').find('xmax').text)
47              ymin = int(element.find('bndbox').find('ymin').text)
48              ymax = int(element.find('bndbox').find('ymax').text)
49              x_asp = float(xmax-xmin)
50              y_asp = float(ymax-ymin)
51              if x_asp<1:
52                  x_asp=1
53              if y_asp<1:
54                  y_asp=1
55              aspect_ratio = x_asp/y_asp
56              csv_string = '%f\n' % (aspect_ratio)
57              result_file.write(csv_string)
58      if USE_MAX_NR_IMAGES and counter == MAX_NR_IMAGES:
59          break
60      counter += 1
61      print('%d/%d' % (counter,nr_files))
```

B.5 Plot Aspect Ratios

```
'''
Plot bounding box aspect ratios saved in given csv file
Master thesis project Cybercom
Roger Kalliomaki
'''

import matplotlib.pyplot as plt
import numpy as np

# Set bin factor (nr of samples in each bar), dataset and
# the boolean USE_ALL_OBJECTS which defines if all objects
# should be used or only cars.
BIN_FACTOR = 5.0
DATASET = 'BDD100K'
USE_ALL_OBJECTS = True
NUM_BINS = 350

# Load csv file into numpy array
if USE_ALL_OBJECTS:
    FILE_PATH = './' + DATASET + '_all_classes_bb_aspect_ratios.csv'
else:
    FILE_PATH = './' + DATASET + '_cars_bb_aspect_ratios.csv'
aspect_ratios = np.genfromtxt(FILE_PATH, dtype=float)

# Plot histogram plot
fig, ax = plt.subplots()
n, bins, patches = ax.hist(aspect_ratios, NUM_BINS, range=(0,3.5))
ax.set_xlabel('Bounding Box Aspect Ratio')
ax.set_ylabel('Object Count')

# Save plot
if USE_ALL_OBJECTS:
    plt.savefig('all_classes_aspect_ratios_%s_num_bins_%d.png' % /
        (DATASET,NUM_BINS))
else:
    plt.savefig('cars_aspect_ratios_%s_num_bins_%d.png' % /
        (DATASET,NUM_BINS))

plt.show()
```

B.6 Determine and Save Object Centers

```
'''
Save bounding box centers in dataset in csv file
Master thesis project Cybercom
Roger Kalliomaki
'''

from lxml.etree import parse
from os import listdir
from os.path import isfile, join
import random

```

```
12  # Set dataset, nr of images and the boolean USE_ALL_OBJECTS which defines if
13  # all objects should be used or only cars.
14  DATASET = 'BDD100K'
15  USE_MAX_NR_IMAGES = True
16  MAX_NR_IMAGES = 5000
17  USE_ALL_OBJECTS = True
18
19  # Set correct annotations path depending on if complete dataset should be used.
20  if USE_ALL_OBJECTS:
21      ANNOTATIONS_PATH = './Annotations_all_classes/'
22  elif DATASET=='BDD100K':
23      ANNOTATIONS_PATH = './Annotations_cars/'
24  else:
25      ANNOTATIONS_PATH = './Annotations/'
26
27  # Read xml files and shuffle them.
28  xml_files = [f for f in listdir(ANNOTATIONS_PATH) if /
        isfile(join(ANNOTATIONS_PATH, f))]
29  random.shuffle(xml_files)
30  # Set nr of files to use
31  nr_files = len(xml_files)
32  if USE_MAX_NR_IMAGES:
33      nr_files = MAX_NR_IMAGES
34
35  # Go throug files and save aspect ratios in a csv file.
36  if USE_ALL_OBJECTS:
37      result_file = open('./'+DATASET+'_all_classes_bb_centers.csv', 'w')
38  else:
39      result_file = open('./'+DATASET+'_cars_bb_centers.csv', 'w')
40  result_file.write('x_center,y_center' + '\n')
41  counter = 0
42  for xml_file_name in xml_files:
43      xmlTree = parse(ANNOTATIONS_PATH + xml_file_name)
44      for element in xmlTree.findall('object'):
45          if USE_ALL_OBJECTS or element.find('name').text == 'car':
46              xmin = int(element.find('bndbox').find('xmin').text)
47              xmax = int(element.find('bndbox').find('xmax').text)
48              ymin = int(element.find('bndbox').find('ymin').text)
49              ymax = int(element.find('bndbox').find('ymax').text)
50              bb_x_center = int(xmin + (xmax-xmin)/2)
51              bb_y_center = int(ymin + (ymax-ymin)/2)
52              csv_string = '%d,%d\n' % (bb_x_center,bb_y_center)
53              result_file.write(csv_string)
54      if USE_MAX_NR_IMAGES and counter == MAX_NR_IMAGES:
55          break
56      counter += 1
57      print('%d/%d' % (counter,nr_files))
```

B.7 Plot Heat Map

```
1  '''
2  Plot bounding box sizes saved in given csv file
3  Master thesis project Cybercom
4  Roger Kalliomaki
5  '''
```

```
6
7  import matplotlib.pyplot as plt
8  import numpy as np
9
10 # Set resolution for dataset
11 #REC = [1242,375] # KITTI 1242x375
12 #REC = [1914,1052] # FCAV = 1914x1052
13 REC = [1280,720] # BDD100K = 1280x720
14
15 # Set bin factor (nr of samples in each heat map point), dataset and
16 # the boolean USE_ALL_OBJECTS which defines if all objects
17 # should be used or only cars.
18 BIN_FACTOR = 5.0
19 DATASET = 'BDD100K'
20 USE_ALL_OBJECTS = True
21
22 # Load csv file into numpy array
23 if USE_ALL_OBJECTS:
24     FILE_PATH = './' + DATASET + '_all_classes_bb_centers.csv'
25 else:
26     FILE_PATH = './' + DATASET + '_cars_bb_centers.csv'
27 bb_centers = np.genfromtxt(FILE_PATH, dtype=int, delimiter=',', skip_header=1)
28
29 # Create numpy matrix representing the image and load values into bins
30 xmax = int(round(REC[0]/BIN_FACTOR))
31 ymax = int(round(REC[1]/BIN_FACTOR))
32 bb_bins = np.zeros((ymax+1, xmax+1)) # numpy matrix = [y,x]
33 for box_center in bb_centers:
34     x_indx = int(box_center[0]/BIN_FACTOR)
35     y_indx = int(box_center[1]/BIN_FACTOR)
36     bb_bins[y_indx,x_indx] = bb_bins[y_indx,x_indx] + 1
37
38 # Normalize values and plot heat map
39 bb_bins_norm = bb_bins/np.size(bb_bins)
40 bin_max_value = bb_bins_norm.max()
41 fig, ax = plt.subplots()
42 im = ax.imshow(bb_bins_norm,interpolation='bicubic')
43 ax.axes.get_xaxis().set_visible(False)
44 ax.axes.get_yaxis().set_visible(False)
45 cbar = ax.figure.colorbar(im, orientation='horizontal', shrink=1.0, ticks=[0, /
       bin_max_value/2, bin_max_value], pad=0.1)
46 cbar.ax.set_xticklabels(['Low', 'Medium', 'High'])
47 if USE_ALL_OBJECTS:
48     cbar.ax.set_xlabel('Number␣of␣object␣instances')
49     # Save plot
50     plt.savefig('all_classes_heatmap_%s_binfactor_%s.png' % /
           (DATASET,int(BIN_FACTOR)))
51
52 else:
53     cbar.ax.set_xlabel('Number␣of␣car␣instances')
54     # Save plot
55     plt.savefig('cars_heatmap_%s_binfactor_%d.png' % /
           (DATASET,int(BIN_FACTOR)))
56
57 plt.show()
```

B.8 Determine and Save Mean RGB Values

```
'''
Save dataset mean RGB values in csv file
Master thesis project Cybercom
Roger Kalliomaki
'''

import os
import numpy
from PIL import Image
import random

# Set dataset, nr of images and image folder path
DATASET = 'BDD100K'
USE_MAX_NR_IMAGES = True
MAX_NR_IMAGES = 1000
DATASET_PATH = './JPEGImages/'

# Function that returns mean rgb value of an image located in argument path
def get_mean_rgb_image(image_path):
    image = Image.open(image_path, 'r')
    width, height = image.size
    pixel_values = list(image.getdata())
    channels = 0
    if image.mode == 'RGB':
        channels = 3
    elif image.mode == 'RGBA':
        channels = 4
    elif image.mode == 'L':
        channels = 1
    else:
        print('Unknown mode: %s' % image.mode)
        return None
    pixel_values = numpy.array(pixel_values).reshape((width, height, channels))
    if channels == 4:
        pixel_values = pixel_values[:,:,:3]
    return numpy.mean(pixel_values, axis=(0, 1)).astype(int)

# Function that goes throug all images in argument path and calls the function
# get_mean_rgb_image for all images in path. Saves mean rgb values for all
# images in csv file
def save_to_file(dataset_path):
    result_file = open('./'+DATASET+'_rgb_values.csv', 'w')
    result_file.write('R,G,B' + '\n')
    path, dirs, files = next(os.walk(dataset_path))
    random.shuffle(files)
    file_count = len(files)
    file_counter = 1
    for filename in files:
        if USE_MAX_NR_IMAGES:
            print('Processing file: %s/%s' % (file_counter,MAX_NR_IMAGES))
        else:
            print('Processing file: %s/%s' % (file_counter,file_count))
        mean_rgb = get_mean_rgb_image(dataset_path + filename)
        csv_string = '%d,%d,%d\n' % (mean_rgb[0],mean_rgb[1],mean_rgb[2])
        result_file.write(csv_string)
```

```
56              file_counter += 1
57              if USE_MAX_NR_IMAGES and MAX_NR_IMAGES == file_counter:
58                  break
59          if file_count == 0:
60              print('No␣files␣in␣path␣%s' % dataset_path)
61
62  if __name__ == "__main__":
63      save_to_file(DATASET_PATH)
```

B.9 Plot Mean RGB Values

```
1   '''
2   Plot 3D scatter plot of mean RGB values saved in given csv file
3   Master thesis project Cybercom
4   Roger Kalliomaki
5   '''
6
7   # This import registers the 3D projection, but is otherwise unused.
8   from mpl_toolkits.mplot3d import Axes3D # noqa: F401 unused import
9
10  import matplotlib.pyplot as plt
11  import numpy as np
12  import csv
13
14  # Set dataset and csv file path
15  FILE_PATH = './rgb_values.csv'
16  DATASET = 'BDD100K'
17
18  # Go through csv file and fill in rgb values in the 3D scatter plot.
19  fig = plt.figure()
20  ax = fig.add_subplot(111, projection='3d')
21  with open(FILE_PATH) as csv_file:
22      csv_reader = csv.reader(csv_file, delimiter=',')
23      line_count = 0
24      for row in csv_reader:
25          if line_count == 0:
26              line_count += 1
27          else:
28              xs = int(row[0])
29              ys = int(row[1])
30              zs = int(row[2])
31              if line_count == 1:
32                  ax.scatter(xs, ys, zs, c='red', s=1, label=DATASET)
33              else:
34                  ax.scatter(xs, ys, zs, c='red', s=1)
35              line_count += 1
36  ax.set_xlabel('R')
37  ax.set_ylabel('G')
38  ax.set_zlabel('B')
39  ax.set_xlim(-50, 300)
40  ax.set_ylim(-50, 300)
41  ax.set_zlim(-50, 300)
42  plt.legend(loc=2)
43  plt.savefig('3Dplot2.png')
44  plt.show()
```

B.10 Split Dataset

```
'''
Split between training and validation data
(can also be used to split training and test data)
Master thesis project Cybercom
Roger Kalliomaki
'''

import numpy as np

NR_OF_VAL_IMAGES = 5000

# Imageset paths
all_images_path = "./ImageSets/Main/train_val.txt"
train_images_file = open("./ImageSets/Main/train.txt", "w")
val_images_file = open("./ImageSets/Main/val.txt", "w")

# Read lines from file
with open(all_images_path) as f:
    content = f.readlines()

# Remove whitespace characters and create numpy array
#lines = np.array([x.strip() for x in content])

# Create numpy array
lines = np.array(content)

# Shuffle lines
np.random.shuffle(lines)

# Split in train val
train, val = lines[NR_OF_VAL_IMAGES:], lines[:NR_OF_VAL_IMAGES]

# Write to files
for line in train:
    train_images_file.write(line)
for line in val:
    val_images_file.write(line)

# Close files
train_images_file.close()
val_images_file.close()
```

B.11 Plot Training History

```
'''
Plot training history
Master thesis project Cybercom
Roger Kalliomaki
'''

from matplotlib import pyplot as plt
import numpy as np

```

```
#load history
history = np.genfromtxt('training_log.csv', delimiter=',')

# Plot iterations or epochs
USE_ITERATIONS = True
ITS_PER_EPOCH = 1000
iterations = []
if USE_ITERATIONS:
    for i in range(0, len(history)):
        iterations.append(history[i][0]*ITS_PER_EPOCH)

# Load loss and validation
loss = []
val = []
for i in range(0, len(history)):
    loss.append(history[i][1])
    val.append(history[i][2])

#plot loss curves
plt.figure(figsize=[8,6])
if USE_ITERATIONS:
    plt.plot(iterations,loss,'r',linewidth=3.0)
    plt.plot(iterations,val,'b',linewidth=3.0)
    plt.xlabel('Iterations ',fontsize=16)
else:
    plt.plot(loss,'r',linewidth=3.0)
    plt.plot(val,'b',linewidth=3.0)
    plt.xlabel('Epochs ',fontsize=16)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.ylabel('Loss',fontsize=16)
plt.title('Model loss',fontsize=16)
plt.savefig('train_im.png')
plt.show()
```

B.12 Annotate Video

```
'''
Annotate video using SSD predictions
Master thesis project Cybercom
Roger Kalliomaki
'''

import cv2
import numpy as np
from models.keras_ssd300 import ssd_300
from keras import backend as K
from keras.optimizers import Adam
from keras_loss_function.keras_ssd_loss import SSDLoss

# Set file paths, image size, nr of classes, if to use png files, model mode,
# and path to saved model file
IN_FILE_PATH = '<PATH TO ORIGINAL VIDEO>'
OUT_FILE_PATH = '<PATH TO NEW ANNOTATED VIDEO>'
IMG_HEIGHT = 300
IMG_WIDTH = 300
```

```
20 NR_CLASSES = 10
21 USE_PNG = False
22 MODEL_MODE = 'inference'
23 SAVED_MODEL_PATH = '<SAVED␣MODEL␣PATH>'
24
25 SHORTEN = False
26 START_SECOND = 3
27 NR_SECONDS = 57
28
29
30 # The per-channel mean of the images in the dataset
31 #VOC 2007 [113, 107, 99]
32 #VOC 2012 [114, 109, 101]
33 #COCO [123, 117, 104]
34 #BBD100K [70, 74, 74]
35 #FCAV [107, 104, 99]
36 MEAN_COLOR = [70, 74, 74]
37 SCALES = [0.05, 0.1, 0.29, 0.48, 0.67, 0.86, 1.05]
38
39 # Aspect ratios the network was trained with
40 ASPECT_RATIOS = [[1.0, 1.5, 0.5],
41                  [1.0, 1.5, 0.5, 2.0, 2.5],
42                  [1.0, 1.5, 0.5, 2.0, 2.5],
43                  [1.0, 1.5, 0.5, 2.0, 2.5],
44                  [1.0, 1.5, 0.5],
45                  [1.0, 1.5, 0.5]]
46
47 # Set classes
48 classes = ['background', 'bus', 'traffic␣light', 'traffic␣sign', 'person', 'bike', 'truck', /
       'motor', 'car', 'train', 'rider']
49
50 # Build new model
51 # Clear previous models from memory
52 K.clear_session()
53
54 # Set parameters
55 model = ssd_300(image_size=(IMG_HEIGHT, IMG_WIDTH, 3),
56                 n_classes=NR_CLASSES,
57                 mode=MODEL_MODE,
58                 l2_regularization=0.0005,
59                 scales=SCALES,
60                 aspect_ratios_per_layer=ASPECT_RATIOS,
61                 two_boxes_for_ar1=True,
62                 steps=[8, 16, 32, 64, 100, 300],
63                 offsets=[0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
64                 clip_boxes=False,
65                 variances=[0.1, 0.1, 0.2, 0.2],
66                 normalize_coords=True,
67                 subtract_mean=MEAN_COLOR,
68                 swap_channels=[2, 1, 0],
69                 confidence_thresh=0.5,
70                 iou_threshold=0.45,
71                 top_k=200,
72                 nms_max_output_size=400)
73
74 # Load the trained weights into the model
75 weights_path = SAVED_MODEL_PATH
```

```
76  model.load_weights(weights_path, by_name=True)
77
78  # Compile the model so that Keras won't complain the next time you load it
79  optmzr = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
80  ssd_loss = SSDLoss(neg_pos_ratio=3, alpha=1.0)
81  model.compile(optimizer=optmzr, loss=ssd_loss.compute_loss)
82
83  # Define colors for each possible object
84  colors = {'aeroplane' : (255,0,0), 'bicycle' : (0,255,0), 'bird' : (0,0,255), 'boat' : /
        (255,255,0), 'bottle' : (0,255,255), 'bus' : (255,0,255), 'car' : (0,255,127), 'cat' : /
        (128,128,0), 'chair' : (0,128,0), 'cow' : (128,0,128), 'diningtable' : (0,0,128), 'dog' : /
        (178,34,34), 'horse' : (220,20,60), 'motorbike' : (205,92,92), 'person' : (255,165,0), /
        'pottedplant' : (128,0,0) , 'sheep' : (0,139,139), 'sofa' : (138,43,226), 'train' : /
        (147,112,219), 'tvmonitor' : (219,112,147), 'traffic␣light' : (112,219,147), 'traffic␣/
        sign' : (138,226,43), 'bike' : (0,255,0), 'truck' : (0,128,0), 'motor' : (205,92,92), /
        'rider' : (230,150,0), 'Car' : (128,0,0)}
85
86  # Settings for relative sizes depending on image size and loading first frame
87  font_scale = 1.5
88  font = cv2.FONT_HERSHEY_PLAIN
89  cap = cv2.VideoCapture(IN_FILE_PATH)
90  fps = int(cap.get(cv2.CAP_PROP_FPS))
91  nr_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
92  nr_seconds = nr_frames/fps
93  fourcc = cv2.VideoWriter_fourcc(*'MJPG')
94  ret, frame = cap.read()
95  orig_width = frame.shape[1]
96  orig_height = frame.shape[0]
97  width_scale = float(13*orig_width)/1280.0
98  height_scale = float(orig_height)/720.0
99  text_box_width = 0
100 text_box_height = int(25.0*height_scale)
101 text_pos_height = int(20.0*height_scale)
102 font_scale = font_scale*height_scale
103 box_line_width = int(3.0*height_scale)
104 out_video = cv2.VideoWriter(OUT_FILE_PATH, fourcc, fps, (orig_width,orig_height))
105 frame_counter = 0
106 seconds = 0
107 # Read video frame by frame
108 while(True):
109     # If not to use complete video
110     if SHORTEN and seconds < START_SECOND:
111         ret, frame = cap.read()
112         frame_counter += 1
113         if frame_counter == fps:
114             seconds += 1
115             print('Video␣second:␣%d/%d␣(skipped)' % (seconds,nr_seconds))
116             frame_counter = 0
117         continue
118
119     if ret == True:
120         # Read image frame
121         frame = cv2.flip(frame,0)
122         frame = cv2.flip(frame,1)
123         img = cv2.resize(frame, dsize=(IMG_WIDTH, IMG_HEIGHT), /
                interpolation=cv2.INTER_CUBIC)
124         input_images = np.array([img])
```

```
125             frame_counter += 1
126
127             # Make predictions
128             y_pred = model.predict(input_images)
129             confidence_threshold = 0.6
130             y_pred_thresh = [y_pred[k][y_pred[k,:,1] > confidence_threshold] for k in /
                    range(y_pred.shape[0])]
131
132             for box in y_pred_thresh[0]:
133                 # Transform the predicted bounding boxes for the 300x300 image
134                 # to the original image dimensions.
135                 xmin = int(box[2] * orig_width / IMG_WIDTH)
136                 ymin = int(box[3] * orig_height / IMG_HEIGHT)
137                 xmax = int(box[4] * orig_width / IMG_WIDTH)
138                 ymax = int(box[5] * orig_height / IMG_HEIGHT)
139                 # Draw it on image
140                 label = '{}: {:.2f}'.format(classes[int(box[0])], box[1])
141                 class_label = '{}'.format(classes[int(box[0])])
142                 text_box_width = int(float(len(label))*width_scale)
143                 color = colors[class_label]
144                 cv2.rectangle(frame, (xmin,ymin),(xmax,ymax), color, box_line_width)
145                 cv2.rectangle(frame, /
                        (xmin,ymin-text_box_height),(xmin+text_box_width, ymin), /
                        color, box_line_width)
146                 cv2.rectangle(frame, (xmin, /
                        ymin-text_box_height),(xmin+text_box_width, ymin), color,-1)
147                 cv2.putText(frame, label, (xmin, /
                        ymin+text_pos_height-text_box_height), font, /
                        fontScale=font_scale, color=(0, 0, 0), thickness=int(height_scale))
148
149             # Write annotated frame to video
150             out_video.write(frame)
151             # Keep track of seconds
152             if frame_counter == fps:
153                 seconds += 1
154                 print('Video second: %d/%d' % (seconds,nr_seconds))
155                 frame_counter = 0
156
157         else:
158             break
159         if SHORTEN and seconds == NR_SECONDS:
160             print('Stopped at second: %d/%d' % (seconds,nr_seconds))
161             break
162         ret, frame = cap.read()
163
164 cap.release()
165 out_video.release()
```