FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Video Object Recognition based on Deep Learning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Stefan Stojanoski, Stefan Stojanoski
Matrikelnummer 1529445

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Wien, 7. März 2019

Stefan Stojanoski	Horst Eidenberger

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Video Object Recognition based on Deep Learning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Stefan Stojanoski, Stefan Stojanoski
Registration Number 1529445

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Vienna, 7th March, 2019

Stefan Stojanoski Horst Eidenberger

Erklärung zur Verfassung der Arbeit

Stefan Stojanoski, Stefan Stojanoski
Address

Wien, 7. März 2019

Stefan Stojanoski

Danksagung

Ich würde gerne meinem Betreuer Prof. Horst Eidenberger für sein großartige Unterstützung und Beratung während des Entstehungsprozesses meiner Masterarbeit danken. Außerdem danke ich der technischen Universität Wien (TU) für das Stipendium und die zur Verfügung gestellten technischen Mittel. Ich bin stolz darauf nie aufgegeben zu haben und möchte mich daher für die Motivation und Unterstützung während des gesamten Studiums bei meiner Familie und Deana Zafirova bedanken.

Acknowledgements

I would like to thank my supervisor Prof. Horst Eidenberger for his exquisite mentorship and advisory during the process of the thesis. I would also like to thank the Technical University of Vienna (TU) for the scholarship they provided me with for the necessary hardware resources. I'm proud that I never gave up and for that I want to thank my family and Deana Zafirova for the motivation and support during the whole studies at TU.

Kurzfassung

In dieser Masterarbeit haben wir eine Client-Server-Architektur zur automatischen Erkennung von Reklametafeln in Videostreams entwickelt. Die Client-Seite wurde mit einer Android-Anwendung umgesetzt, die dem Zweck dient Videodaten für die Server-Seite zu sammeln. Für die Server-Seite hingegen wurde *StefanNet* entwickelt, wobei es sich um ein Deep Neural Network handelt. *StefanNet* kann Reklametafeln in einem Video Frame erkennen und klassifizieren. *StefanNet* hat einen Feature Extractor mit 23 konvolutionären Ebenen und benutzt einen Single Shot Detector (SSD) zur Objekterkennung. Das Netz wurde mit dem selbst erstellten *BillboardDataset* trainiert, welches 4042 Beispielbilder beinhaltet, die von Reklametafeln in den U-Bahn-Stationen Wiens gemacht wurden. Zusätzlich wurden Data-Augmentation-Techniken angewendet um den Datensatz künstlich um 25% zu vergrößern. Außerdem wurden Quantisierungstechniken auf *StefanNet* angewendet um die Bittiefe, die notwendig ist, um die Gewichte des Netzwerks zu speichern, von float32 auf float16 zu verringern. Wir haben die Performance von *StefanNet* evaluiert, indem wir es mit den state of the art Netzwerken ResNet, MobileNet, Inception und VGG16 verglichen haben. Der Validierungsdatensatz setzt sich zusammen aus Ansichten der Reklametafeln von vorne und von der Seite. *StefanNet* erreichte 91% mean average precision (mAp) auf dem Testdatensatz, 98% mAp für Ansichten von vorne und 82% mAp für Ansichten von der Seite. Die Inferenzgeschwindigkeit war 40 Bilder pro Sekunde (FPS) auf einer Nvidia 1080 Grafikkarte. Die quantisierte Version von *StefanNet* erreichte 91% mAp auf dem Testdatensatz, 96% mAp für Ansichten von vorne und 85% mAp für Ansichten von der Seite. Die Inferenzgeschwindigkeit für die quantisierte Version war 45 FPS. Sowohl *StefanNet* als auch dessen quantisierte Version hat eine höhere mAp als die anderen evaluierten Netzwerke erreicht. Das bestätigt, dass die Architektur von *StefanNet* die derzeit am Besten passende Architektur für das Problem der automatischen Reklametafel-Erkennung in einem Video Stream ist.

Abstract

In this master thesis we designed a client server system for automatic billboard recognition in video streams. The client side is represented by an Android application which serves the purpose of collecting various video data streams for the server side. For the server side a deep neural network, called *StefanNet*, was designed. *StefanNet* is a fully convolutional neural network which is able to properly classify and localize billboard objects within a video frame. *StefanNet* has a feature extractor which contains 23 convolutional layers and uses a single shot detector (SSD) as an object detector. *StefanNet* has been trained on the self-designed *BillboardDataset* which contains 4042 image samples taken from the billboards located throughout the metro stations in Vienna. Additionally, data augmentation techniques have been implemented to artificially augment the dataset with a 25% increase rate. Furthermore, the compression-based quantization technique has been applied to the *StefanNet* model to reduce the bit-width necessary for storing the weights of the network from float32 to float16. We evaluated the performance of *StefanNet* by comparing against the state-of-the-art networks ResNet, MobileNet, Inception and VGG16. The validation dataset contains both side and frontal views of the billboards. *StefanNet* achieved 91% mean average precision (mAp) on the test dataset, 98% mAp on the frontal view validation dataset and 82% mAp on the side view validation dataset. The inference rate was 40 FPS on a Nvidia 1080 graphics card. The quantized version of the *StefanNet* model achieved 91% mAp on the test dataset, 96% mAp on the frontal view validation dataset and 85% mAp on the side view validation at an inference rate of 45 FPS. In comparison to the other evaluated networks both the *StefanNet* model and the quantized version of the model produce superior results and outperform the benchmark network models on all datasets. This confirms that the architecture of *StefanNet* is currently the most suitable for the specific problem of automatic billboard detection in video streams.

Contents

CHAPTER 1

Introduction

1.1 Motivation

For humans, visual perception is the sense of most vital importance. Unlike other animals, that mostly rely on the senses of smell and sound, humans rely heavily on the visual system in every interaction with the environment. Everyday tasks like moving around and picking up objects, recognizing a face, driving and reading would not be so easy without our highly-developed system for visual perception. Essentially, all of these simple tasks come down to object detection, localization and recognition. To pick up an object, humans first determine which part of the visual impression corresponds to the desired object. To recognize a person, humans first locate the person, then the face within the image and then perform detection.

The human visual system is special, as it has the ability to maintain a stable and constant perception of the ever-so changing environment. Formally known as perceptual invariance [LRM04], this ability is obtained from the complex hierarchical connections in the human brain. The brain can perform object recognition under a number of challenging conditions. For instance, person recognition still runs smoothly even if the person we want to recognize has gained weight, is far away, is illuminated by the sun or in the shadow. This is only feasible because the human brain learns abstractions of visual impressions and representations. In turn, these abstractions are invariant to size, illumination, contrast, rotation, orientation or any combination of them. This makes the human visual system the ultimate pattern recognition system, unparalleled by any so far existing, computer vision system.

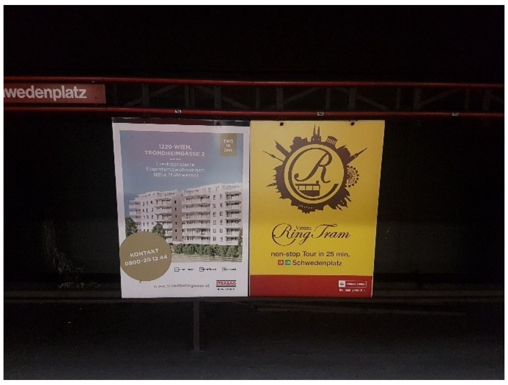
The processing steps required for object recognition are performed so fast by the brain that we as humans do not even notice or think about them actively. However, what might seem so trivially effortless for us, still represents one of the greatest challenges that computers and the field of computer vision have faced. Currently, the state-of-the-art solutions are able to reproduce only a small subset of the various tasks the brain performs when it comes to image processing and interpretation. The approaches are generally complex and even mimicking a small fraction of the human perception abilities requires a combination of several techniques and algorithms. Furthermore, the current solutions require a massive amount of computational resources, data, storage and runtime. Running these combined approaches in real time, though proven to be possible, is still not optimal and easy, and thus poses a big challenge.

Despite these setbacks, the field of object recognition, at the moment has a wide domain of applications. Some examples include video surveillance, autonomous vehicles, person and face detection. Regardless of the earlier efforts, it was only in the past few years that science had a glimpse of how powerful deep convolutional neural networks can be in supporting complex tasks such as object recognition. With the advances in deep learning, object recognition has the potential to extend its applicability and usability to a number of other versatile fields. Hence, it is of crucial importance to proceed with the research and development of the field of deep learning in order to help the area of object recognition evolve and grow. This work endeavors to bring us one step closer towards that goal of designing and training neural network models for performing object recognition in the field of media. In my diploma thesis I explore various techniques for preventing and avoiding the challenges neural networks face when dealing with the task of media (billboard) recognition.

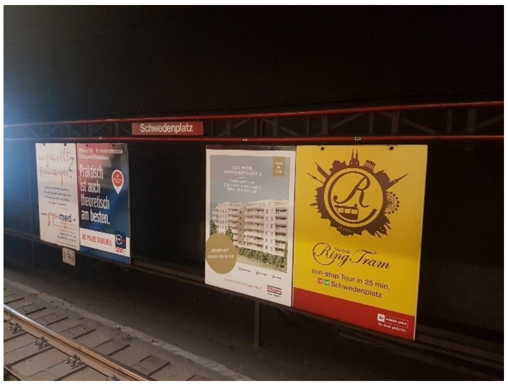
1.2 Problem Statement

This master thesis aims to address the problem of object recognition in video data streams. In particular, the main focus of this work lies in detecting and localizing billboard objects with the help of convolutional neural networks in pre-recorded videos. Furthermore, this master thesis deals with the challenges that convolutional neural networks presently face, by exploring and analysing the effect of the application of several efficiency and performance improving techniques. Namely, methods for reducing the amount of storage, increasing the network inference rate, as well as faster training of neural networks have been applied and evaluated.

Convolutional neural networks are the reason for the recent breakthroughs and successful advances in the research area of computer vision (see [KSH12a], [SEZ$^+$]). Currently, convolutional neural networks represent the state-of-the-art method for dealing with the problem of object recognition [KOLW16]. This is partly due to the hardware advances and the promotion of the utilization of Graphic Processing Units (GPUs). The powerful and at the same time parallelizable GPUs have enabled the application of convolutional neural networks to large datasets (see [DDS$^+$09] [EEG$^+$15a] [LMB$^+$14]) and with that achieve outstanding performance (see [SZ14b] [HZRS15] [SLJ$^+$14]). These accomplishments have helped convolutional neural networks establish themselves as the correct method for dealing with computer vision problems and worthy of further research and improvement. Hence, the main task of this thesis is to propose a design and



(a) Front-on view



(b) Side view

Figure 1.1: Samples of *BillboardDataset*

implement a deep convolutional neural network as a solution to the stated problem.

However, to design and implement an efficient deep learning algorithm for the recognition of billboard objects is a highly complex task. For this reason, the following two aspects must be taken into consideration: the training image dataset and the structure of the convolutional neural network.

Firstly, the training dataset can have a large impact on the efficiency and the overall performance of the object recognition algorithm. Thus, it is of crucial importance that the gathered data for the training set is diverse and of correct proportion (in terms of the number of images provisioned). Moreover, the gathering of data and provision of training samples is also part of this thesis. Since the number of billboard image datasets online is very small, a *BillboardDataset* image dataset has been specifically created for this thesis. The *BillboardDataset* contains 4.042 images captured from all of the metro lines in Vienna, Austria. Diversity of the images is ensured in a way that the dataset contains images taken with a front-on view as well as images with a profile view (or side-view) of the billboard. Samples of the *BillboardDataset* are provided in Figure 1.1a and Figure 1.1b. Additionally, to make up for the lack of data, techniques for artificial augmentation of the image dataset have been applied with an increase rate of 25%.

Secondly, it has been shown that there exists a correlation between the depth of the convolutional network and the accuracy it achieves with respect to object recognition (see [SLJ$^+$15], [SZ14a]). Additionally, recent studies state that the incorporation of recurrent connections into convolutional neural networks can help to improve the object recognition performance (see [LH15], [BAA16], [AHYT17]). This thesis works towards exploring the architecture and the depth of convolutional neural networks in order to provide a convolutional network design that suits the particular problem best. For this reason, a design of a network that, given the current problem, will improve on the performance of the current best-performing state-of-the-art convolutional networks is the main task of this thesis. The evaluation of the performance will be provided with respect to several criteria, including the inference rate and accuracy.

Lastly, in order to tackle the current challenges convolutional networks face, compression-based method like quantization [Kri18] will be executed to fully optimize the designed convolutional neural network model. Quantization [Kri18] helps in reducing the size of the model and increasing the speed of both the inference rate and the training of the convolutional neural network. In addition to that, a thorough analysis will be provided in order to evaluate the feasibility of the application of the trained model for billboard object recognition.

1.3 Aim of the work

The aim of this work is to create a client server application, where the client is an Android application and the server is a deep neural network model used for object recognition. The server is designed to provide a model which will serve the purpose of detecting, localizing and properly classifying billboard objects. In other words, this means to create the image dataset *BillboardDataset* and develop a deep neural network model which will be exhaustively trained on the *BillboardDataset*. This model in turn, will be able to clearly distinguish, classify and localize a billboard object (against another non-billboard objects), within the provided video input streams.

The client side, on the other hand, is represented by an Android application which is responsible for collecting the input video data streams. These collected video data streams will be exclusively used for testing the performance of the neural network model on the task of object recognition. The Android application allows sharing of the video data streams to a cloud storage environment for easier access and memory preservation purposes. The application has three main functionalities: record, share and play video. A visual representation of the client server architecture is provided in Figure 1.2.

The *BillboardDataset* required by the server side has been acquired through the metro lines, in indoor as well as outdoor settings and in various lightning conditions over the course of a couple of acquisition days. Thus, the dataset contains fully labeled diverse data. As this thesis has the goal to study the correlation between the structural characteristics and the performance of the network, a new self-designed neural network *StefanNet* is developed and compared to four distinct state-of-the-art deep neural network models. *StefanNet* represents the server side and along with the rest of the models is trained on the image dataset *BillboardDataset*. The results of their output are examined and a thorough comparison of their performance is performed.

Secondary objective of this thesis is to determine whether and to which extent the compression-based technique quantization affects the performance of trained neural network models. This method has been applied to the trained neural network model of *StefanNet*, after which the model has once again been trained for several epochs to retrieve its original accuracy. The influence of this technique has been measured by several criteria and reported respectively. Hence, in the last phase of the practical part

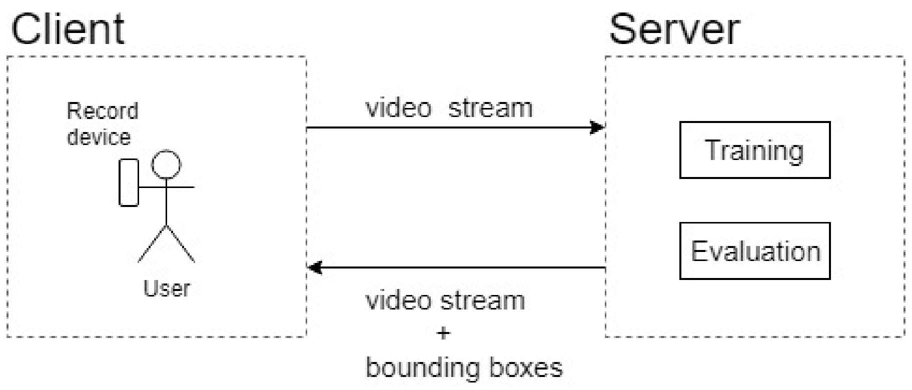


Figure 1.2: Proposed client server architecture

of this thesis, an extensive testing and comparison of the performance of all of the neural network models has been executed.

To summarize, the main goals of this thesis are:

- Development of an Android application used for gathering video data streams and sharing them to a cloud environment (client side).
- Design and implementation of *StefanNet* (server side).
- Training of all convolutional neural network models used for billboard object recognition (server side).
- Evaluation of the performance of the different architectures of the distinct convolutional neural networks.
- Implementation of the compression-based quantization technique on the neural network model of *StefanNet* and evaluation of its effect on the performance.

1.4 Structure of the work

The rest of this work is organized as follows. First a brief introduction of deep learning is provided along with a description of the base functionalities and concepts related to convolutional neural networks. Next, the related work and state-of-the-art achievements in the field of object recognition with focus on media and billboard recognition are discussed and reviewed. Afterwards the details regarding the methodology used in designing the proposed system and the creation of the *BillboardDataset* are explained. In chapter 5 the specifics with respect to hardware, software, frameworks as well as the architecture and design of *StefanNet* are presented. Furthermore, this chapter also provides the training details, the implementation process of the quantization technique as well as the design and specifics of the Android application. The achieved results are evaluated and analyzed in the evaluation chapter. Lastly, the thesis is closed with a summary and a conclusion illuminating both all the achievements accomplished and the future work to come.

CHAPTER 2

Background

2.1 Introduction to Artificial Intelligence

Artificial intelligence [PMG97] is formally defined as the study of devices that have the ability not only to observe the environment, but also to self-define a course of action that will maximize their chance of achieving a specific goal. In 1955 John McCarthy [MRSM06] defined AI as the science and engineering of making intelligent machines that have the ability to achieve goals like humans do. Since both definitions are rather broad and general, it is safe to say that any human intelligence performed by a machine is in fact artificial intelligence. It is believed that it was Alan Turing who started the whole AI era by simply proposing weather machines can think [Tur50]. Since Alan Turing, there was a major focus-shift as well as progress in the field of AI from machine bots that play board games, to smart homes and autonomous vehicles. But what does AI really encompass?

AI is the future of humanity as well as non-humanity i.e machines. The key ingredient an artificial machine needs in order to perform well, is data [LGEC17]. Nowadays with the current evolution of technology, the Internet, social media, cameras and smartphones capture and provide massive amounts of data constantly. In 1959 Arthur Samuel [Sam59] recognized Machine Learning as one of the largest subfields of AI and Tom Mitchell [Mit06] defined it as the study of computer algorithms that improve automatically through experience. Machine Learning [Alp10] heavily relies on working with large datasets and performs evaluations and comparisons of the data in order to obtain a common pattern or to discover variations. Essentially, the practice of machine learning includes learning from large amounts of parsed data in order to make an intelligent prediction [Alp10]. Thus, instead of hard-coding the actions the machine needs to perform, the machine is trained with data, utilizing algorithms which specify the form of learning [Bis07]. As a result, Machine Learning is often thought to be an approach of achieving AI [Alp16]. So one may ask, where does Deep Learning come into play?

Deep Learning [DY14] is a sub-branch of Machine Learning that deals and focuses on the algorithms inspired by the structure and function of the brain called artificial neural networks. Unlike the common belief, the study of deep learning has a long history and dates back to the early 1940s [MP43]. The study itself has gone by many names and the initial perspective of its computational models was to recreate the structure of the human brain [Hay09]. Even though the original models were intended to capture the process of biological learning or in other words to mimic the way of how learning and understanding happens in humans, the models are generally not designed to be a realistic image of the brain function [GBC16]. The reason why Deep Learning has gained so much attention lately is due to one of its greatest advantages and that is its ability to generalize and abstract [KKB18][Zuc03]. In Deep Learning, this is done by representing the world as a nested hierarchy of concepts [GBC16]. Hence, every such concept is defined with respect to simpler, less complex concepts. This is possible as a result of the many successive layers of neurons which increase in complexity for every newly-learned concept. This trait of deep learning makes it very suitable and able to tackle computer vision problems such as image classification and object recognition.

2.2 The Perceptron

In an effort to reproduce the process of human learning in computers, in 1958 Rosenblatt [Ros58] revolutionized the field of machine learning and AI by inventing the perceptron. The perceptron is a linear model intended to perform binary classification and is commonly known as single-layer neural network. This is because its structure contains one artificial neuron which is thought to resemble the neurons in the human brain [Ros58]. While in his original paper Rosenblatt [Ros58] interprets the perceptron with a rather biolgical motivation, we provide the nowadays more common, mathematical definition of the preceptron.

The perceptron [PG17] is defined in terms of a set of inputs x_1, ..., x_n, unique weights w_1, ..., w_n, a bias b which is a constant and an activation function σ as shown by the Equation 2.1.

$$z = \sigma(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

Essentially, the sum of each input multiplied with its associated weight, also called the **dot product**, is sent to the activation function σ with a defined threshold θ. The activation function σ then takes the dot product and produces a single output (0 or 1) with respect to the defined threshold θ. While there exist many activation functions which will be discussed later on in this chapter, the perceptron utilizes the Heavyside step function which depending on the input, outputs a binary value (0 or 1).[PG17] This process and architecture of the perceptron are depicted in Figure 2.1.

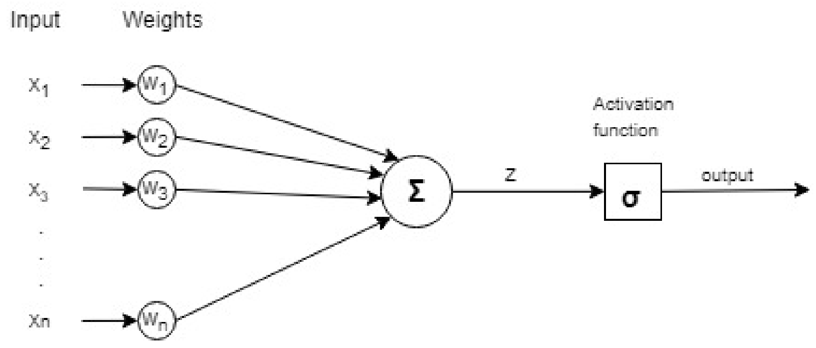


Figure 2.1: Perceptron architecture

The perceptron was one of the first algorithms that could self-learn and adjust the weights [Ros58]. At the beginning of the training, the weight vector is typically initialized to a small value ranging from -1 to 1. During training, the perceptron learning algorithm takes each input sample and computes the output classification. If the classification is correct no changes are necessary, whereas if the classification is incorrect the weights are updated accordingly. This iterative approach is repeated until all training samples are correctly classified.

For its time, the perceptron learning algorithm was pretty impressive. However, a major drawback discovered by Minsky [MP69] is the fact that the perceptron could not model the simple exclusive (XOR) function. Minsky [MP69] went even further and proved that it was impossible for the perceptron to ever learn the XOR function. This may come as no surprise, since the perceptron is a linear model while the XOR function is non-linear (see Figure 2.2). What this means is, that as linear model, the representational power of the perceptron is limited. As a result, the perceptron can only work with linearly separable data i.e. data for which, values of a hyperplane can be found that can be cleanly divided into two classes by a straight line [MP69].

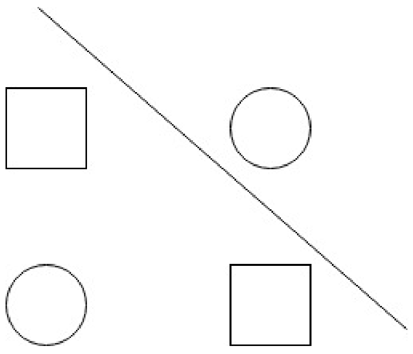


Figure 2.2: XOR function is non-linear

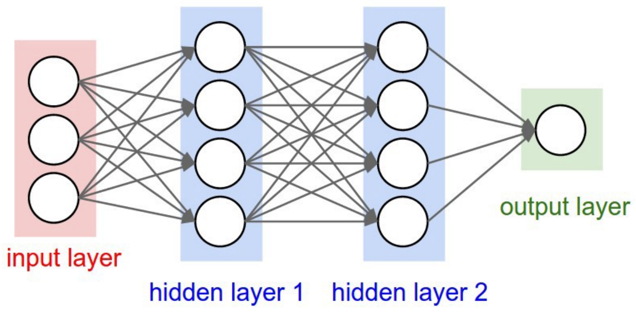


Figure 2.3: Neural network architecture [Kar16]

2.3 Neural Networks

The drawbacks of the perceptron discovered by Minsky [MP69], prompted the development of the neural networks. The simplest form of a neural network or artificial neural network is the feedforward neural network or the Multi-Layer Perceptron (MLP) [RM86] [Hin87]. The structure typically consists of multiple perceptrons (referred to as neurons), organized in several layers. The layers of neurons are connected in a form of a directed acyclic graph. Consequently, the networks are primarily known as feedforward, meaning information only flows forward through the network layers until the final output is generated. Hence, there is no feedback mechanism integrated so that the network could feed its output back in itself. In addition to the several internal (hidden) layers neural networks traditionally include an input layer and an output layer. The complexity and size of the neural network models normally depends on the number of hidden layers the network contains [RM86]. An example of a neural network architecture is provided in Figure 2.3.

Feedforward neural networks are generally represented by chaining together many (usually simple) non-linear functions [RHM86]. For instance, n arbitrary functions f_1, f_2, ..., f_n can be chained to $f(x) = f_{\text{n}}(...f_2(f_1(x)))$ where f_1 represents the first layer of the network, f_2 the second and so forth. The total length of the chained function provides the depth of the network. As witnessed, each layer of the network has an activation function which takes the dot product of input and weights as input. However, in contrast to the perceptron, each layer of artificial neurons used in the neural network can have a different activation function [RHW88]. This allows the neural networks to have a far less restrictive representational power and capabilities. Furthermore, in 1989, Cybenko [Cyb89] proved that the Multi-Layer Perceptron is a universal approximator. This confirms that neural networks can learn to approximate any (including non-linear) continuous function and with that overcome the limitations of the perceptron [Cyb89][RHW88].

The layers (input, hidden, output) of a neural network are traditionally fully-connected [RM86]. That is, from every neuron in the current layer there exists an outgoing connection to every neuron of the next layer. However, the neurons within the same layer do not share a connection. The input fed to the input layer is a raw vector whereas the input fed to the neurons of the other layers is the output of the activation function of the neurons of the previous layer. The output layer is the one that returns the final prediction of the model and typically uses a softmax or a sigmoid activation function [RHM86].

The training of neural networks encompases the learning of the weights, similar to the training of the perceptron [RM86]. However, the training of neural networks is a much more complex process when compared to the one of the perceptron as neural networks include many layers and thus a large number of neurons whose weights must be updated. The training of neural networks at its core, is a form of the classic gradient descent algorithm [RM51][KW52]. The gradient descent algorithm [RM51][KW52] is an optimization algorithm which minimizes the gradient of the loss function. The loss function is an error metric obtained when comparing the output of the network with the desired output [RHW88]. It is defined in terms of the precision the network would lose (hence the name) if the desired output was substituted with the actual output, the current state of the network produces. The gradient of the loss function is in fact the first derivative of the loss function (or the error measure as stated in [RHW88]) and provides the rate at which the loss function changes [RHW88]. To obtain the gradient, the gradient descent algorithm is often paired with the backpropagation algorithm developed by Rumelhart [RM86][RHW88] which successfully computes the gradient. Another important aspect of the gradient descent algorithm is its hyperparameter called *learning rate* which decides the pace at which the weights are updated [RM86]. With this small bit of theoretical background in mind, we now provide a description of the actual training process of neural networks as defined by Rummelhart in [RHW88].

The first step of the training process is to randomly initialize the learning rate and the weights of each of the neurons of the neural network layers. Then a training data sample is forwarded (forward pass) through the network firing the activation functions of every layer until an output is obtained from the output layer. In the next step the loss function, as well as the gradient of the loss function are calculated. With the help of backpropagation the weights of the layer leading to the output layer are updated. The backpropagation is then continued for every subsequent layer of the network. The error is propagated and the weights are updated until the input layer is reached (backward pass). It is very important to note that the weights are always updated in the negative direction of the gradient, as we opt for a loss value closer to zero. This iterative approach is continued until the network has converged or some termination criteria has been reached [RHW88].

2.3.1 Gradient Descent: Variations

Currently, there exist several variations and optimizations of the Gradient Descent algorithm [Rud16]. The variants mainly differ in the number of training samples that need to be processed before performing the backpropagation with the update of the weights [WM00]. In this work we cover Batch Gradient Descent, Mini-Batch Gradient Descent and Stochastic Gradient Descent.

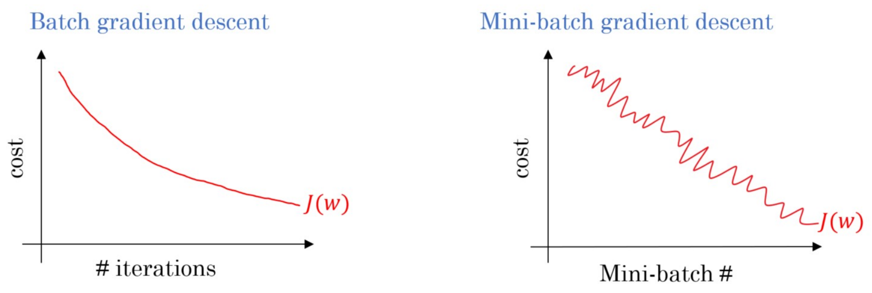


Figure 2.4: Batch gradient descent vs. Mini-batch gradient descent [Dab17]

- **Batch Gradient Descent**

  Batch Gradient Descent is a form of Gradient Descent [WM00] in which all training samples are summed in each iteration when performing the updates to the parameters. Theoretically speaking, this approach is bound to converge either to a global minimum if the loss function is convex or to a local minimum if the loss function is not convex [Rud16]. The main advantage of this approach is that it has a fixed learning rate during the training as well as an unbiased estimate of the gradient. Hence, the greater the number of training samples, the lower the standard error. However, as learning only happens after processing all training samples, this approach may be time-consuming especially when dealing with large datasets [WM00].

- **Mini-Batch Gradient Descent**

  Rather than summing over all training samples like the Batch Gradient Descent, the Mini-Batch Gradient Descent (also known as *Semi-Batch*) sums up over a smaller number of samples based on a provided batch size [WM00]. As a result, the learning happens on every mini-batch of n samples. While this approach is certainly faster than the batch approach, its convergence is not guaranteed [Rud16].

- **Stochastic Gradient Descent**

  The Stochastic Gradient Descent [Rud16] performs updates of the parameters for each training sample. Thus learning occurs on every training instance. As a result it is usually much faster then Batch Gradient Descent. On the other hand, the frequent weight adjustments (after each sample) vastly increase the variance and lead to heavy fluctuation of the objective function [Rud16].

2.3.2 Activation Functions

Activation functions are crucial for the training of neural networks as they are used to propagate the output of the current layer's nodes forward to the next layer until the output layer is reached (forward pass) [RHW88]. Williams [Wil86] elaborates the logic behind activation functions and their importance to feed-forward neural networks. Basically, activation functions are scalar to scalar functions and produce the activation of the neuron [Wil86]. They are of crucial importance to feed-forward neural networks as they are directly connected to the representational power of the neural network [Wil86][RHW88]. Currently, there exists a variety of activation functions [BA18][Ped18]. The most prominent and widely utilized activation functions are thus described below.

- **Linear.** The linear activation function is a straight line function used mostly as an activation function for the input layer. The function is defined as $f(x) = Wx$, where the dependent variable has a direct and proportional relationship with the independent variable [PG17].
- **Sigmoid.** The sigmoid function is a nonlinear function and when depicted on a graph resembles the letter S (see Figure 2.5). The sigmoid function has the ability to reduce extreme values or so called outliers without removing them. An advantage of this function is that its output values are in a range of (0, 1), meaning activations are bound to a certain range. The sigmoid function is smooth and popular when dealing with classification problems [PG17].
- **Rectified Linear Unit (ReLU).** The ReLU activation function is an function that outputs the input x if x is positive and 0 otherwise. The ReLU activation function is nonlinear and the current state-of-the-art, as it has been proven to work well for many computer vision problems. It is less computationally expensive and trains better than the sigmoid function, as it involves only simple mathematical operations [ABMM16].

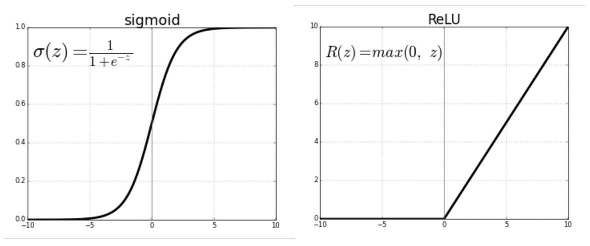


Figure 2.5: Sigmoid (left) and ReLU (right) activation functions [SHA17]

- **Softmax.** The softmax activation function is usually used in the output layer of the neural network. The softmax function is a generalization of the logistic regression and thus can be applied to continuous data and can involve several decision boundaries [PG17].

2.3.3 Loss Functions

The loss function determines how close the current state of the trained neural network is to the ideal, desired state of the same network [RHW86]. Generally, the loss function calculates the error found in the actual predictions of the network for the training data set. As a result, the goal is to minimize the loss obtained from the errors. This way, the loss function helps to redefine the training process of a neural network as an optimization problem [RHW86]. For regression problems the most preferred loss functions are the Mean Squared Error (MSE), Mean Absolute Error and Mean Log Error [RDVC$^+$04]. Loss functions used for the task of classification are Hinge Loss, Logistic Loss and Negative Log Likelihood [RDVC$^+$04][JC17].

2.3.4 Regularization

A central problem in the field of deep learning and machine learning in general is to design and train a model in such a way that it performs well, not only on the training set but also on new and unseen data [SHK$^+$14]. Regularization [GBC16] is a measure performed to help models achieve that goal, be more generalized and prevent overfitting. Overfitting [Die95] is a common problem in deep learning and typically occurs when the model learns the training set so well that it cannot generalize to new data inputs. Overfitted models lack their predictive ability when it comes to unseen data. Consequently, regularization is utilized to modify the gradient so that it avoids going in directions that lead to overfitting. Regularization is a wide term and comprises many techniques including: Dropout, DropConnect, L1 Parameter Regularization (Penalty), L2 Parameter Regularization (Penalty) and Data Augmentation.

Dropout is a regularization technique introduced recently by Krizevsky et al. [HSK$^+$12] to prevent overfitting. Essentially the idea of the dropout technique is to learn less in order to learn better. Hence, the method consists of setting the output of every hidden layer to zero with a probability of 0.5%. The "dropped out" neurons do not contribute and participate neither in the forward nor in the backward pass. This technique is computationally inexpensive and can be applied to any type of neural network. The main advantage lies in the reduction of co-adaptation between the neurons. The neurons are thus forced to learn more robust features which results in a more generalized model that performs better on held-out data. [HSK$^+$12]

DropConnect, on the other hand, performs the same operation as Dropout. DropConnect [WZZ$^+$13] is not an actual variant of Dropout as instead of operating on the hidden layers, it temporarily mutates the connections between the neurons.

The **Laplacian L1** and **Gaussian L2 penalties** [HP89][KH92], in contrast, avoid overfitting by preventing the neural network parameter space from getting too big in one direction. Fundamentally, they make large weights smaller (weight decay)[HP89][KH92]. Krogh and Hertz [KH92] were among the first to practically examine the effects of weight decay. The L1 penalty is computationally inefficient and multiplies the absolute value of the weights instead of their squares. Interestingly, the L1 penalty has a built-in mechanism for feature selection. The L1 has a smaller penalty for larger weights, yet it drives many weights to zero. This means the resulting weight vector can be rather sparse. Contrary to L1, the L2 penalty is in fact computationally efficient as a result of its analytical solutions and non-sparse outputs. The L2 penalizes larger weights heavily but it does not drive them to zero. It does this by adding a term to the objective function to decrease the weights. L2 penalty does not have an automatic feature selection but overall it effectively improves generalization and helps the model ignore unnecessary weights.

Data augmentation [KSH12a] is the simplest and easiest method to reduce overfitting of a neural network model. It consists of artificially enlarging the amount of data and with that allowing the model to train on more data. The technique of data augmentation has been particularly effective for the specific classification problem of object recognition [PW17]. There exist many ways to artificially augment a dataset. Techniques like cropping, scaling, rotating, illuminating can all be used separately or combined to achieve this task [KSH12a][PW17].

2.4 Convolutional Neural Networks

Among the artificial neural networks there exists a special kind of neural network called Convolutional Neural Network (CNN) [KSH12b]. CNNs have been initially developed by LeCun in 1989 [LBD+89], but only received the attention and popularity they deserve in the last decade. By achieving outstanding results in several challenging Computer Vision tasks [KSH12b][SEZ+13], CNNs have installed themselves as the go-to method in this field. As indicated by their name, CNNs employ convolutions which form the core building blocks of the network [SSM+16]. Convolutions (convolutional layers) separate CNNs from the traditional neural networks which use general matrix multiplication instead [GBC16][SSM+16]. Though this difference might seem small, convolutions do much more than just substitute the general matrix multiplication. In fact, convolutions provide many important features that help CNNs outperform the standard neural networks including: parameter sharing and local connectivity [GBC16][SSM+16]. Before discussing these features in detail, we need to explain the notion of convolution and convolutional layer first.

A convolution is a mathematical operation which is performed on two real-valued functions with the goal to produce a third function [GBC16]. Essentially, the convolution operation merges the two inputs in order to produce an output.

$$s(t) = (x * w)(t) \tag{2.2}$$

The first argument (x in the Formula 2.2) is often referred to as *input* whereas the second argument w is referred to as *kernel* [GBC16]. In terms of CNNs and the field of Computer Vision, the input argument is an input image represented as a matrix of pixels and the kernel is defined as a specific set of features, represented as a multi-dimensional matrix [GBC16]. The kernel is also known as filter and is responsible for detecting the specific features within the input image [ATY+18]. The output produced by the convolution operation is known as *feature map* [GBC16][ATY+18]. Finally, the feature map will serve

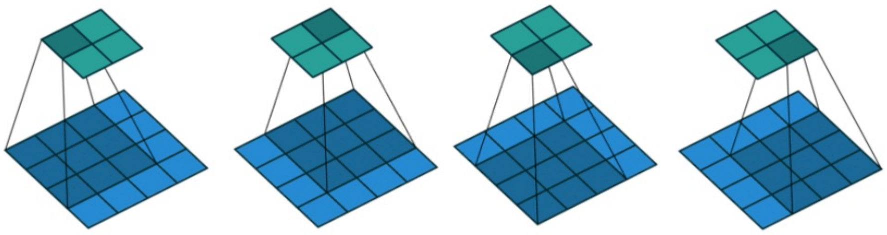


Figure 2.6: Convolving 4x4 image with 3x3 kernel results in 2x2 feature map [Zaw18]

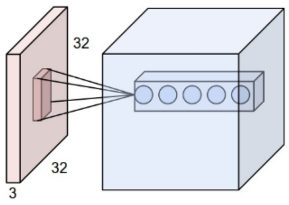


Figure 2.7: Receptive field of a neuron in CNNs [Kar16]

as the input image for another layer of filters. Overall, a convolution can be described as a process in which the network makes an effort to label the input image by referring to something previously learned [ATY$^+$18]. In Figure 2.6 the process of convolving a 4x4 image with a 3x3 kernel is depicted. The process results in a 2x2 feature map.

We now discuss the advantages that the utilization of convolutions brings.

Local connectivity. In traditional neural networks, every input unit interacts with every output unit [RHW88]. This is due to the full connectivity of the layers of the standard neural networks which is often unnecessary. Convolutional neural networks, however follow a concept known as sparse interactions or local connectivity [GBC16]. That means that each neuron is connected to and considers only a small region of the input and disregards the rest. In this way spatial position is not disregarded and only relations between close pixels are considered. The input region connected to the neuron is a hyperparameter and is known as *receptive field* of the neuron [GBC16] [ATY$^+$18]. An example of the receptive field of a neuron is provided in Figure 2.7. Each receptive field is represented as a 3D space with equal height and width. Furthermore, the receptive field of a neuron is not exclusive and can overlap with the receptive field of another neuron [ATY$^+$18]. The concept of local connectivity is tightly connected to the concept of parameter sharing, which is discussed next.

Parameter sharing is another important concept used in CNNs with the goal to reduce the number of parameters used in the convolutional layer [GBC16]. The basic idea behind the concept of parameter sharing is to look for the same recognizable set of elements or features around the whole input image [GBC16][SSM$^+$16]. The motivation for parameter sharing is derived from the assumption that if a feature is useful at some specific spatial position, then it could be useful at a different position as well [SSM$^+$16]. In

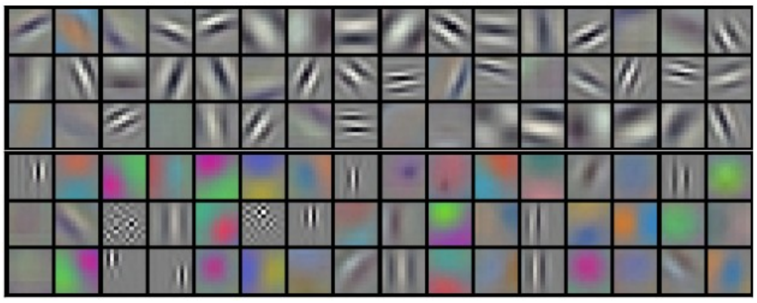


Figure 2.8: Example of a feature map learned by AlexNet [KSH12b]

traditional neural networks, each element of the weighted matrix is used exactly once when computing the output of a layer [GBC16]. However, in convolutional neural networks every element of the kernel is used at every position of the input [GBC16][SSM$^+$16]. This makes the training of large networks feasible as the number of weights or filters is controlled to remain relatively small. It is important to note that the application of a parameter sharing scheme might not always be reasonable [SSM$^+$16][TYRW14]. This would be the case where one expects to always learn completely different features from one side of the image [TYRW14]. A typical example are faces (different eye and hair color) located centrally within the image [SSM$^+$16][TYRW14]. In that case, only local connections without weight sharing are used and the layer is called Locally Connected Layer [TYRW14].

2.4.1 Convolutional layer

When dealing with convolutional neural networks and thus convolutional layers it is important to mention that every convolutional layer receives an input volume of size *width*1 x *height*1 x *depth*1 [PG17]. Additionally there are four main hyperparameters that every convolutional layer requires and need to be considered: depth, kernel size, stride and padding [ATY$^+$18]. The hyperparameters are important as they control the size of the output [ATY$^+$18]. Eventually, every convolutional layer outputs a volume of size *width*2 x *height*2 x *depth* where the depth of the output volume is equal to the value of the depth hyperparameter [PG17]. We now discuss the four hyperparameters of the convolutional layer.

Depth. The depth hyperparameter specifies the number of filters (number of kernels) to be used within that convolutional layer [PG17]. Each filter would learn to look for something different within the input volume. Additionally, the depth hyperparameter sets the depth (the desired number of channels) of the output volume [PG17].

Kernel size. The kernel size sets the width and the height of the kernel and at the same time it sets the width and height of the receptive fields of the neurons. The width and height are usually chosen to be of the same size. [PG17].

Stride. The stride hyperparameter specifies the number of pixels we move the convolution

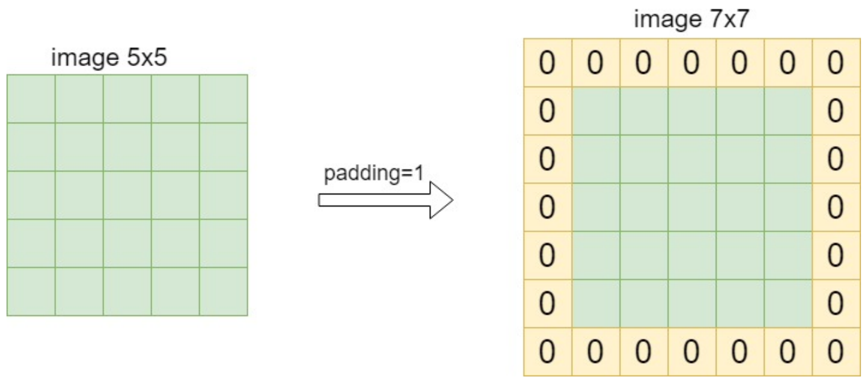


Figure 2.9: Applying padding of 1 on an input image of size 5x5

filter at every iteration step [PG17]. The default value of the stride is 1, meaning the filter is moved by only one pixel. However, if we want to reduce the overlapping of the receptive fields between neurons, then choosing a larger stride value would be more suitable [PG17]. A larger stride value also means skipping of some possible locations and potential loss of vital information, which eventually results in a feature map of smaller size [GBC16]. Hence, choosing the right stride value is of crucial importance.

Padding. In order to control the spatial size and avoid fast decrease of dimensionality of the output volume it is useful to pad the input image with (most oftenly) zeros around the border [GBC16][ATY+18]. The padding hyperparameter allows us to specify the size of the padding we want to be added to the input volume. By padding the input volume the size of the feature map (output volume) is preserved, which would otherwise shrink too rapidly [GBC16][ATY+18]. Since padding with zeros is the most common pattern, this hyperparameter is also known as zero-padding [PG17]. Figure 2.9 shows a practical example of how an image looks after a padding of 1 has been applied.

Figure 2.10 shows an example of a convolutional operation of an input image of size 7x7 and a kernel of size 3x3. Assuming the stride is set to 1 and the padding is set to 0, then by convolving the input image with the kernel, an output image (feature map) of size 5x5 will be obtained. If a padding of 1 instead of 0 is used, after convolving the input image with the same kernel, the size of output image (feature map) will be 7x7. This shows that the utilization of padding prevents the size of the output image to shrink [ATY+18]. Since convolutions are often stacked, without the use of padding the size of the image will be reduced fast and the information will be lost.

2.4.2 Architecture

As explained earlier in this chapter, the building blocks of a CNN are the convolutional layers [SSM+16]. Apart from the convolutional layers, a typical architecture of a convolutional neural network (CNN) also includes a pooling layer and a fully connected layer [ATY+18]. In Figure 2.11 a common CNN architecture is shown. The first rectangle

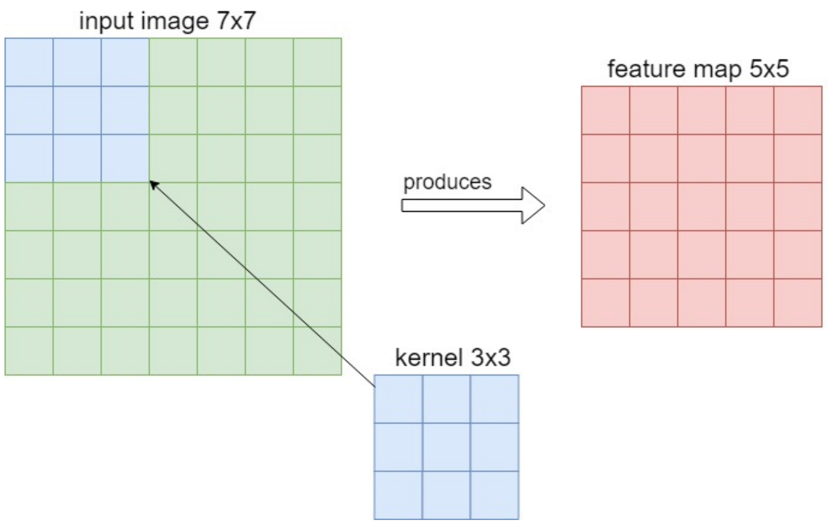


Figure 2.10: Performing convolutional operation on a 7x7 input image with a 3x3 kernel

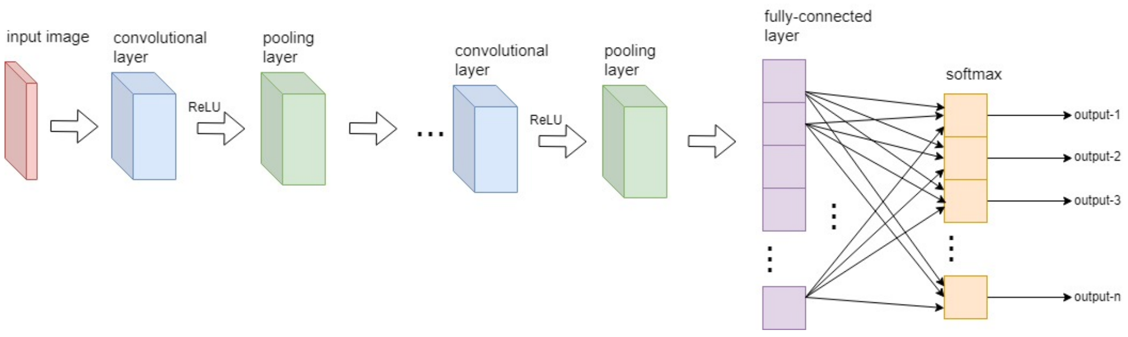


Figure 2.11: Architecture of Convolutional Neural Network (CNN)

represents the input image, the second rectangle depicts the convolutional layer, the third rectangle represents the pooling layer and the second to last rectangle is the fully connected layer. Important to note is that the convolutional and pooling layers have three dimensions (hence, the input is of size *width* x *height* x *depth*) and the input of the fully connected layer is a 1D array. The output of the whole CNN is determined by the softmax function [ATY+18]. The activation function most commonly used in CNNs is the ReLU activation function and is typically computed after the convolutional layer or after a stack of convolutional layers [SZ14b][HZC+17]. Moreover, the ReLU activation function forms another layer within the convolutional neural network [SZ14b][HZC+17].

We now focus on the other layer types that occur in the architecture of convolutional neural networks. For each layer type, we will discuss its importance and the purpose it serves individually.

Pooling layer. As witnessed by the example in Figure 2.9, when using a convolutional layer with padding of size 0, the size of the image will rapidly decrease. The typical layer used for controlling the size of the image in CNNs is the pooling layer [GBC16]. Commonly, the pooling layer is inserted in-between successive convolutional layers or convolutional layer stacks [PG17]. The core function of the pooling layer is to slowly, but continuously reduce the spatial size of the output volume in order to reduce the number of parameters within the network [PG17]. Hence, the pooling layer mainly helps in reducing the amount of computation required and prevents overfitting [PG17]. The hyperparameters for the pooling layer are: *type*, *kernel* and *stride* [ATY+18]. The parameter *type* specifies the type of the pooling operation that needs to be performed. The most common one is *max_pooling* [ATY+18]. For example when the kernel size is 2x2 and the stride is set to 2, then the image is reduced by a factor of 2 (see Figure 2.12). With the pooling operation, only the size of the image is spatially downsampled (reduced), whereas the depth dimension stays the same [ATY+18]. Apart from the max_pooling, there exist other types of pooling such as sum pooling and average pooling [ATY+18].

Normalization Layer (optional). This layer is used to normalize the output values of a layer during the process of training the convolutional neural network [IS15]. Sometimes during training, the weight values can vary by large margins, so it is always good to have normalized values. If one of the weights of the neurons becomes drastically larger than the weights of the other neurons, then this weight will be cascaded through the network and can generally cause instability of the network leading to the exploding gradient problem. That is why a new normalization layer is created which is called *BatchNormalization* [IS15]. As indicated by its name, this layer performs a batch-wise normalization of the output values of the activation function of the specific layer it has been applied to. It takes the output of the activation function of the previous layer, and from it, subtracts the batch mean and divides it by the batch standard deviation [IS15].

Fully Connected Layer. The Fully Connected Layer is attached at the very end of the convolutional neural network (see [SZ14b][HZRS15] [HZC+17]). Before feeding the output of the previous layer, the output (feature map) matrix is flattened into a vector and fed into the fully connected layer which corresponds to the ones in traditional neural networks [PG17]. The fully connected layer outputs an n-dimensional vector where n is the number of possible target classes the network has to choose from [ATY+18]. Each value of the n-dimensional vector corresponds to the probability of a certain class (softmax approach) [ATY+18]. The way the fully connected layer works is that it looks at the feature map (output of the previous layer) and determines what high level features correspond most to a particular class [ATY+18]. It also has specific weights. Thus, by multiplying the weights and the previous layer, the probabilities of the classes are computed.

2.4.3 Types of Convolutions

It is a fact that convolutional neural networks and their convolutional layers have revolutionized the field of Computer Vision with their speed and powerful performance [KSH12a][SZ14b] [HZRS15]. The question that remains to be discussed is the following:

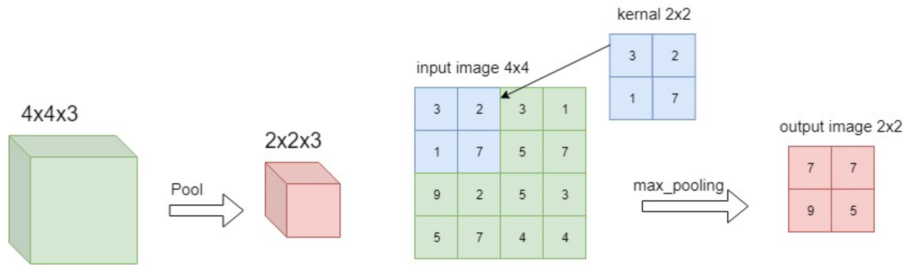


Figure 2.12: An example of a pooling operation on input image 4x4x3 with kernel size 2x2 and stride 2

Has the top performance been reached with the convolutional layers or can they be further optimized to unravel their full potential? Hence, in this section we focus on the existing types of convolutions, reviewing their advantages and the improvement on performance they bring. Generally, there exist several types of convolutions, the most important ones of which include normal convolutions and separable convolutions [GLL$^+$18]. The introduction of separable convolutions has shown that convolutional layers can learn to use their parameters even more effectively [Cho16][GLL$^+$18]. There are two well-known types of separable convolutions: spatial separable convolutions [JVZ14] and depthwise separable convolutions [Cho16].

Spatial separable convolutions are the simpler kind of separable convolutions [JVZ14]. Essentially, spatial separable convolutions work in the way that they separate the convolution into two convolutions [JVZ14][TXWE15]. Spatial separable convolutions deal with the spatial dimension of the image (height and width) and the kernel [JVZ14][TXWE15]. They divide the original kernel into two smaller kernels with the goal to reduce the number of multiplications and thus the computational complexity [JVZ14]. Though the idea of doubling the number of convolutions to reduce the number of multiplications necessary, might appear counter-intuitive, the practice proves otherwise [JVZ14][TXWE15]. For instance, if we have an input image and a kernel of size 3x3, the number of multiplications to be performed is 3*3=9. Now, assuming we split the kernel of size 3x3 into two kernels of size 3x1 and 1x3 respectively, the number of multiplications goes down to (3*1)+(1*3)=6. So just by splitting the kernel into two less complex kernels, the number of multiplications is reduced by 30%. Furthermore, by reducing the number of multiplications, the computational complexity is reduced and the performance speed is increased [JVZ14]. Spatial separable convolutions thus provide a great performance gain over normal convolutions. However, the downside of using spatial separable convolutions is that not all kernels can be separated into two kernels i.e. convolutions [JVZ14][TXWE15]. As a result, spatial separable convolutions do not always come as a first pick when choosing the right convolution type.

Normal convolutions have been covered by the previous section, hence now we will just briefly review the core concepts and the process with the help of a practical example. An input image is represented by three dimensions: width, height and depth, where the depth dimension is also known as the number of channels of the image. Assume we have an input image of size 7x7x3 (an image with 3 channels) and a kernel of size 3x3 and stride of value 1. Convolving the image with the filter will produce an output image of size 5x5x1 if the number of output channels is one. Clearly, the size of the output has been reduced. Since in the process of training, many convolutions need to be carried out, fast downsizing of the number of channels would lead to loss of vital information and would badly affect the training process in general. Now assume we want to prevent this and instead of an output image with one channel we wish for an output image with ten channels. We can achieve this by creating ten 3x3x3 kernels. Each of the kernels would produce an output image of size 5x5x1, hence we would end up with 10 5x5x1 output images. If we stack them up together we would end up with an output image of size 5x5x10. Although we achieved the desired number of channels, this process is complex and requires a large number of multiplications. The number of multiplications necessary can be calculated with the help of the following formula [GLL+18]:

$$outputChannels * kernel * inputChannels * outputImage(width, height) \tag{2.3}$$

Hence in our case we have 10 3x3x3 kernels that we move 5x5 times. The total number is thus, 6750 multiplications. The way to optimize this process is by using depthwise separable convolutions [GLL+18].

Depthwise separable convolutions, in contrast to spatial separable convolutions deal with the depth dimension of the input image (weight, height, depth) [GLL+18]. Furthermore, they are often preferred over spatial separable convolutions as they are more powerful and have less constraints to their application. The process of doing depthwise separable convolution is broken into two parts i.e. convolutions: the first one is depthwise convolution and the second one is called pointwise convolution [GLL+18] [Cho16].

1. The depthwise convolution is applied to a single channel of the input and focuses on the spatial relationship modeling [GLL+18]. For example if the input is 7x7x3, the kernel is 3x3 and the number of desired output channels is 10, then three 3x3x1 convolutions will be created. Every 3x3x1 kernel iterates over one channel of the image and produces an output image of size 5x5x1. Stacking the images together will create a 5x5x3 output image. The complexity of the depthwise convolution is provided by Equation 2.4 [GLL+18]. In order to obtain the number of wished output channels, the pointwise convolution is applied next.

$$inputChannels * kernel * 1 * outputImage(width, height) \tag{2.4}$$

2. Pointwise convolution is a convolution with a kernel of size 1x1 and is typically applied on all of the channels (cross-channel relationship modeling) of the input image [GLL+18]. Therefore, a 1x1x3 kernel is created that iterates over the 5x5x3 image to get 5x5x1 image. Providing ten kernels of size 1x1x3 will output ten 5x5x1 images and then stacking them would result in the output image of size 5x5x10. Equation 2.5 provides the complexity of the pointwise convolution [GLL+18].

$$outputChannels * 1 * 1 * inputChannels * outputImage(width, height) \tag{2.5}$$

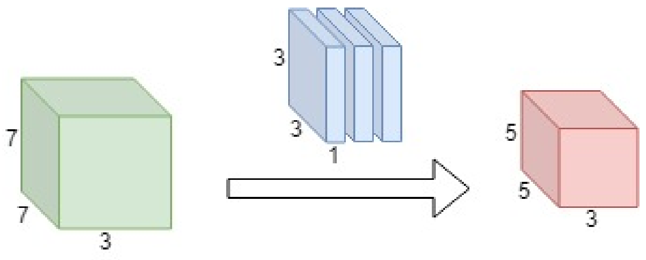


Figure 2.13: Depthwise convolution: input image 7x7x3 and kernel 3x3x1

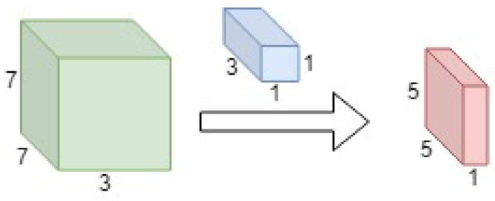


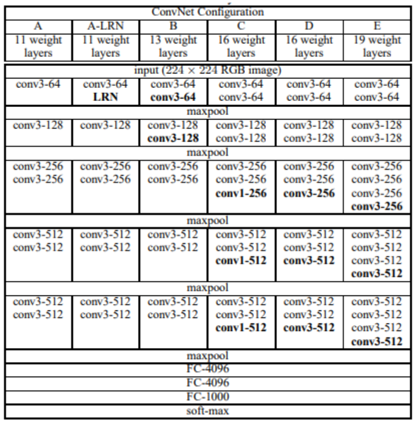
Figure 2.14: Pointwise convolution: input image 7x7x3 and kernel 1x1x3

Essentially, this process is similar to the one of the normal convolutions. In both cases the image is passed through a 3x3 kernel and from 3 channels is expanded to 10 channels. The main difference is that in normal convolutions the whole image is transformed n times (where n is the number of wished output channels), whereas in the depthwise separable convolutions the image is transformed only once, during the depthwise convolution process [GLL+18]. We have seen that it takes 6750 multiplications for normal convolutions to compute this example. Depthwise separable convolutions can compute the same example with 1425 multiplications. (First depthwise convolution 3*3*3*1*5*5=675, Second pointwise convolution 10*1*1*3*5*5=750). Cutting down the number of multiplications increases the inference speed of the convolutional neural network and reduces the number of parameters [GLL+18].

2.4.4 Feature Extractors

The computer vision task of object recognition is separated into two main subtasks: classification and regression [ZZXW18]. The classification task is computed by a CNN model for classification also known as feature extractor whereas the localization of an object within an image is performed by a regression task [ZZXW18]. The regression task is also computed by a CNN model, called object detector [ZZXW18]. In the following the most famous state-of-the-art feature extractors will be analyzed with the accomplishments they achieved on the ImageNet Large Scale Visual Recognition Competition (ILSVRC) [RDS+15] classification challenge. ImageNet Large Scale Visual Recognition Competition (ILSVRC) [RDS+15] is a competition which involves classifying an image into one of 1000 classes.

VGG proposed by Simonyan et al. [SZ14b] is a very deep convolutional neural network

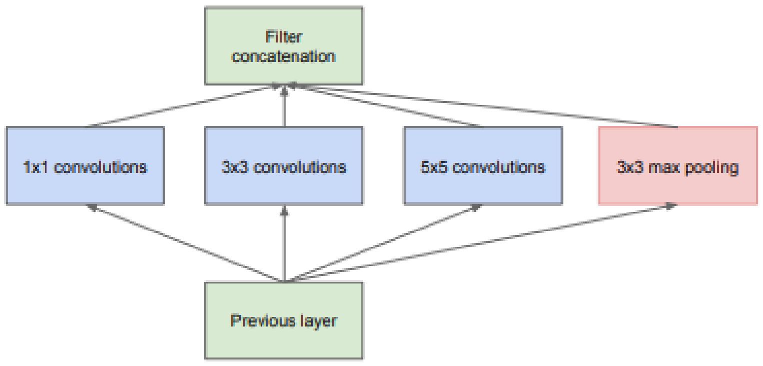


ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

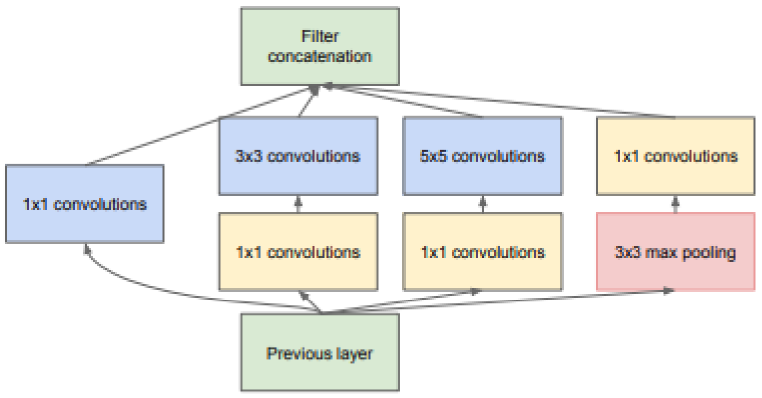
Figure 2.15: Architecture of VGG [SZ14b]

model. Simonyan et al. [SZ14b] investigate the relationship between the architecture of a convolutional network and the accuracy it achieves. In their work, they show that an increased depth and small size convolution filters result in higher efficiency and better accuracy. The convolutions that VGG uses are configured with a kernel size of 3x3, padding of 1 and a stride of 1. Utilizing convolutions with a kernel size of 3x3 has proven to have many advantages. Their experiments show that stacking three 3x3 convolutions is essentially the same as one 7x7 convolution, however the computation is more efficient [SZ14b]. The idea behind this is to reduce the number of parameters and to make the decision function more discriminative. The convolutional neural network VGG consists of several configurations: VGG-11, VGG-16 and VGG-19 [SZ14b]. At the end of each configuration there are three fully connected layers and a softmax layer used for classification. VGG-11 consists of 8 convolutional layers, VGG-16 of 13 convolutional layers and VGG-19 consists of 16 convolutional layers. The activation function used in all of the layers is ReLU. There are 5 pooling layers which are applied after a stack of convolutions, with type max_pool, kernel 2x2 and stride 2. For Simonyan et al. [SZ14b] increasing the network depth and decreasing the number of parameters worked well as VGG won both the first place of the localization challenge and second place in the classification challenge in the ILSVRC 2014. Figure 2.15, demonstrates the configurations of the convolutional neural network VGG [SZ14b].

GoogleNet, developed by Szegedy et al. [SLJ+14] won the ILSVRC 2014 by achieving a top-5 error rate of below 7%. At its core, GoogleNet represents a state-of-the-art 22-layer deep neural network used for classification and detection purposes [SLJ+14]. The authors try to exploit the architectural properties of a convolutional network by increasing the depth of the network and reducing the dimension of the convolutional layers. In fact, the network's great performance is largely due to its massive depth. The so-called *inception modules* (sub-networks used by GoogleNet) allow a much more efficient utilization of the parameters, when compared to the VGG [SZ14b] architecture [SLJ+14]. Each module consists of 1x1, 3x3, 5x5 convolutional layers and a 3x3 max



(a) Naive version

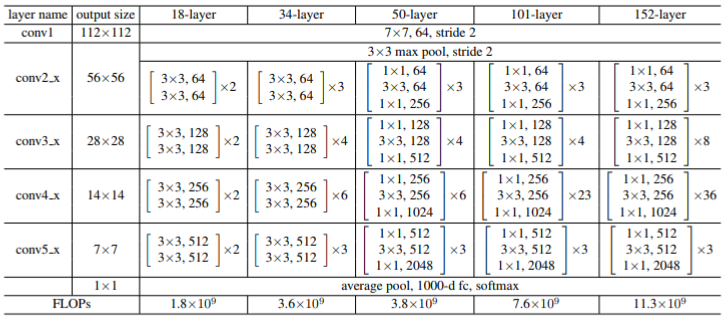


(b) Dimension reductions

Figure 2.16: Inception module [SLJ+14]

pooling layer to increase the diversity of the model and obtain different type of patterns. GoogleNet is also called Inception, because the building blocks of the network are the inception modules [SZ14b]. The Inception neural network consists of a stack of modules shown in Figure 2.16, periodically followed by pooling layers of type max_pooling and stride 2 to reduce the size of the image. Apart from pooling layers of type max_pool, pooling layers of type avg_pool can also be observed within the Inception architecture. Furthermore, fully connected layers are also used and are followed by a softmax layer used for classification. Szegedy et al. [SZ14b] explicitly favour 1x1 convolutions in order to reduce the computational cost, yet achieve a greater depth. The architecture of GoogleNet keeps the computational costs constant, but at the same time provides a significant quality gain. [SZ14b]

Residual Networks, known as ResNet [HZRS15], which as a winner of the ILSVRC 2015 also takes a significant spot on this list, with an achieved top-5 error of 3.57%. Developed by Kaiming He et al.[HZRS15] ResNet follows the trend of *"deeper is better"*

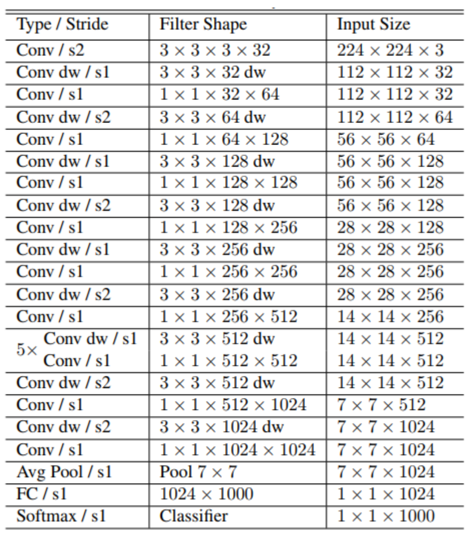


layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2.17: Architecture of ResNet [HZRS15]

and is built up of 152 convolutional layers (see Figure 2.17). This makes ResNet eight times deeper, but nonetheless, not more complex than VGG [SZ14b][HZRS15]. Instead of learning unreferenced functions, in this architecture Kaiming et al. [HZRS15] defined the layers in terms of learning residual functions with the reference to the input layers. This means if the target underlying mapping is $H(x)$, then the stack of non-linear layers are redefined as a residual mapping of the form $F(x) = H(x) - x$, casting the original mapping to $F(x) + x$. In their paper, the authors prove that it is easier to optimize the residual mapping rather than the initial unreferenced mapping $H(x)$. Due to its depth, the training of the network posed a massive challenge. For this reason Kaiming et al. [HZRS15] introduced the so called *skip (shortcut)* connections to make the training feasible. Shortcut connections serve the purpose of skipping one or more stacks of convolutional layers and are essentially represented by the identity function [HZRS15]. Overall this helped ResNet exhibit large accuracy gains and enjoy great increase in depth while preserving a minimal error rate. The structure of the block of convolutional layers consist of kernels of sizes: 1x1, 3x3 and 1x1. Furthermore, the convolutional layers perform downsampling of the size of the image by using a stride of 2. It is important to note that ResNet utilizes only two pooling layers: as a second layer with kernel 3x3, stride 2 and type max_pooling and lastly before the fully connected layer, where the type of the pooling layer is avg_pooling. [HZRS15]

MobileNet [HZC$^+$17], is an efficient, lightweight convolutional neural network with significantly less parameters than VGG, Inception and ResNet. The secret behind MobileNets lies in the fact that Howard et al. [HZC$^+$17] cleverly utilize depth-wise separable convolutional layers instead of convolutional layers. The kernel used in the depthwise convolution is of size 3x3, with stride 2 and padding 1. As previously discussed in Section 2.4.3, after the depthwise convolution, a pointwise convolution with kernel 1x1 is performed, with the goal to adjust the number of output filters. A Batch Normalization layer [IS15] is also utilized within the architecture of MobileNet and is included after every depthwise and pointwise convolutional layer [HZC$^+$17]. The activation function used in MobileNet is ReLU. At the end of the network there is an average pooling layer and a fully connected layer followed by a softmax layer used for classification. MobileNet is the most suitable convolutional neural network to be used in mobile and embedded devices,



Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
5× Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 2.18: Architecture of MobileNet [HZC+17]

because of its low complexity and the high accuracy it achieves [HZC+17]. Due to the low complexity and its lightweight architecture MobileNet requires only low computational power. The great performance of MobileNets has been established across significant range of applications, including: object detection, large scale geo-localization and face attributes [HZC+17].

2.4.5 Object detectors

The problem of object recognition differs from the simpler problem of image classification in that it also requires localization of (possibly multiple) objects within the image [ZZXW18]. The focus of this section is on the localization sub-task which, as stated in the previous section, is essentially a regression task performed by a fast CNN model, known as object detector [ZZXW18]. In the following, the state-of-the-art object detectors are discussed.

R-CNN. In 2014 Girshick et al. [GDDM14] tried to conquer the problem of object localization and detection jointly, by taking a slightly different approach. They propose a solution which follows a region-based paradigm which, as the name suggests, performs recognition within designated regions. The solution presented by Girshick et al. [GDDM14] first proposes 2000 category-independent regions for each input image. The region proposals are generated by a fast, selective search algorithm and together represent a set of candidate detections for the given image. From all of the region proposals, a feature vector of fixed length is extracted by means of convolutional neural networks. Lastly, a category-specific linear Support Vector Machine (SVM) [SC08] is employed to score the extracted feature vectors and perform the classification of each individual region. Since this approach successfully employs the utilization of CNNs as well as of region

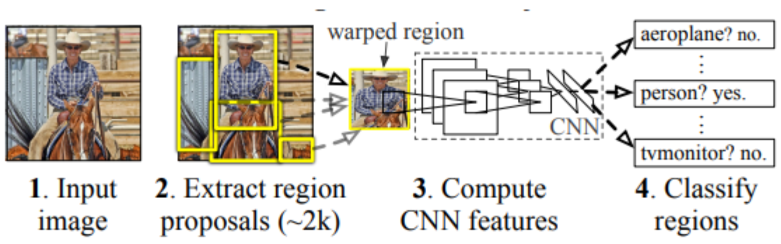


Figure 2.19: Architecture of R-CNN [GDDM14]

proposals, it has been described as a Region-based CNN or R-CNN [GDDM14]. With this technique, R-CNNs have achieved the best object detection results so far (62.4% Mean Average Precision mAp) on the PASCAL VOC 2012 [EVGW+10] dataset. The method has also been applied and proven to work well on the problem of semantic image segmentation [CS12][GDDM14].

Fast R-CNN. Even though the R-CNN approach performed well, it was still fairly slow [GDDM14]. In particular, the training of the R-CNN networks was quite time-consuming, since for each image 2000 regions had to be classified. Furthermore, the selective search algorithm, which produced these regions, was fixed and predicted the regions without any prior knowledge [GDDM14]. Naturally, the algorithm sometimes produced regions of little interest. This prompted Girshick et al. [Gir15] to further improve and develop R-CNN which resulted in a network we now know as Fast R-CNN. The Fast Region-based Convolutional Neural Network builds upon its predecessor R-CNN and has a couple of innovations that make it faster. Firstly, the Fast R-CNN utilizes the very deep VGG-16 convolutional neural network [SZ14b] as feature extractor [Gir15]. Secondly, instead of taking just region proposals like in R-CNN, the Fast R-CNN takes the whole input image and a set of object proposals [Gir15]. The input image is then processed through several convolutional and max pooling layers, so that a convolutional feature map is produced. From the convolutional feature map, a region of interest RoI pooling layer extracts a feature vector of fixed size for every object proposal. At the end, a softmax function is used to classify the proposed object and a regressor function to predict the coordinates of the bounding box. The advantage of this approach lies in the fact that the convolutional operation is performed only once per image which results in a significant speed up in both the training and testing of the network [Gir15]. This approach achieved 68.4% mAp (the mAp measure is explained in Section 2.5.2) on the PASCAL VOC 2012 [EVGW+10] dataset. Fast R-CNNs suggest that less object proposals increase the quality and performance of the object detector and the network in general [Gir15].

Faster R-CNN. The attempts to enhance the speed and performance of Region-based Convolutional Networks did not stop with the Fast R-CNN approach. Despite the significant improvements, researchers were still unsatisfied with the results Fast R-CNNs achieved. Cleary, the bottleneck of both of the approaches R-CNN and Fast R-CNN was the selective search algorithm [GDDM14][Gir15]. Due to the vast number of regions it proposed, the performance suffered. Therefore, Shaoqing Ren et al. [RHGS15] proposed

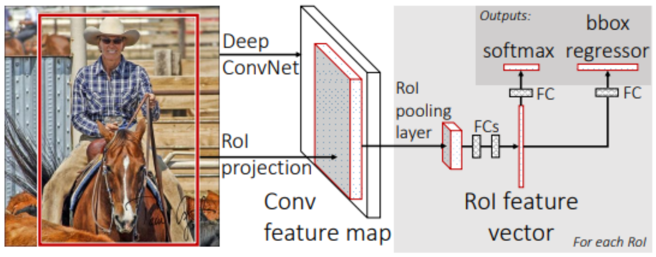


Figure 2.20: Architecture of Fast R-CNN [Gir15]

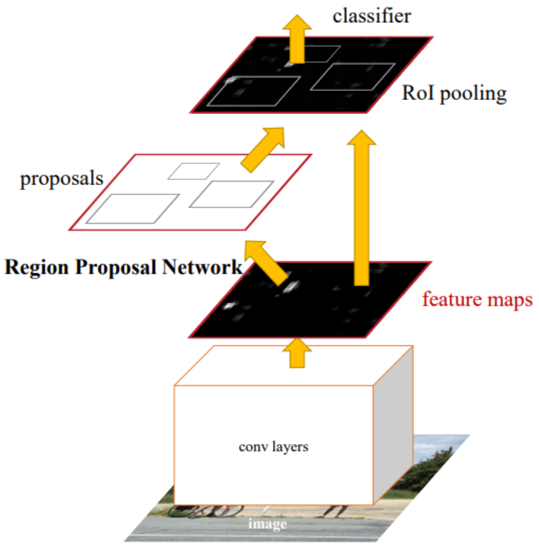


Figure 2.21: Architecture of Faster R-CNN [RHGS15]

Faster R-CNNs to overcome these limitations and achieve the desired performance. Shaoqing Ren et al. [RHGS15] completely eliminate the selective search algorithm for region proposals and cleverly replace it with another convolutional neural network. This sophisticated solution is composed of two core modules: a deep fully convolutional network that provides the region proposals (RPN) and the Fast R-CNN object detector [Gir15] used to classify the proposed regions [RHGS15]. Both of the modules are elegantly merged and operate as a single network for object detection. This architecture brought a massive increase in performance. With a 75.9% mAp score on the PASCAL VOC 2012 [EVGW$^+$10] dataset, the Faster R-CNN method initiated and gave serious hope to the idea of real-time object detection [RHGS15].

YOLO. In 2016 Redmont et al. [RDGF15] introduced a novel approach with a completely different approach to solving the problem of object detection. Nowadays, the approach represents a revelation in the branch of real-time object detection and is primarily known as YOLO or the *"You only look once"* neural network. In their paper, Redmont et al. [RDGF15] redefine the object detection problem to pure regression problem which

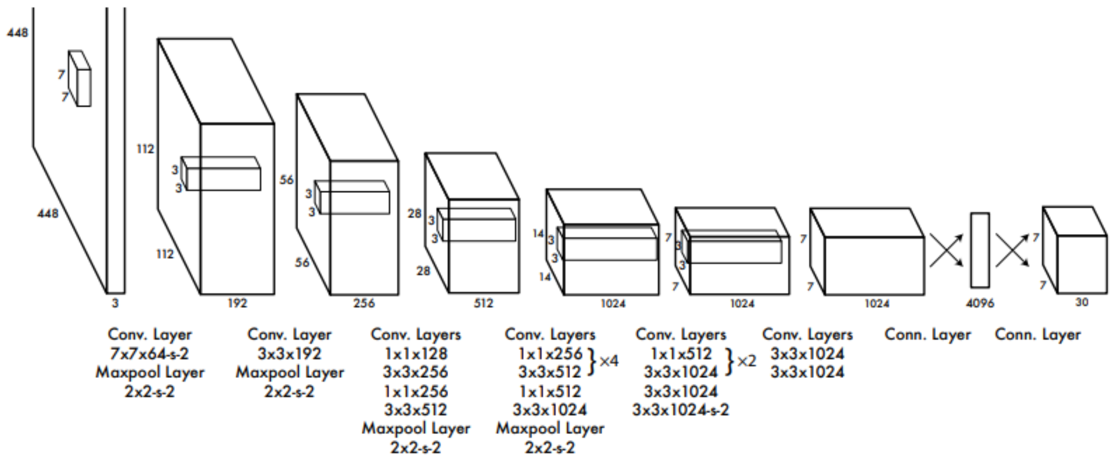
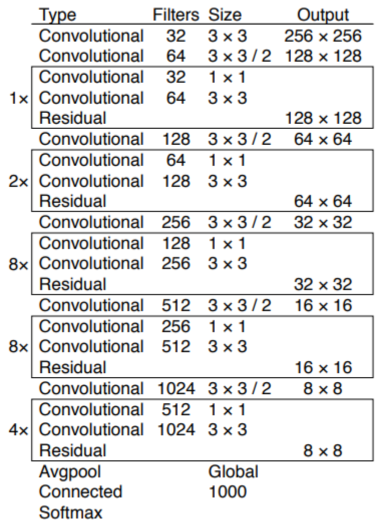


Figure 2.22: Architecture of YOLO [RDGF15]

requires a simple pipeline. A convolutional neural network takes an image as input, looks at it only once (hence the name) and simultaneously predicts the bounding boxes and the class probabilities for those boxes. The YOLO CNN model is comprised of 24 convolutional layers, pooling layers of type max_pool and 2 fully connected layers at the end [RDGF15]. The YOLO architecture favours convolutional kernels of size 1x1 and 3x3. Furthermore, YOLO trains the network on full images and sees the full images during test time as well [RDGF15]. This means, it implicitly encodes contextual and background information about the classes and the possibilities of their appearances. As a result, YOLO is highly generalizable and makes 50% less background errors when compared to Fast R-CNNs [Gir15][RDGF15]. There also exists a reduced, faster version of YOLO which is called Fast-YOLO and the architecture contains 9 convolutional layers instead of 24. YOLO runs at 45 FPS and can easily process a video data stream in real-time. On the PASCAL VOC 2007 and 2012 [EVGW$^+$10] datasets, the YOLO approach achieves 63.4% and 57.9% mAp respectively, while running in real time [RDGF15].

YOLO9000 and **YOLOv2.** Not long after the success of YOLO, Redmont et al. [RF16] released another paper introducing two new versions of YOLO: YOLO9000 and YOLOv2. YOLO9000 represents a model with the ability of detecting over 9000 different object classes [RF16]. The new model could retain its speed and still run in real-time, despite the large increase in object categories. On the other hand, YOLOv2 introduces slight changes to the state-of-the-art YOLO architecture, in order to increase the accuracy [RF16]. In contrast to YOLO, YOLOv2 takes as input images with higher resolution, with the goal to enable better object detection of small objects. Furthermore, YOLOv2 uses Darkenet-19 [RF16] as feature extractor and introduces batch normalisation [IS15] to all convolutional layers of the model in order to prevent overfitting of the network [RF16]. To predict the bounding boxes, YOLOv2 uses anchor boxes. The anchor boxes are hard-coded and completely replace the fully-connected layers of the model. This



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

Figure 2.23: Architecture of DarkNet-53 [RF18]

makes YOLOv2 a fully-convolutional network and was inspired by the Faster R-CNNs [RHGS15] which also follow the anchor box approach [RF16]. YOLOv2 obtains the dimensions of the anchor boxes by the K-Means [AV07] clustering algorithm which is run on the training set. The utilization of anchor boxes brings a slight drop in the mAp score of YOLOv2 but significantly increases the recall. Overall, YOLOv2 outperforms YOLO and Faster R-CNN achieving 78.6% mAp at 40 FPS on the PASCAL VOC 2007 and PASCAL VOC 2012 [EVGW$^+$10] datasets [RF16].

YOLOv3. The latest version of YOLO is YOLOv3, released in the likewise named paper by Redmont et al. [RF18] in the mid of 2018. The new version of YOLO shows some significant improvements over YOLOv2 [RF16] in predicting small objects. YOLOv3 performs multilabel classification and predicts the bounding boxes over three different scales [RF18]. Each of the scales uses three anchor boxes which totals in nine anchor boxes. From these scales, features are extracted using a novel feature extractor. The new feature extractor represents a network built as a combination of the network used in YOLOv2, Darknet-19 [Red16] and a residual network. The network's architecture comprises convolutional layers with kernels 3x3 and 1x1 stacked on top of each other as well as shortcut connections which make the network much larger in size. The resulting feature extraction network has 53 layers and is called Darknet-53 [RF18]. Darknet-53 [RF18] is much more powerful than Darknet-19, 1.5 times faster than ResNet-101 [HZRS15] and 2 times faster than ResNet-152 [HZRS15]. This helps YOLOv3 perform faster and hence stay competitive and even outperform the current state-of-the-art object detection systems. Interestingly, Redmont et al. [RF18] report that having solved the *"detection of small objects"* problem YOLOv3 now experiences problems with detecting objects which are medium or larger in size which requires further investigation. [RF18]

SSD. Similarly to the idea of the YOLO model [RDGF15], Liu et al. [LAE$^+$15] developed

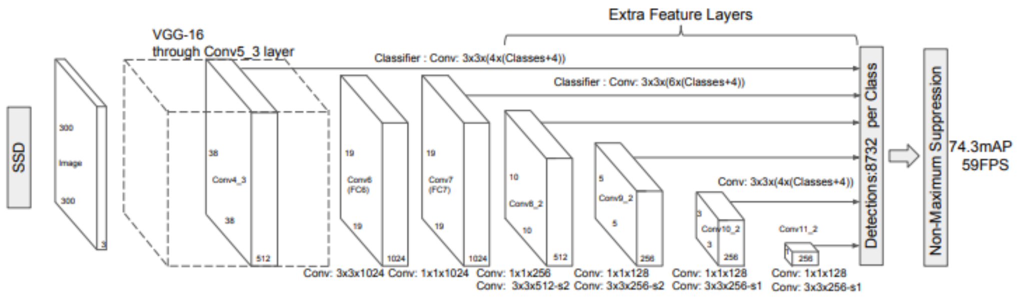


Figure 2.24: Architecture of SSD [LAE+15]

SSD, the Single Shot Detector model for object recognition. The SSD model uses a single deep neural network to predict all the bounding boxes and object class probabilities at once [LAE+15]. In Figure 2.24 the architecture of SSD is shown. SSD utilizes VGG-16 [SZ14b] as feature extractor and adds a convolutional layers to predict the bounding boxes. As input, the SSD model accepts an image which is then processed through multiple convolutional layers with different kernel sizes. While YOLO [RDGF15] uses a single scale feature map, the prediction of the bounding boxes in SSD [LAE+15] is performed with the help of multiscale feature maps. The feature maps are processed by special 3x3 convolutional layers (feature layers) to produce bounding boxes which resemble the anchor boxes used by Faster R-CNNs [RHGS15]. Moreover, during the training phase, the SSD model utilizes hard negative mining and data augmentation strategies which result in more stable training and a robust and generalizable model. Liu et al. [LAE+15] distinguish between two variations of the SSD model: SSD300 and SSD512. The SSD512 is essentially the same as SSD300 except it has an extra convolutional layer for prediction and the input size of the image is 512 and 300 pixels respectively, with the aim to enhance the performance. The best models of SSD outperform YOLO and run at 59 FPS in real time [LAE+15]. The models achieved 73.4% mAp on the PASCAL VOC 2007 [EVGW+10] for 300 x 300 input images and 76.8% mAp for 512 x 512 input images. [LAE+15]

Object detection: Summary

Table 2.1 provides an overview of the most important characteristics of the real time object detection neural networks discussed in Section 2.4.5. The datasets used for evaluation are the PASCAL VOC07 and the PASCAL VOC12 [EVGW+10].

2.5 Evaluation measures

Evaluation metrics are performance measures that provide information and describe how well a model performs. In the machine learning area there exists a number of possible ways to measure the performance of a network model. In this section we focus on the

Network	Data Shape	Feature Extractor	Convolutional layers	Pooling layers	Dataset	Accuracy	FPS
Faster R-CNN	512	VGG16	16	5 (max_pool)	12	75.9%	5
YOLO	448	YOLO CNN	24	4 (max_pool)	12	57.9%	45
YOLO9000 and YOLOv2	544	Darknet-19	19	5 (max_pool)	07+12	78.6%	40
YOLOv3	256	Darknet-53	53	1 (average_pool)	07+12	77.2%	78
SSD	300	VGG16	16	5 (max_pool)	07+12	76.8%	59

Table 2.1: Overview of the real time object detectors.

most common and important metrics used when evaluating convolutional neural network models for the task of object recognition.

2.5.1 Confusion Matrix

Confusion matrices are used to make an in-depth analysis of the achieved results of the model and compare them to the ground truth [Faw06]. They are particularly practical as they provide a good visual representation of the predictions that the model makes. Confusion matrices can be used for evaluating binary as well as multi-class classification problems [Faw06] [Pow11]. Figure 2.25 demonstrates a sample of a confusion matrix. It is composed of two columns and two rows. Basically, the matrix consists of four combinations of predicted and actual values: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) [Faw06] [Pow11]. The definition of these terms is provided below.

- **True Positive (TP):** ground truth is true and the prediction of the model is positive.
- **True Negative (TN):** ground truth is false and the prediction of the model is negative.
- **False Positive (FP):** ground truth is false and the prediction of the model is positive.
- **False Negative (FN):** ground truth is true and the prediction of the model is negative.

The case of False Positive (FP) is also known as *Type I Error* and the case of False Negative (FN) is known as *Type II Error* [Pow11]. With the help of the confusion matrix, several evaluation metrics can be calculated [Pow11][Faw06]:

- **Accuracy** is defined in terms of the ratio of the total number of correctly predicted observations to the total number of observations the model made in general. The accuracy measure can often be misleading due to the common case of uneven distribution of the classes. Hence, in order to get the true evaluation of the model,

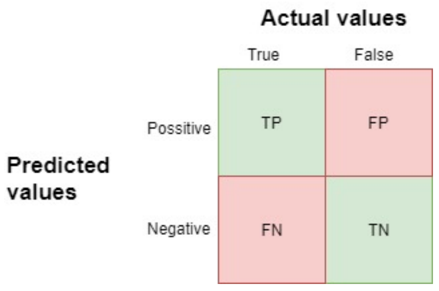


Figure 2.25: Confusion Matrix

the accuracy is often paired with other measures such as Recall and Precision [Faw06].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.6}$$

- **Precision** provides the ratio of correctly predicted positive observations to the total predicted positive observations. Precision measures how accurate the positive predictions of the model are. High precision is related to low numbers of False Positives [Pow11].

$$Precision = \frac{TP}{TP + FP} \tag{2.7}$$

- **Recall** is defined as the ratio between correctly predicted positive observations and the total number of actual true instances in the ground truth. Recall measures the model's ability of finding all the positive samples. A high recall value means that the total number of False Negatives (Type II Error) was relatively low [Pow11].

$$Recall = \frac{TP}{TP + FN} \tag{2.8}$$

- **F-measure** is the weighted average of Precision and Recall. As a result it takes into account both: the False Negatives and the False Positives. The F-measure is particularly useful when the distribution of true and false samples within the ground truth is uneven [Faw06].

$$F - measure = \frac{2 * Recall * Precision}{Recall + Precision} \tag{2.9}$$

2.5.2 Evaluation in Object Detection

Convolutional neural network models for the task of object detection are commonly evaluated using either the *Average Precision (AP)* or the *mean Average Precision (mAp)* evaluation metric depending on the number of target classes the model detects [EEG+15b].

For the task of object detection the following aspects must be evaluated [EEG+15b]:

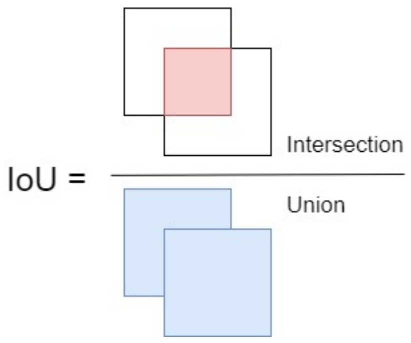


Figure 2.26: Intersection over Union (IoU)

1. The class of the detected object (classification task: *"Is the object class present within the image?"*)
2. The localization (bounding box) of the detected object (localization, regression task: *"Where is the object class present within the image?"*)

The object detection models are evaluated on a *"per class"* basis, meaning a separate score is computed for every class the model detects [EEG+15b]. The evaluation of the object detection task comes down to determining the correctness of the bounding boxes [EEG+15b]. The metric that provides the information regarding the correctness of a certain box is commonly known as Intersection over Union (IoU) [BL08]. The IoU metric provides the ratio between the intersection and the union of the predicted bounding boxes and the ground truth bounding boxes (see Figure 2.26). The intersection is the overlapping area of the predicted and the ground truth bounding box, whereas the union includes both of the regions (see Equation 2.10).

$$IoU = \frac{BBox_{\text{p}} \cap BBox_{\text{gt}}}{BBox_{\text{p}} \cup BBox_{\text{gt}}} \tag{2.10}$$

A predicted detection is considered correct or regarded as True Positive if the IoU value for the given instance exceeds 50% [EEG+15b]. Otherwise the predicted detection is considered a False Positive. The ground truth objects that the model missed, i.e. the objects with no matching bounding box are counted and represent as False Negatives [EEG+15b]. With the provided values of True Positives, False Positives and False Negatives, the Precision and Recall values are calculated according to Equation 2.7 and Equation 2.8. The evaluation proceeds with calculating the Average Precision (AP) for the particular class.

The Average Precision is expressed as the Precision across all Recall values [EEG+15b]. Essentially, this becomes a single value, summarizing the shape of the so-called *Precision-Recall curve*. Since 2015, the PASCAL VOC Competition [EEG+15b] recommends to calculate this by taking the mean Precision, across a set of 11 different, equally spaced confidence values for the Recall, such that *Recall*$_i$*=[0, 0.1, 0.2, 0.3, ..., 0.9, 1]*. Equation 2.11, provides the mathematical formula for calculating the Average Precision. Thus, the Precision at Recall level i is taken to be the maximum precision measured at a Recall greater or equal *Recall*$_i$ [EEG+15b].

$$AP = \frac{1}{11} \sum_{Recall_{\mathrm{i}}} Precision(Recall_{\mathrm{i}}) \tag{2.11}$$

The mean Average Precision (mAp) is then expressed as the mean of the Average Precision score across *all* classes the model needs to detect.

CHAPTER 3

Related Work

With the massive growth and expansion of the global economic markets, good marketing campaigns and advertisements continue to play a crucial role in keeping the popularity rate high and gaining advantage over competitors. Nowadays, almost every (marketing) company benefits from the widely distributed smartphone and social media usage [IAB17]. This enabled an abundance of advertising possibilities as well as a unique way of reaching the target groups of interest due to the constantly increasing amounts of data social media companies collect on a daily basis [IAB17]. Though online advertising opened the door to a large number of new advertisement opportunities, it did not help the companies understand why specific ads affect people better than others let alone make the advertisement process less challenging [CSL$^+$16]. A poorly chosen billboard or ad location, inappropriate size, design or wording could still affect the company's image and prosperity negatively. Thus, making the right decision as to where, how and for how long to place an advertisement or a billboard are all factors which could likely determine a company's profitability. As a result, any unconventional or innovative approach to reach the desired target group or improve on the traditional advertisement process is highly valuable and sought after.

Luckily, the ongoing revolution in the field of Computer Vision has elicited many research possibilities and enabled great progress in many vital areas. Since the field of Computer Vision experienced the breakthrough of convolutional neural networks with AlexNet [KSH12b] on the task of image classification, convolutional neural networks have not only proven their suitability for other computer vision tasks such as detection, recognition and image segmentation, but also extended their applicability to a larger domain including the area of (online) advertisements and billboards. With the large amounts of available data [DDS$^+$09][EEG$^+$15a][LMB$^+$14], convolutional neural networks can be the answer to many so-far unsolved (or manually executed) problems that the area of advertisement and billboard placement faces. From automatic billboard classification [HDN$^+$18] and billboard frame recognition [ZCHC17] in images and video streams respectively to analysing and understanding the context of an ad [HZZ+17], the application possibilities of convolutional neural networks in the advertisement domain is almost endless. In this section, we provide an overview of the related work and the current state-of-the-art CNN contributions to the area of advertising.

In order to avoid the tedious and undesirable (pre-, mid- or post-) insertion of advertisements in online videos and movies and yet still expand the target audience range, the current focus of advertising lies in a technique known as embedded marketing or product placement [KUE11][LL15][HDN+18]. Embedded marketing involves integrating advertisements directly into the scenes of the video stream, in a way that specific products and brands are purposely inserted within the video settings [KUE11][HDN+18]. While this technique allows an uninterrupted viewing experience, as well as reaching out to specific demographic regions or markets it has mostly been executed manually. That means manually analysing every video frame and investigating the possible frames for ad placement. This triggered Hossari et al. [HDN+18] to propose a solution that automates the otherwise time-consuming and expensive frame identification task. Hossari et al. [HDN+18] developed ADNet, a deep convolutional neural network that specializes in classifying billboard advertisements in video streams. Hence, ADNet has the ability to successfully distinguish a billboard-frame from a non-billboard frame. The architecture of ADNet was inspired by and based on the powerful deep neural network VGG19 [SZ14a]. At its core ADNet contains 16 convolutional layers with a kernel size of 3x3 and a stack of 3 fully connected layers at the end of the network [HDN+18]. The first two fully connected layers contain 1024 channels each and use ReLU as their activation function. The last fully connected layer utilizes the softmax function which outputs the probabilities of the billboard or non-billboard classes. To prevent overfitting, ADNet also employs a dropout layer as a regularization technique with a rate of 0.5. For evaluating the performance of the proposed model, the authors also trained the Inception-v3 [AHYT17] model on the same dataset and parameter configuration and used the achieved results as a benchmark. ADNet achieved 94% classification accuracy and outperformed Inception-v3 which achieved 56% accuracy on the billboard frame classification task [HDN+18].

Similarly to the motivation of Hossari et al., Covell et al. [CBF06] proposed a method for automatic detection of TV advertisements that appear during TV shows and movies. Since television material is redistributed globally it may come as no surprise that the original TV advertisements are replaced with new ads depending on many criteria including: region, broadcast time and season [CBF06]. To properly detect the starting and ending point of a TV commercial Covell et al. developed a three-stages approach which relies on audio and video features. In the first stage, the authors extract the audio features and perform audio-repetition detection. This is achieved by performing acoustic matching on the audio features in order to determine the potential advertisement matches. In the second stage, the candidate matches are verified and validated against an advertisement database with the help of video feature extraction. In the final stage, the candidate matches that passed the audio and the video checks, undergo a so-called endpoint detection (using forced Viterbi [GM99]) in order to correctly locate the advertisement boundaries. With the proposed method Covell et al. [CBF06] achieved 99% precision and 95% recall rates in extracting advertisements from video sequences.

Almgren et al. [AKAL18] proposed a simple convolutional neural network based approach for advertisement classification in newspapers and magazines. In their paper Almgren et al. define their proposed solution as a two-step approach, consisting of a feature extraction step and a classification step. The feature extraction layer in the CNN model includes a convolutional layer with a 5x5 kernel, a ReLU activation function and a max_pooling layer with window size of 2x2. For the classification on the other hand, the CNN model employs a fully connected layer with 10000 neurons. The input of the network is a grayscale image of size 100x100 pixels, while the output is defined by the two target classes: “advertisement” or “non-advertisement”. With their CNN model, the authors achieved 78% accuracy, 57% recall and 87% specificity, outperforming the standard machine learning algorithms such as Random Forest, Multilayer Perceptron (MLP) and Support Vector Machines (SVM).

Bianco et al. [BBMS17] introduced a method for logo recognition based on deep learning. Their method is composed of a recall-oriented logo region proposal and a convolutional neural network trained specifically for logo classification [BBMS17]. Since there is no way of knowing the precise location of logos in an image, for each image the authors generate a set of object proposals. Essentially, an object proposal is a region that is more likely to contain the objects of interest i.e. logos in this context. The object proposals are obtained with the help of the highly recall-oriented Selective Search algorithm [vdSUGS11][USGS13]. Once generated, the object proposals are then cropped to match the input size dimension of the convolutional neural network. Afterwards, the cropped object proposals are contrast normalized and passed to the convolutional neural network model used for logo classification. The architecture of the CNN consists of three convolutional and pooling layers, followed by two fully connected layers and the softmax function used for classification. The activation function used in the convolutional layers is ReLU. The CNN model was trained on an extension of the FlickrLogos-32 [RPLvZ11] dataset (Logos-32plus) which was designed by the authors. The Logos-32plus dataset contains 12312 images and was enlarged using data augmentation techniques. At the end, the dataset contained 32 classes with circa 400 training samples per class. With their best configuration, Bianco et al. achieved a 95.8% accuracy, 98.9% precision, 90.6% recall and 94.6% F1 score.

Analysing and understanding the content of an image or a video is a complex computer vision task. In their paper, Hussain et al. [HZZ$^+$17] proposed the novel problem of automatic advertisement *understanding*. The problem of automatic advertisement understanding goes beyond the problem of object recognition and task of producing sentences about images as it requires a special skill set to determine what, how and why objects are depicted in a certain way in order to correctly understand the meaning of an ad [HZZ$^+$17]. Hussain et al. delivered a new solution for the problems of automatic image as well as video advertvertisment understanding. For the automatic ad image understanding problem they created a dataset containing 65000 images, most of which were labelled (as ad or non-ad) using the ResNet neural network [HZRS16] for classification. Furthermore, each image was manually tagged with additional annotations regarding the topic of the ad, the sentiment it conveys, the strategy and symbolism it uses, as well as answers to two questions (“What should the viewer do?” and “Why should he do it?”) [HZZ+17]. For the automatic video ad understanding problem a separate dataset containing 3000 video ads was created. The videos contained the same label types as the ones used in the image dataset, omitting the symbolic labels, but included answers to some additional questions such as “Is the video funny?” [HZZ+17]. As a solution, the authors developed a neural network model which combines the layers of the LSTM [HS97] and the VGG [SZ14a] network. They are followed by a softmax layer which determines the output of the network. In their study the authors evaluate their neural network on the following tasks: Questions and Answers (regarding ads), Symbolism prediction and Topic and Sentiment Analysis. On the ad image questions and answers (QA) task, their model achieved an accuracy rate of 11.48%. Moreover, on the task of symbolism prediction in ad images their model achieved 15.79% F-measure. The model scored an even lower accuracy on the QA task regarding video ads. For the topic and sentiment analysis in image and video advertisements the authors utilized 152-ResNet [HZRS16] and achieved an accuracy of 60.34% and 27.92% respectively. In their work the authors show that existing methods still do not have the ability to perform the high level QA task when it comes to advertisements. However, their research results on Topic and Sentiment Analysis in advertisements give hope and provide room for further improvement [HZZ+17].

According to Zhang et al. [ZCHC17], one of the primary objectives of video online advertising is to create a relevant ad that meets the interest of the target customers at the right place and time without intrusion of uninterested viewers. Hence, to increase the attractiveness and lower the intrusion of an ad, the authors suggest to make the advertised products relevant to the contents (objects) in the video. For this reason, the authors propose Object Level Video Advertising (OLVA), an optimization framework for target online advertising in video sequences. The underlying idea of the framework is to locate the potential shot for advertising in the video sequence and compute the sentiment of the shot in order to determine the advertisement type [ZCHC17]. The framework itself is composed of the following components: shot segmentation, object detection, optimization-based object selection and ad retrieval. For detecting potential shots in the video the authors utilized a threshold-based method which relies on the Histogram of Oriented Gradient (HOG) features as well as on a color histogram and the Local Binary Patterns (LBP) features to calculate the distance between two adjacent frames. A distance larger than some predefined threshold determines the boundaries of the two shots. For each shot, object detection is performed using the HOG features. The OLVA framework was tested with five object classes: person, car, bottle, dog and bicycles. In the case where the detected object belongs to the person class, gender recognition is performed using a convolutional neural network with 4 convolutional layers and 2 fully connected layers [ZCHC17]. The authors formulate the decision as to which shot to add which ads as an optimization problem, which is dependent on the object class. The potential frames for ad-insertion are usually the ones containing fewer objects. To solve the optimization problem Zhang et al. propose a Heuristic Algorithm as well as a Genetic Algorithm which solves the global optimization problem. In the end, the ads are placed as hyperlinks in the right lower corner of the video. The achieved results from the conducted experiment show that the OLVA framework can improve the attractiveness of the advertisements and the level of comfort in watching the videos [ZCHC17].

As witnessed by the previous paper contributions, presenting relevant advertisements through the content i.e. frames of the video is not an easy task. Youtube as a world-leading application for video content and advertising desires to trick the users into spending more time on the application and watch video content i.e. more advertisements [CAS16]. The primary method that achieves this is the personalized recommendation of videos presented to the target user, prompting him to use the application longer. Covington et al. [CAS16] designed a system architecture based on deep neural networks that computes Youtube recommendations. The system is composed of two deep neural networks: a candidate generation deep neural network and a candidate ranking deep neural network [CAS16]. The task of the candidate generation deep neural network is to propose personalized candidate videos based on the viewing history of the user [CAS16]. This is a simple feed forward neural network composed of fully connected layers using ReLU as an activation function. The candidate ranking deep neural network is designed with the aim to rank the proposed videos which were generated by the candidate generation neural network [CAS16]. Furthermore, the structure of the candidate ranking deep neural network is similar to the structure of the candidate generation deep neural network. The only difference is that the candidate generation CNN is used for classification and the task of the candidate ranking CNN is to perform logistic regression. The final ranking score of a video depends on the viewing time of the user regarding a specific video. As a result, another model for predicting the viewing time of the user with respect to a specific video is defined with the goal to perform a weighted regression task [CAS16]. The best configuration for predicting the expected watch time of the user was obtained by making the model not only deeper but also wider in size. It included a 1024-wide ReLU which was followed by a 512-wide ReLU and a 256-wide ReLU. With this configuration, Covington et al. achieved a 34.6% “weighted, per user loss” which represents the total amount of *mispredicted watch time per user*. These results prove that increasing the width of the hidden layers can also have a positive impact on the performance. With the proposed system, Covington et al. [CAS16] achieved an outstanding and effective performance for recommendation of videos on the Youtube application.

CHAPTER 4

Methodology

4.1 System design

In this master thesis a system for automatic billboard detection is designed. Specifically, the main focus of the designed system lies in automatically detecting billboards that are located in metro stations. The proposed system design represents a deep learning based approach and is composed of two core components that we refer to as the client (side) and the server (side) of the system (see Figure 4.1). Both the client side and the server side are responsible for and carry out subtasks equally important for achieving the ultimate goal of automatic billboard detection.

The client is designed as a lightweight Android application with the goal of recording and providing video sequences to the server. For this reason, the client is developed with three main functionalities: record, play and share video as depicted by Figure 4.2. The feature of sharing of videos to a Cloud platform is purposefully integrated in the Android

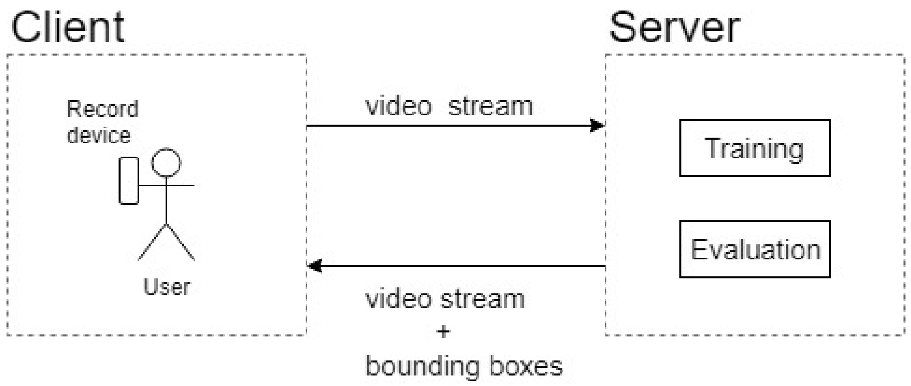


Figure 4.1: Proposed client server architecture

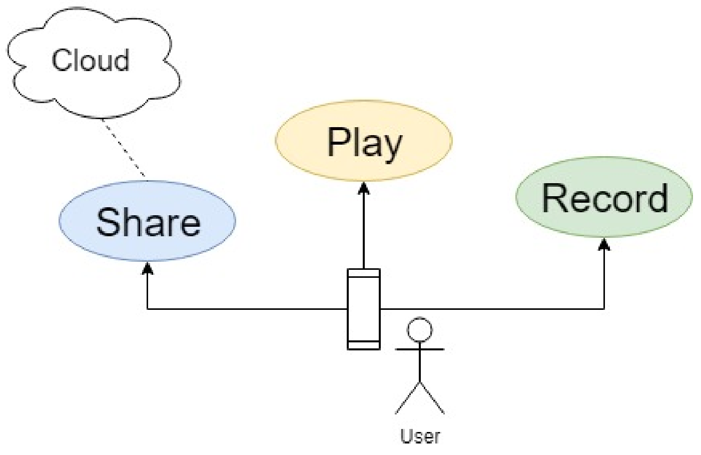


Figure 4.2: Features of the client side

application in order to enable memory preservation on the mobile device. Moreover the sharing functionality also allows for an easier access to the recorded videos which can be synchronized with the server. The main idea behind the Android application is that the user employs it to record a video which is then transferred to the server for automatic billboard detection. Afterwards, the user receives the same video sequence with the detected billboards (if any) in bounding boxes, together with a confidence score of the detection. Hence, the video streams recorded and provided by the client side serve only for testing and evaluation purposes.

The server, on the other hand, is responsible for the automatic billboard detection. It is represented by a self-designed deep convolutional neural network model, which is called *StefanNet*. *StefanNet* is a fully convolutional neural network (CNN) which means its architecture is composed of convolutional layers only, not including any fully connected layers. The model is based and inspired by the architecture of the state-of-the-art deep convolutional neural network model VGG [SZ14a]. The input of the model is a 2D image of size 300x300 pixels. The output of the model is a list of (possibly multiple) bounding boxes with confidence scores for the given input image. The bounding box is defined as a list containing the following information: *classId*, *confidence score*, *xmin*, *ymin*, *xmax* and *ymax*. The bounding box is determined by its four coordinates (xmin, ymin, xmax, ymax) which form a rectangle.

StefanNet is exhaustively trained on a dataset containing billboard images called the *BillboardDataset* which will be discussed in detail in Section 4.2. Furthermore, compression in the form of quantization is applied to *StefanNet* in order to reduce its size and improve its performance. Quantization [HMD15] represents an explorative performance optimization method that enables faster training of *StefanNet* and a significant increase in its inference rate. This is achieved by transforming the data type of the layers from *float32* to *float16*, resulting in a reduction of the network size. The overall performance of *StefanNet* is evaluated on a separate dataset and compared to the performance of four

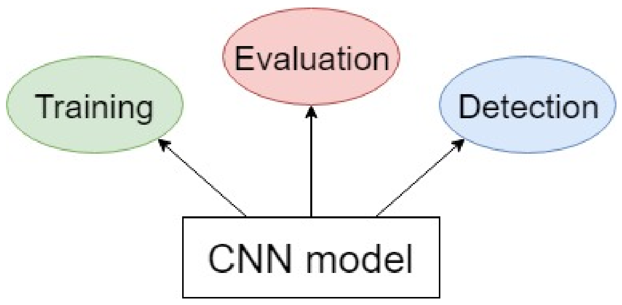


Figure 4.3: Functionalities of the server side

other state-of-the-art deep neural networks, which were trained and evaluated on the same datasets as *StefanNet*. Hence, *StefanNet* is designed to successfully and efficiently detect and localize billboards in videos provided by the client. In Figure 4.3 the three main tasks executed by the CNN model *StefanNet* (server side) are shown.

The flowchart depicted in Figure 4.4 demonstrates the interaction between the client and the server side as well as the actual order of the execution of actions i.e. tasks. The user records a video sequence using the Android application. Afterwards the user has the option to either share the video content to a Cloud platform or to store it on the mobile device. From the storage the video is then passed to the server. Once the server obtains the video sequence from the client, it loads the *StefanNet* model which processes the video sequence on a frame-wise basis. For each frame of the video, the CNN model tries to localize and detect any present billboards in the given frame. The next step is to filter the obtained detections. Detections with a confidence score lower than 50% are considered unreliable and are therefore rejected. On the other hand, detections with a confidence score greater or equal to 50% are taken into consideration and thus imported into the frame as bounding boxes. At the end, the same video frame together with the imported detections (with confidence scores) of billboards is presented to the user. The iterative process continues for every frame of the video until the processing of the whole video is finished.

4.2 Data engineering

The first step towards building an efficient and accurate deep neural network model is collecting a sufficient amount of problem-specific data. As a result, the process of data engineering plays a crucial role not only in the scope of this master thesis, but in the field of Deep Learning in general. Depending on the problem domain, collecting the right amount of data could either be trivial or extremely problematic. Due to the lack of billboard data and pure billboard image datasets online, the images used for the training process in this master thesis are self-collected. *StefanNet* is trained on the *BillboardDataset* which consists of billboard images gathered from the metro stations of all metro lines in Vienna, Austria. Since the process of collecting the data was very time-

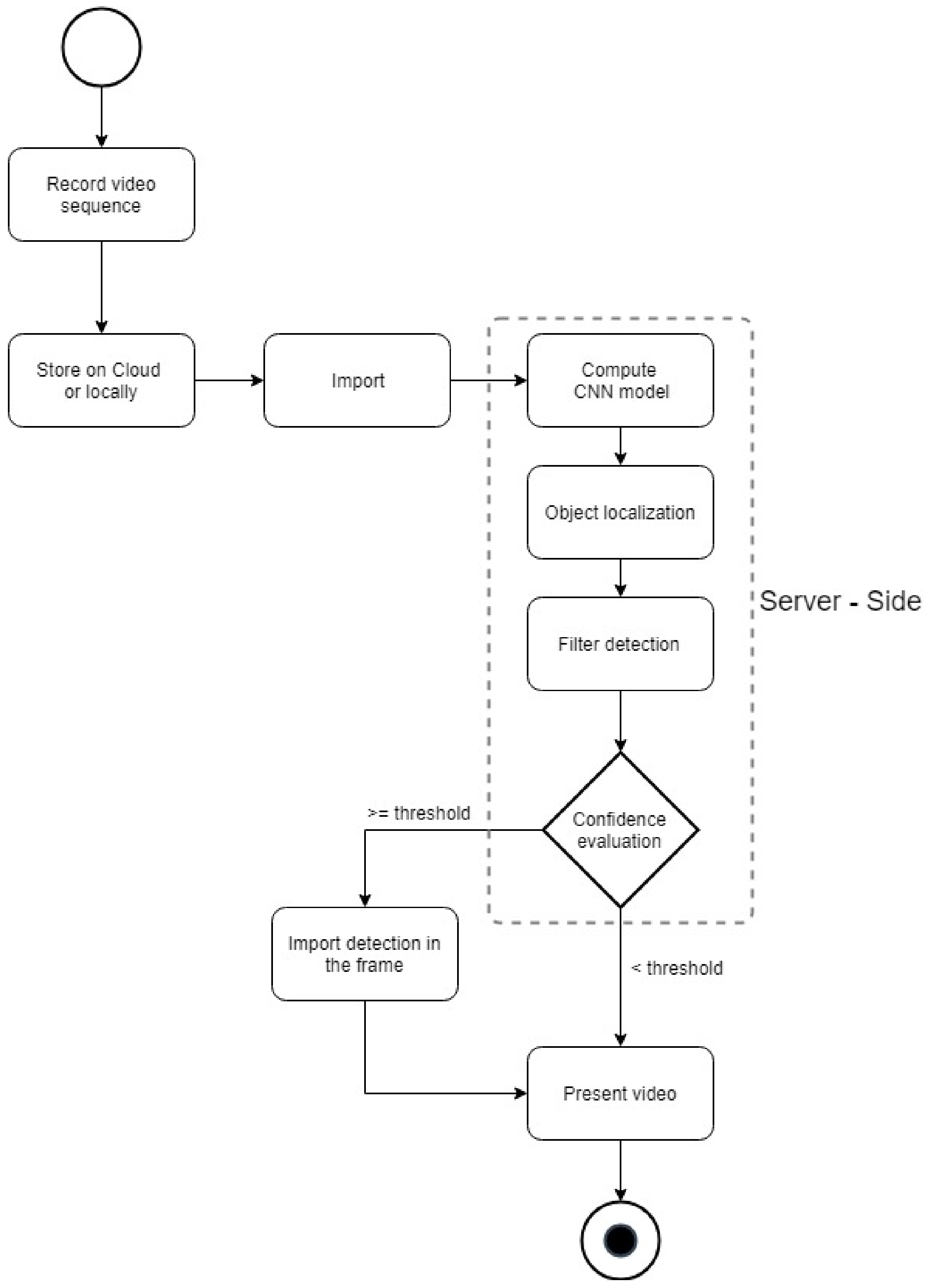


Figure 4.4: Flowchart diagram of the designed system

Class	#train	#test	#infront	#side
billboard	2340	1002	350	350

Table 4.1: Separation of *BillboardDataset* for training, testing and validation (infront, side) in number of images

consuming, the dataset was collected batch-wise in a certain period of time. Diversity of the data is ensured by capturing images under different lightning conditions as well as taking images from different angles. Furthermore, the images were captured not only indoor, but also in outdoor settings. Figure 4.5 demonstrates a couple of samples of the *BillboardDataset*. While the images 4.5b, 4.5e and 4.5f represent samples taken in outdoor settings, the images 4.5a, 4.5c and 4.5d represent samples captured in indoor settings. Important to note is that the background of the images depends on the metro station they belong to.

The image samples from the *BillboardDataset* are categorized into two main groups. There exist image samples taken with a front-on view of the billboard such as the sample images in 4.5d and 4.5f in Figure 4.5. Additionally, there are image samples captured from a side angle i.e. images with a profile view or side view of the billboard (see images 4.5a, 4.5b, 4.5c, 4.5e in Figure 4.5). Moreover, in both of the categories, there exist images depicting a single billboard as well as images that show multiple billboard objects. The sizes of the image samples in the *BillboardDataset* are: 3024x4032, 4128x2322 and 4032x3024 pixels. The whole *BillboardDataset* contains 4042 image samples of billboards.

The *BillboardDataset* is carefully split into three smaller datasets which are used for training, testing and validation of the *StefanNet* model. The pie chart in Figure 4.6 and Table 4.1 depict the separation of the *BillboardDataset* in percentage and number of images respectively. The largest dataset is used for training of *StefanNet* and contains 2340 image samples (57% of the *BillboardDataset*). The dataset used for testing of *StefanNet* contains 1002 image samples and represents 25% of the *BillboardDataset*. The smallest dataset is used for validation of the model and contains 700 images samples (18% of the *BillboardDataset*). Fifty percent of the images used in the validation dataset (350 image samples) belong to the group of images with a front-on view of a billboard object, whereas the other half of the validation images have a profile or side view of the billboard objects.

Since developing a high performance model depends on correctly labeled data, the task of labeling the collected data is an important part of the data engineering process. The labeling of the image samples from the *BillboardDataset* is executed with the help of the Labelme application [Lab16]. The labeling of a billboard object in an image captured from a front-on angle is an easy task (see Figure 4.7). This is due to the fact that the rectangle from the label matches the billboard object shape perfectly. In contrast, labeling the images that have a side or profile view of the billboard object correctly posed a certain challenge. Since the shape of the billboard is no longer a perfect rectangle, it

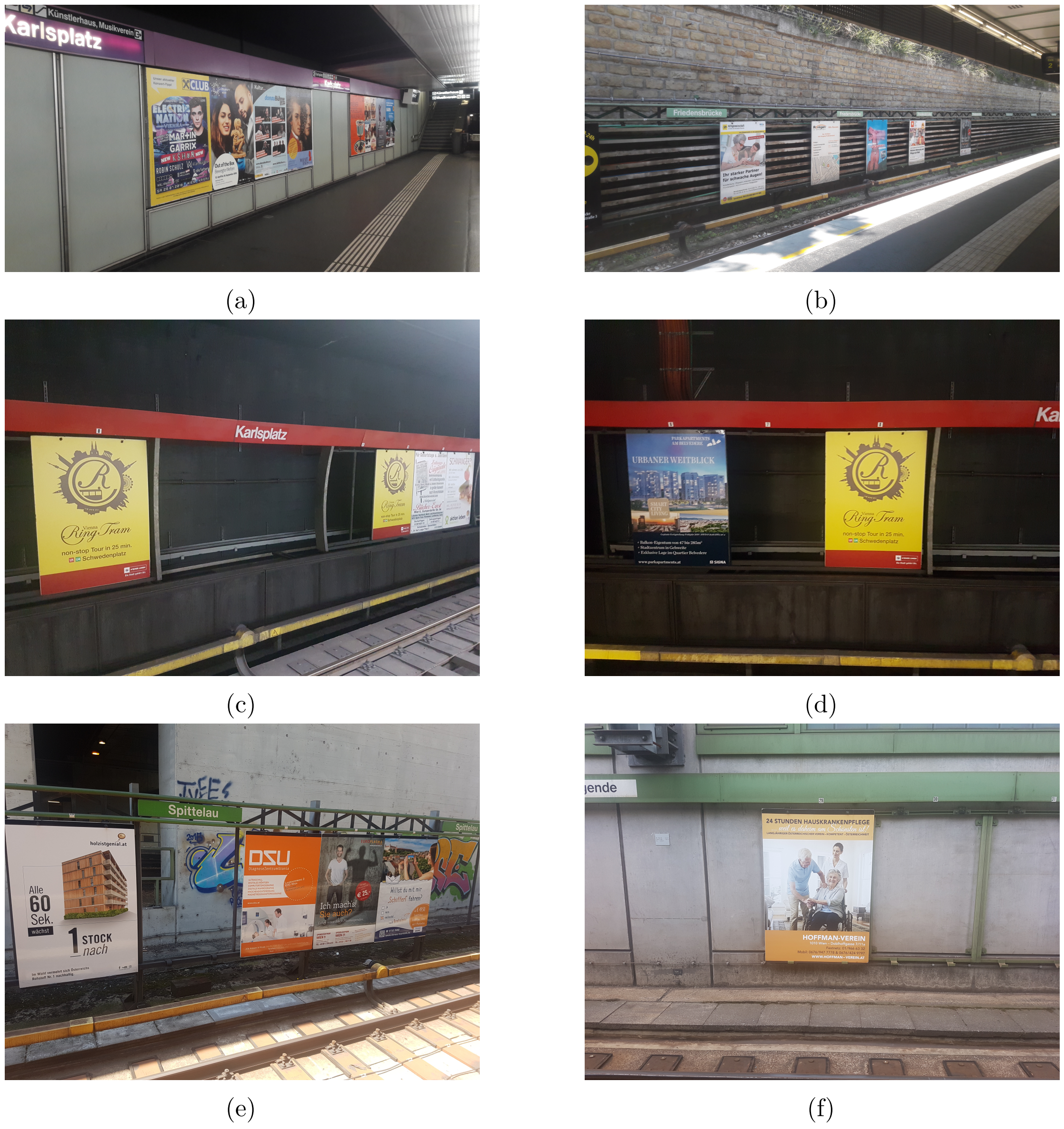


Figure 4.5: Samples from the *BillboardDataset*

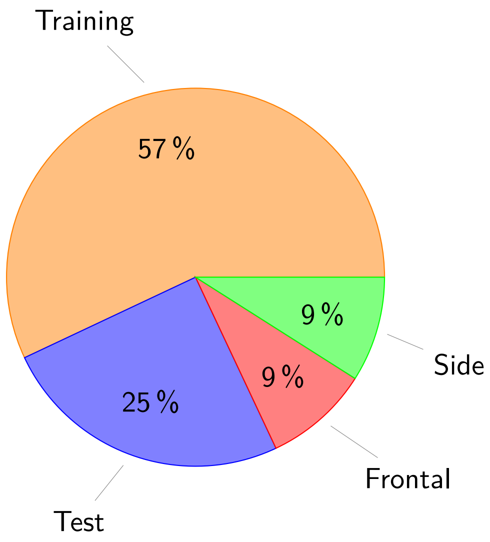
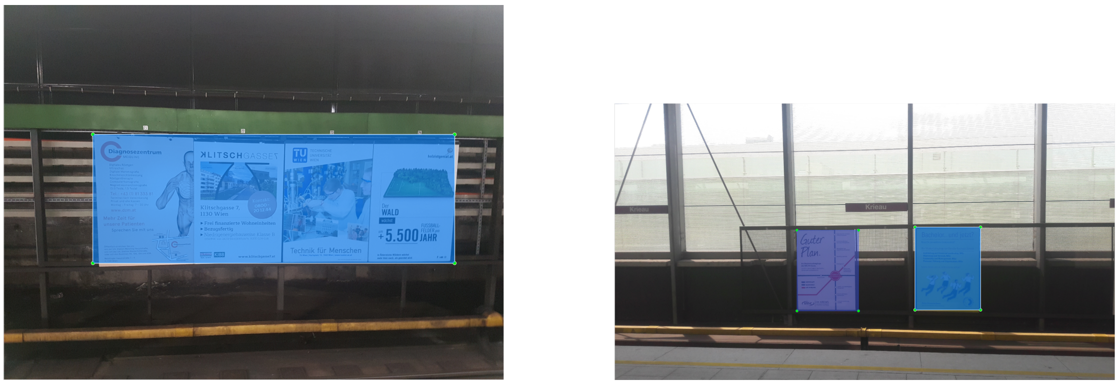


Figure 4.6: Separation of *BillboardDataset* for training, testing and validation, in percent

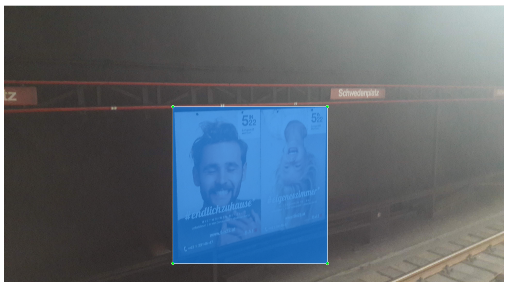


(a) (b)

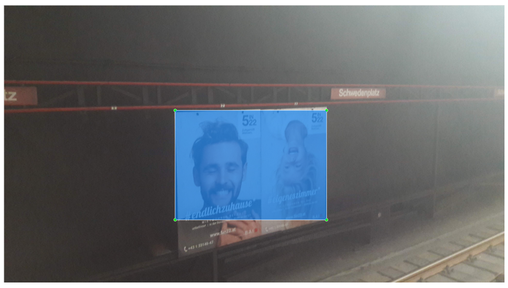
Figure 4.7: Labeling images with frontal view of billboard objects

does not correctly match label shape. Hence we are left with two scenarios: the label shape includes parts of the background or the label shape does not cover the whole billboard image completely (see Figure 4.8). The question that remained was: "Which of these two scenarios is the preferable one?". In this case, we chose the scenario in which the label shape includes parts of the background as shown in Figure 4.9, because the achieved results with this method were particularly high (for more details on the achieved results see Chapter 6). Lastly, the labeling of billboard objects with little or no distance between them was also a delicate task.

The label of an image sample is saved in an XML file and contains information about the image sample such as: storage location of the image sample, size in pixels, number

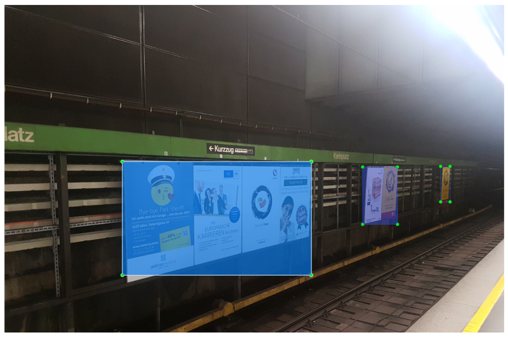


(a)

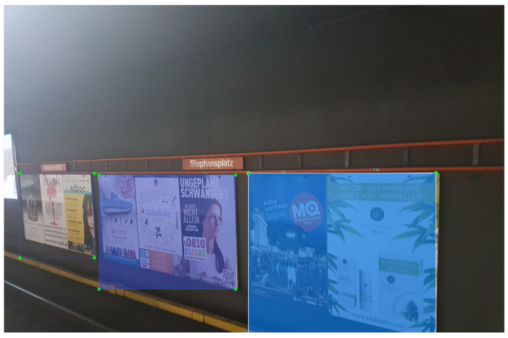


(b)

Figure 4.8: The challenges of labeling images with a side view of billboard objects. The label in the image on the left contains parts of the background, wheres the label on the image on the right does not cover the whole billboard object.



(a)



(b)

Figure 4.9: The chosen method for labeling images with a side view of a billboard object (label includes parts of the background).

of labeled billboard objects, label name, i.e. the class name and the coordinates of the (possibly multiple) labeled billboard objects. The coordinates of a labeled billboard object form a rectangle and are represented in the following format: [xmin, ymin, xmax, ymax]. From the XML file a simple text file is created. The text file is named after the image sample it corresponds to and contains only the coordinates of the labeled billboard objects for the given image sample and their class. Eventually, the image sample together with the labels of the image are used for training the CNN model *StefanNet*. The process of training *StefanNet* is explained in Section 5.4.

	#train	#test	#infront	#side
number of samples	3300	1002	350	350
% of *BillboardDatasetPlus*	66%	20%	7%	7%

Table 4.2: Separation of *BillboardDatasetPlus* for training, testing and validation (frontal, side) in number of images

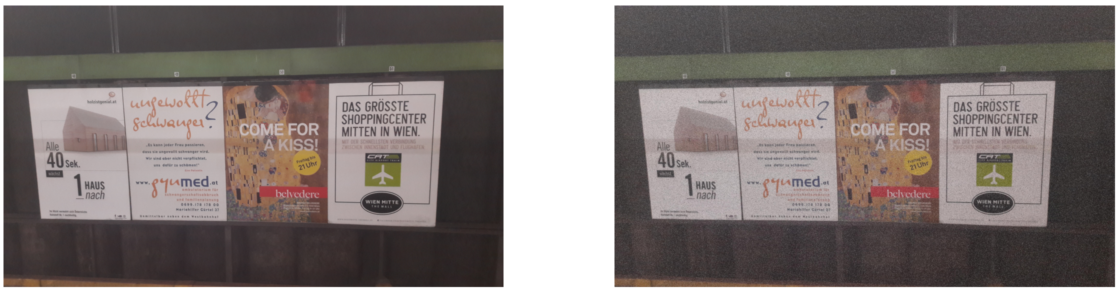
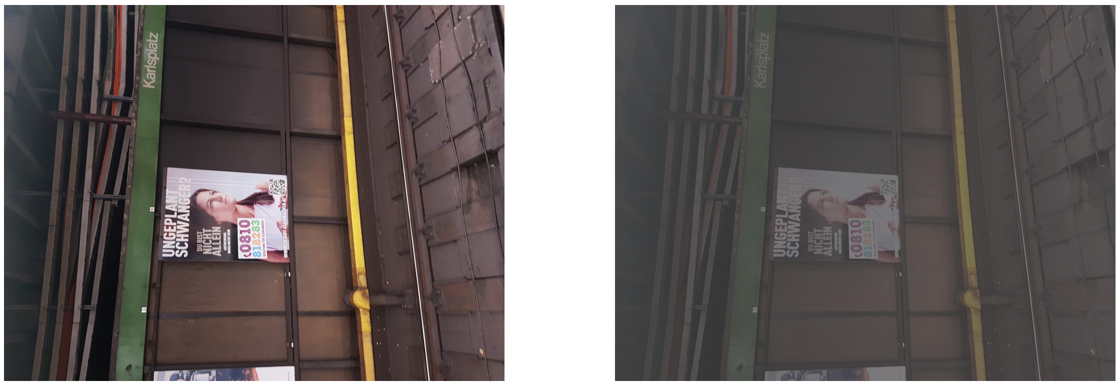

(a) normal image (b) augmented image

Figure 4.10: The effect of adding salt&pepper noise and a Gaussian blur to an input image.

4.3 Data augmentation

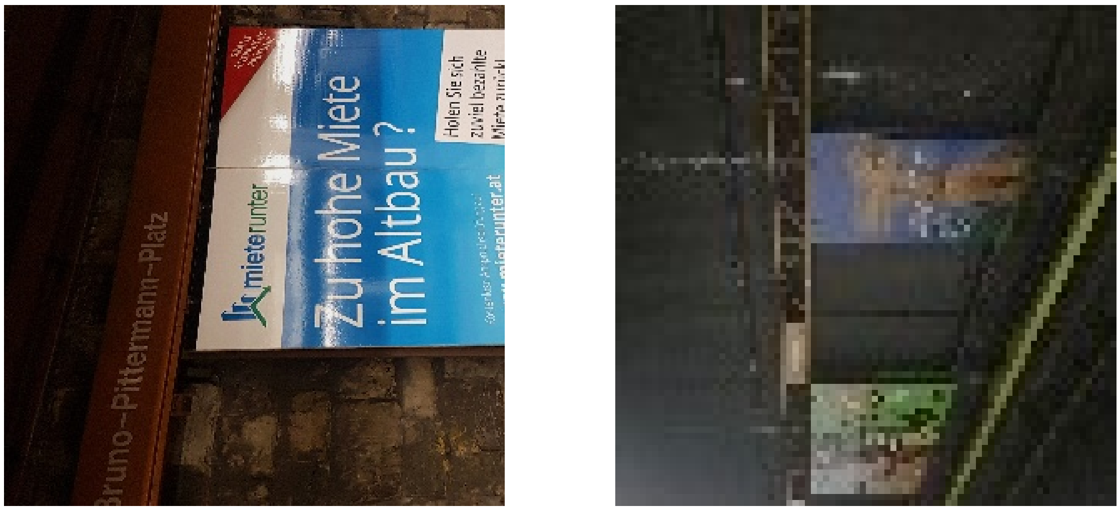
As stated in Section 4.2, the data used during the training process of a CNN model are of vital importance, as they have a tremendous effect on the performance capabilities of the resulting convolutional neural network. The process of gathering and collecting data can be very time consuming and expensive. In the cases where the data is limited in size or difficult to obtain, the method of data augmentation proves to be useful. Data augmentation takes an image dataset of limited size and artificially augments it by performing various transformations on the images already provided [KSH12a]. Some of the basic transformations which can be performed on images include: scaling, cropping, rotation, illumination, contrast or a combination of any of them [KSH12a]. Generally, the techniques for data augmentation can be applied either during the data preprocessing stage or during the training process in which they are integrated on the fly [KSH12a].

Since during the process of creating the *BillboardDataset*, we also faced difficulties in obtaining a sufficient amount of data, data augmentation techniques were employed to artificially increase the number of billboard image samples. The original *BillboardDataset*, as already explained in Section 4.2, contains 4042 samples. With the help of the data augmentation techniques an extension of the *BillboardDataset*, called *BillboardDatasetPlus* was created. In contrast to the *BillboardDataset*, the *BillboardDatasetPlus* is circa 25% larger and contains a total of 5002 billboard image samples. Similarly to the *BillboardDataset*, the *BillboardDatasetPlus* is also split into three smaller datasets for training, testing and validation purposes. Table 4.2 demonstrates the number of billboard image samples used for each of the three datasets. Important to note is that the *StefanNet*



(a) normal image (b) augmented image

Figure 4.11: An example of reducing the contrast of an input image.



(a) crop + rotate (b) scale + rotate

Figure 4.12: Examples of data augmentation techniques applied on the fly.

model was trained and tested on both: the *BillboardDataset* as well as the augmented version *BillboardDatasetPlus*.

The *BillboardDatasetPlus* was created using two data augmentation techniques which were applied during the data preprocessing stage. The first data augmentation method which was applied represents a two-step approach and includes adding noise to the images ("*salt & pepper*") and then smoothing the images with the help of *Gaussian blurring*. The "*salt & pepper*" noise is defined as sudden occurrences of black and white pixels throughout the image. Moreover, the kernel size of the *GaussianBlur* filter is 5x5. The effect of this method can be seen in Figure 4.10, which depicts the pre and post-application effects on a given input image. The second data augmentation method employed adjusts the contrast of the provided input image. This means reducing the brightness of the input image by a factor of 30%. An example of the application of reducing the contrast of the input image is shown by Figure 4.11. Furthermore, data augmentation techniques were also integrated on the fly, during the training process of *StefanNet*. The data augmentation methods which were integrated on the fly include rotation, cropping and scaling the input images. An example of these techniques is shown in Figure 4.12. Based on the collected data as well as the described processes and techniques, the network *StefanNet* explained in Chapter 5, was developed and tested.

CHAPTER 5

Implementation

5.1 Introduction

In this chapter the implementation of the client server application designed for automatic billboard detection is presented and explained in detail. As discussed in Chapter 4, the client is responsible for gathering and providing videos to the server, which in turn performs automatic billboard detection on the provided video sequences. We begin this chapter by providing an overview of the experimental test setup including the technical details of the hardware and software components utilized. Afterwards, we focus on the architectural design as well as the implementation details of the server side which is primarily represented by *StefanNet*, a deep convolutional neural network implemented in the MXNet framework. Furthermore, we provide a description of the most important libraries and frameworks employed (including MXNet) and discuss their advantages. The CUDA platform together with the cuDNN library are used for acceleration of the GPU-based computing. The GPU is the main hardware component used for training *StefanNet*. We proceed by reviewing the implementation procedure of the compression-based quantization technique and highlight the effect this method has on deep neural network models like *StefanNet*. The end of this chapter shifts its attention to the client side of the system and discusses the implementation details of its use case. The client side is represented as a lightweight Android application implemented in Android Studio.

5.2 Test setup

As the modern graphical processing units (GPU) have given rise and essentially enabled the evolution of the field of deep learning, they are an inevitable hardware component when it comes to deep neural network models [CWHH14]. This is mainly due to the massive number of matrix multiplications as well as weight adjustments that need to be carried out during the process of training a neural network. As a result, in order to allow an execution of the training in a timely manner, it is crucial that some of these (sub-) procedures are parallelized. This is enabled by the GPU as it contains a very high number of threads [OHL+08]. The GPU used in our setup is the Nvidia Geforce GTX 1080 which contains 8GB of memory. The core API for utilizing the GPU is CUDA, which is also developed by Nvidia. In this master thesis, the CUDA version used is 9.0. Furthermore, the GPU-accelerated cuDNN library, which is specialized in training deep neural networks on GPU is employed. The version of cuDNN library used in our experimental test setup is 7.2.1. During the training process of the *StefanNet* model, the GPU load was very high and the batch size configuration is 8.

The CPU configuration employed in our test setup is Intel® Core^{TM} i7-7820HK @2.90GHz and can be overclocked to 3.90GHz. During the training process, the data is stored on an SSD with a 512GB memory in order to ensure fast and easy data access. All additional data together with the trained models are stored on the HDD with 1TB storage available. The RAM memory in the system is 32GB and the operating system used is Windows 10 64Bit.

For the implementation, we utilize Anaconda as a python distribution. From Anaconda, the python language as well as many data science packages (libraries) are easily installed. The version of Anaconda used in this master thesis is 4.5.10 and the python version is 2.7.16. The fundamental software employed in this master thesis is the MXNet framework and the OpenCV library. MXNet is a framework that supports the python programming language [CLL+15]. Furthermore, it is used for designing, training and testing deep neural networks. The MXNet version used in our setup is 1.2.0. Further details regarding MXNet can be found in Section 5.4. OpenCV, on the other hand, is the goto library when dealing with computer vision problems [CAP+12]. Additionally, OpenCV provides computationally efficient functions for real-time computer vision. The version of OpenCV employed in our setup is 3.4.2. Finally, the Android application is implemented in Android Studio with version 3.0.1. Table 5.1a and Table 5.1b summarize the most important hardware and software (frameworks and libraries) used for the implementation of the client server application for automatic billboard detection.

5.3 Architecture of StefanNet

Finding an optimal structure and architectural design of deep convolutional neural networks is still an ongoing research topic (see [SZ14b], [HZRS15], [HZC+17]). While the latest state-of-the-art neural network architectures might perform well on a single problem domain, they fail to achieve their top performance in the case of a slightly different or even more specific problem domains. Sadly, as it turns out, there does not exist a "one size, fits all" deep neural network architecture that would be a solution to all of the Computer Vision related problem domains. Depending on the complexity of the problem, the availability of data as well as the main task that needs to be executed, the architecture of the neural network varies.

In this master thesis, for the task of automatic billboard detection in video sequences we

Software	Version
Windows	10 64bit
Anaconda	4.5.10
Python	2.7.15
CUDA	9.0
CuDNN	7.2.1
MXNet	1.2.0
OpenCV	3.4.2
Android Studio	3.0.1

(a)

Hardware	Configuration
RAM	32GB
HDD	1TB
SSD	512
CPU	Intel® Core^{TM} i7-7820HK @2.90GHz
GPU	Nvidia Geforce GTX 1080 8GB

(b)

Table 5.1: Software and Hardware specification

propose a self-defined deep neural network model called *StefanNet*. The architecture of *StefanNet* that we present is a result of the multiple experimental architectural decisions that were carried out in the implementation phase. During the creation process of *StefanNet*, we experimented with different structures, number of convolutional layers as well as number of convolutional filters in each layer and investigated the impact of the regularization layers and techniques. In order to find the best configuration of *StefanNet* and enable it to achieve top results, we also explored and considered the effect each type of layer has on the performance. As a result, *StefanNet* is a fully convolutional deep neural network i.e. it is composed of convolutional layers only and does not contain a single fully connected layer. The feature extractor of *StefanNet* is inspired by the state-of-the-art deep convolutional network VGG [SZ14b] as it uses the same kernel size 3x3. Additionally, *StefanNet* employs SDD, as a real-time object detector. The *StefanNet* neural network model contains 23 convolutional layers, 5 pooling layers of type max_pool and one global pooling layer. Furthermore, *StefanNet* utilizes Batch Normalization layers and employs ReLU as its activation function. Lastly, the output layer of *StefanNet* is characterized by a softmax function which provides the probability of the billboard class together with the coordinates of the bounding boxes of the detected billboards. Figure 5.1 represents the architecture of the feature extractor of *StefanNet*.

5.3.1 Types of layers

In the following, we provide an overview over the layer types of *StefanNet* and discuss the configuration of the hyperparameters of each layer in detail.

Input layer. The input layer of *StefanNet* is represented by a 2D input image of size 300x300 pixels. In addition to the input image, the label of the image (see Section 4.2 for more details) containing the bounding boxes of the billboards in the image, is also provided to the network.

Convolutional layer. As stated earlier, *StefanNet* contains 23 convolutional layers.

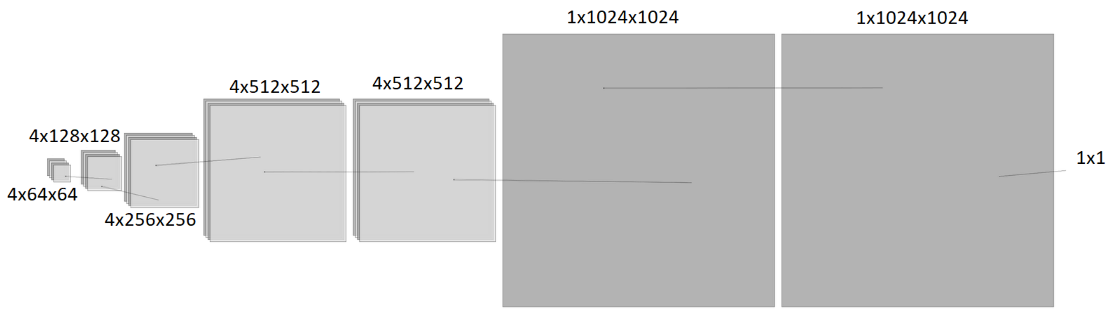


Figure 5.1: Feature extractor of *StefanNet*

The first 20 convolutional layers share the same values for the hyperparameters: *kernel size*, *stride* and *padding*. The *kernel size* used in each of these layers is set to 3x3. In these layers we also use zero-padding of size 1. The *stride* hyperparameter is equal for all 23 layers and has a size of 1. The *depth* hyperparameter (number of convolutional kernels per layer), however, varies between these 20 convolutional layers. The values used for the *depth* hyperparameter in the first 20 convolutional layers are: 64, 128, 256 and 512. In contrast to the first 20 convolutional layers, the last three convolutional layers are configured differently. In the last three convolutional layers we use 1024, 1024 and 1 as number of convolutional kernels (depth values) respectively. The kernel size employed for these layers is 3x3, 1x1 and 1x1. Furthermore, we use zero-padding of size 6, 0, and 0. The reason for this clear separation between the convolutional layers is reviewed and explained in detail in Section 5.3.2.

Batch Normalization layer. The batch normalization layer is utilized to perform normalization of the data by mean and variance on each batch. Furthermore, the batch normalization layers are configured to apply gamma scaling on the input data. Generally, we include the batch normalization layer right after the convolutional layer and before the activation function layer in order to maximize the performance gain of *StefanNet.*

Pooling layer. *StefanNet* contains 5 pooling layers of type max_pool and 1 pooling layer of type global pool. For the first four pooling layers the hyperparameter *kernel size* is 2x2 and the *stride* is set to the value of 2. The fifth pooling layer has a *kernel size* 3x3 and *stride* of size 1. The last pooling layer has *kernel size* 7x7 and the type of this layer is global pool.

Dropout layer. *StefanNet* contains 7 dropout layers with the *drop coefficient* configured to 0.2. This means that every time a sample of an image is passed through the network during training, 20% of the connections are dropped. Overall, the dropout layer helps to reduce the overfitting of *StefanNet* and to decrease the amount of time necessary for its training.

Output layer. The output of *StefanNet* is represented by a softmax function that provides the final output of the network. Essentially, the output contains the percentage

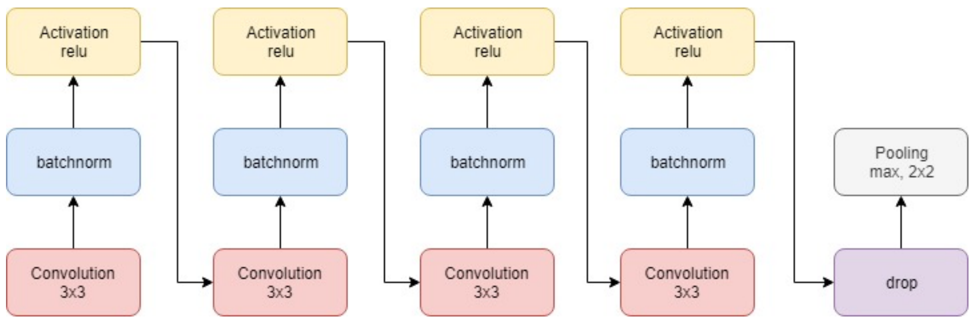


Figure 5.2: Block of *StefanNet*

of confidence of the billboard class and the four coordinates of the bounding boxes of the detected billboard in the given input image.

5.3.2 Building blocks

The number of blocks that build up the structure of a network is an important architectural decision that greatly impacts the overall performance of the network. The whole *StefanNet* network is constituted of a total of five blocks. The fundamental component of a *StefanNet* Block is represented by a so-called CBA (Convolution - Batch Normalization - Activation) structure. After conducting several experiments, it has been concluded that four CBA structures per *StefanNet* block provide the optimal performance of the network. Hence a block of *StefanNet* is composed of four CBA structures plus an additional pooling and a dropout layer.

Figure 5.2 represents the structure of a characteristic single block used in the *StefanNet* network. Each block of *StefanNet* consists of four CBA structures consisting of a convolutional layer with kernel of size 3x3 and padding of size 1x1 followed by a batch normalization layer and a ReLU activation function layer. At the end of each *StefanNet* block there is a dropout layer with drop coefficient of 20% and a pooling layer of type max_pool with both kernel and stride of size 2x2. All four convolutional layers that belong to the same *StefanNet* block have the same number of convolutional filters. Since *StefanNet* contains five blocks, the number of convolutional filters used for each block in the order of appearance in the network is 64, 128, 256, 512 and 512. After the five blocks, there are two more CBA structures in the network. The number of filters used in the convolutional layers of these two CBA structures is 1024. The first CBA structure has a convolutional layer with a kernel of size 3x3 and a padding of size 6x6. The second CBA structure, on the other hand, has a convolutional layer with kernel of size 1x1 and padding of size 0x0. Each CBA structure is followed by a dropout layer with a drop coefficient of 20%. At the very end of the network there is a global pooling layer with a kernel of size 7x7 and a convolutional layer with a 1x1 kernel in which the number of convolutional filters is equal to the number of target classes. Lastly, the output layer is represented by a softmax function which transforms all of the network activations into a

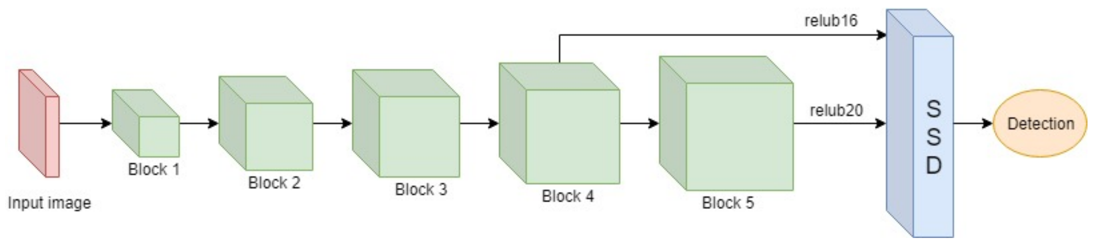


Figure 5.3: Integration of *SSD* in *StefanNet*

series of probabilities.

5.3.3 Object detector

StefanNet employs SSD as object detector. SSD is integrated in *StefanNet* in the 16th convolutional layer (block 4) called "relub16" and in the 20th convolutional layer (block 5) called "relub20". Figure 5.3 depicts the integration of SSD in *StefanNet*. From the convolutional layer "relub20" four stacks of convolutional layers are created. The newly created convolutional layer stacks represent the building blocks of the SSD detection module. Each stack consists of two convolutional layers with kernel size 1x1 and 3x3 respectively. The number of filters in each of these eight layers (four stacks with 2 layers per stack) is 256, 512, 128, 256, 128, 256, 128, 256. The names of these layers correspond to the following naming pattern: *multi_feat_x_conv_3x3_anchors* (where x depicts the number of the layer). Generally, these layers are used for generating the anchor boxes and locating the appearance of the bounding box in the input image. Furthermore, from the convolutional layer "relub16" two more convolutional layers with kernels of size 1x1 and 3x3 are created. These two layers help to improve the class prediction capabilities of the network. Essentially, the newly generated convolutional layers from relub20 and relub16 together with the number of classes serve as input to the integrated module for SSD detection. The output of the SSD detection module is then a list of outputs containing the localization regression prediction, the classification prediction and the generated anchor boxes. Further details about the process of object detection in SSD can be found in Section 2.4.5.

In this section we provided an overview of the proposed deep neural network architecture of *StefanNet*. Table 5.2 presents all layers of *StefanNet* along with their layer-specific configuration parameters. Table 5.3, on the other hand, summarizes the most important characteristics of *StefanNet*. In the next section, we proceed with elaborating on the training process of *StefanNet* as well as on the frameworks and libraries that were used in the training phase.

No.	Name	Type	Pad	Stride	Filter shape	Input size
1.	Conv1	conv	1	1	3x3x3x64	300x300x3
2.	Conv2	conv	1	1	3x3x64x64	300x300x64
3.	Conv3	conv	1	1	3x3x64x64	300x300x64
4.	Conv4	conv	1	1	3x3x64x64	300x300x64
5.	Drop1	dropout			coefficient = 0.2	
6.	Pool1	max_pool	0	2	Pool 2x2	300x300x64
7.	Conv5	conv	1	1	3x3x64x128	150x150x64
8.	Conv6	conv	1	1	3x3x128x128	150x150x128
9.	Conv7	conv	1	1	3x3x128x128	150x150x128
10.	Conv8	conv	1	1	3x3x128x128	150x150x128
11.	Drop2	dropout			coefficient = 0.2	
12.	Pool2	max_pool	0	2	Pool 2x2	150x150x128
13.	Conv9	conv	1	1	3x3x128x256	75x75x128
14.	Conv10	conv	1	1	3x3x256x256	75x75x256
15.	Conv11	conv	1	1	3x3x256x256	75x75x256
16.	Conv12	conv	1	1	3x3x256x256	75x75x256
17.	Drop3	dropout			coefficient = 0.2	
18.	Pool3	max_pool	0	2	Pool 2x2	75x75x256
19.	Conv13	conv	1	1	3x3x256x512	37x37x256
20.	Conv14	conv	1	1	3x3x512x512	37x37x512
21.	Conv15	conv	1	1	3x3x512x512	37x37x512
22.	Conv16	conv	1	1	3x3x512x512	37x37x512
23.	Drop4	dropout			coefficient = 0.2	
24.	Pool4	max_pool	0	2	Pool 2x2	37x37x512
25.	Conv17	conv	1	1	3x3x512x512	18x18x512
26.	Conv18	conv	1	1	3x3x512x512	18x18x512
27.	Conv19	conv	1	1	3x3x512x512	18x18x512
28.	Conv20	conv	1	1	3x3x512x512	18x18x512
29.	Drop5	dropout			coefficient = 0.2	
30.	Pool5	max_pool	0	2	Pool 3x3	18x18x512
31.	Conv21	conv	6	1	3x3x512x1024	16x16x512
32.	Drop6	dropout			coefficient = 0.2	
33.	Conv22	conv	0	1	1x1x1024x1024	16x16x1024
34.	Drop6	dropout			coefficient = 0.2	
35.	Global_pool	avg_pool			Pool 7x7	16x16x1024
36.	Conv23	conv	1	1	1x1x1024x1	12x12x1024
37.	Flatten1	flatten				12x12x1
38.	Softmax	softmax			classifier	1x1

Table 5.2: Layers of *StefanNet*

Input size	300x300
No. convolutional layers	23
No. max pooling layers	5
No. global pooling layers	1
Feature extractor	StefanNet
Object detector	SSD

Table 5.3: Overview of *StefanNet*

5.4 MXNet

MXNet is an efficient open source deep learning framework specialized in training and deploying deep neural network models [CLL+15]. Developed by Apache, MXNet is highly scalable, portable and flexible, and offers advanced support for GPU optimization [Apa19]. The core language of MXNet is C++, however MXNet provides embedded integration for several host programming languages such as Python, R, Julia, Scala, Matlab, Java, Javascript and Go. The framework represents an approach that combines both a declarative and an imperative programming paradigm, hence the name MXNet (*"mix - net"*) [CLL+15]. The MXNet library is very lightweight as its source code fits into a single 50K line C++ source file with no further dependencies required [CLL+15]. When compared to the other state-of-the-art deep learning frameworks such as Tensorflow, Theano, Caffe and Keras, MXNet has the advantage of successfully merging declarative symbolic expressions together with high performance imperative tensor computations [CLL+15]. Furthermore, MXNet runs extremely fast on any device and operative system without setbacks. As a result, MXNet has been Amazon's choice of deep learning framework and is offered on its AWS cloud service [aws19]. Before explaining how we employ the MXNet framework for the training process of *StefanNet*, we first provide an overview of the core concepts of MXNet we utilize. The programming interface of MXNet introduces several APIs including: symbol, ndarray and module [CLL+15].

5.4.1 Programing APIs for DNN

Symbol API. The Symbol API of MXNet is used to declare the computation graph i.e. the neural network graph [CLL+15]. The concept of a Symbol in MXNet represents a declarative multi-output symbolic expression [MXN19e]. A symbol can be composed of simple matrix operations such as addition or of more complex mathematical operations [CLL+15]. The operators are allowed to take more than one variable as input and can produce several output variables. Furthermore, a Symbol can be represented by a whole neural network layer of any type. Figure 5.4 demonstrates the use of the Symbol API for defining the variables data and label (lines 4-5) as well as for creating a convolutional layer, a batch normalisation layer and an activation layer (lines 8-11). The most important input parameters of the *Convolution()* function are: *kernel* which sets the kernel size, *pad* which defines the size of the padding and *num_filters* i.e. the number of convolutional

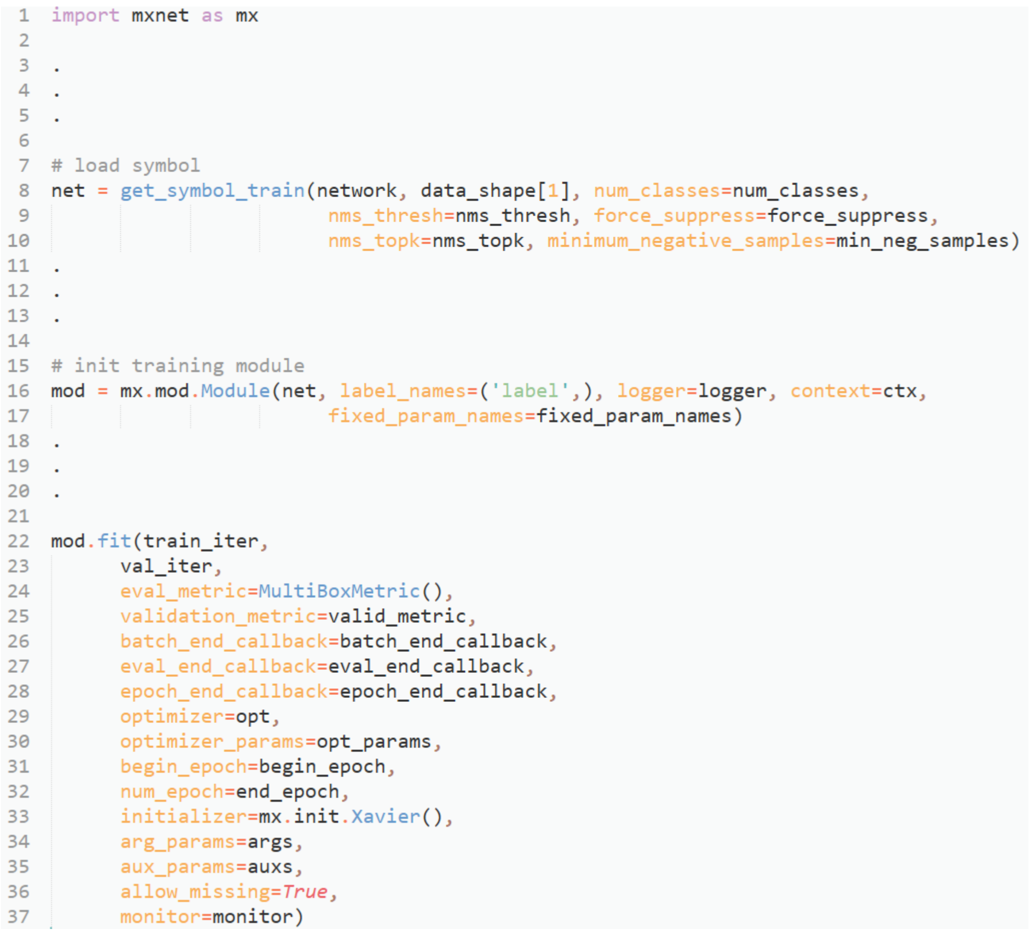
```python
import mxnet as mx

def get_symbol(num_classes=1, **kwargs):
    data = mx.symbol.Variable(name="data")
    label = mx.symbol.Variable(name="label")

    #conv1
    net = mx.symbol.Convolution(
        data=data, kernel=(3, 3), pad=(1, 1), num_filter=64, name="conv1")
    net = mx.sym.BatchNorm(data=net, name='batchnorm1', fix_gamma=True)
    net = mx.sym.Activation(data=net, act_type='relu', name='relub1')
```

Figure 5.4: Symbol API

filters for the layer. The activation layer on the other hand, is identified by its type which is specified in the *act_type* input parameter of the *Activation()* function. The possible values for the *act_type* input parameter can be either relu, sigmoid, softrelu, softsign or tanh. When evaluating a symbol, the free variables are bound with data and the required outputs are declared [CLL+15]. Important to note is that the Symbol API supports not only evaluation ("forward") of symbols but also auto symbolic differentiation ("backward") [CLL+15].

NDArray API. To make up for the gap between the declarative symbolic expressions and the host language MXNet utilizes imperative tensor computation enabled by the NDArray API [CLL+15]. Essentially, an NDArray represents a multi-dimensional, fixed-size homogeneous array [MXN19d]. The NDArray API thus, provides a number of functions for manipulating the shape, size and contents of NDArrays. Some of the operators provided by the NDArray API are quite similar to the ones provided by the Symbol API [MXN19d]. The main difference is, that the NDArray API utilizes imperative programming and the Symbol API follows the declarative programming principles [MXN19d]. However, both APIs can work seamlessly with each other and mixing them is not forbidden [CLL+15]. The NDArray API also provides great support for GPU acceleration.

Module API. The Module API represents a high level interface designed to perform computations on Symbols [MXN19c]. The *Module()* function from the Module API takes a Symbol as an input parameter and creates a Module [MXN19c]. The most important high-level computing functions of the Module API are the *fit()* and *predict()* functions [MXN19c]. The *fit()* function initializes the training process of the deep neural network and will be further elaborated in Section 5.5. In contrast to the *fit()* function, the *predict()* function computes the predictions of the network on new data. Apart from the high-level *fit()* and *predict()* functions, in the Module API there also exist intermediate functions that perform step by step computations. These include the *forward()*, *backward()* and *update()* functions [MXN19c]. While the *forward()* function exhibits a forward pass of



```
 1  import mxnet as mx
 2
 3  .
 4  .
 5  .
 6
 7  # load symbol
 8  net = get_symbol_train(network, data_shape[1], num_classes=num_classes,
 9                                 nms_thresh=nms_thresh, force_suppress=force_suppress,
10                                 nms_topk=nms_topk, minimum_negative_samples=min_neg_samples)
11  .
12  .
13  .
14
15  # init training module
16  mod = mx.mod.Module(net, label_names=('label',), logger=logger, context=ctx,
17                                 fixed_param_names=fixed_param_names)
18  .
19  .
20  .
21
22  mod.fit(train_iter,
23          val_iter,
24          eval_metric=MultiBoxMetric(),
25          validation_metric=valid_metric,
26          batch_end_callback=batch_end_callback,
27          eval_end_callback=eval_end_callback,
28          epoch_end_callback=epoch_end_callback,
29          optimizer=opt,
30          optimizer_params=opt_params,
31          begin_epoch=begin_epoch,
32          num_epoch=end_epoch,
33          initializer=mx.init.Xavier(),
34          arg_params=args,
35          aux_params=auxs,
36          allow_missing=True,
37          monitor=monitor)
```

Figure 5.5: Module API

the data through the network, the *backward()* function performs a backward pass and calculates the gradient [MXN19c]. The *update()* function then updates the parameters using the default optimizer [MXN19c]. Figure 5.5 demonstrates the way we utilized the Symbol and the Module API. In line 8 the Symbol *net* representing the deep neural network is declared. The Symbol *net* then serves as input to the *Module()* function which creates a Module for the given Symbol (line 16). In line 22 the training of the deep neural network is initiated with the help of the *fit()* function.

Data Loading API. Before the training process of a deep neural network model is initialized, the data need to be preprocessed i.e. transformed in a way that corresponds to the format MXNet supports. Apart from the Symbol, Model and NDArray APIs, MXNet also provides a so-called Data Loading API that provides common utility functions which ease the process of iterating and formatting data [MXN19b]. The file format supported by MXNet is called Record IO [MXN19b]. All Record IO files are named with a .rec extension and have numerous advantages over plain image file formats. The Record IO files compactly pack and store the data for efficient read and write operations [MXN19b]. We use the Data Loading API in the way that we create a list of image samples from the BillboardDataset together with their corresponding labels and pass them as an argument to the *im2rec()* ("image to record") function. The *im2rec()* function generates record files from the provided input images which are later on used for training, testing and validation of the deep neural network model [MXN19b]. The iteration of the records is performed with the help of the function *ImageDetRecordIter()* [MXN19b]. This function also offers the possibility for shuffling the data and applying data augmentation techniques on the fly.

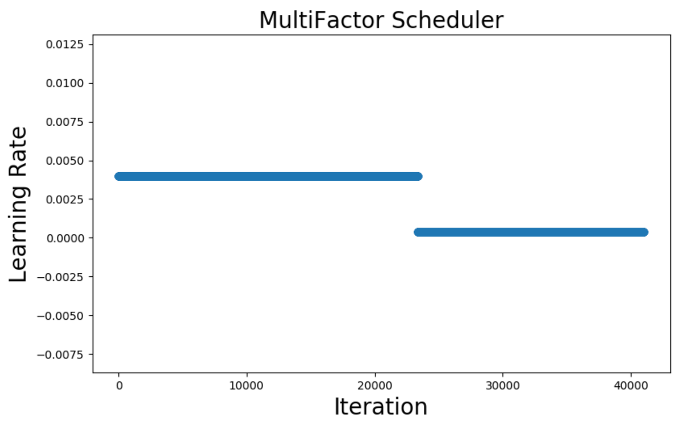
5.5 Training

For the implementation of the *StefanNet* neural network model, the MXNet SSD framework [Zha16] was employed. The MXNet SSD framework enables the creation of deep neural network models with an integration to SSD as object detector. This section focuses on the training phase of *StefanNet* and is divided into two subsections. We first provide an overview of the hyperparameters used for the training and then elaborate the actual training process of *StefanNet* in detail.

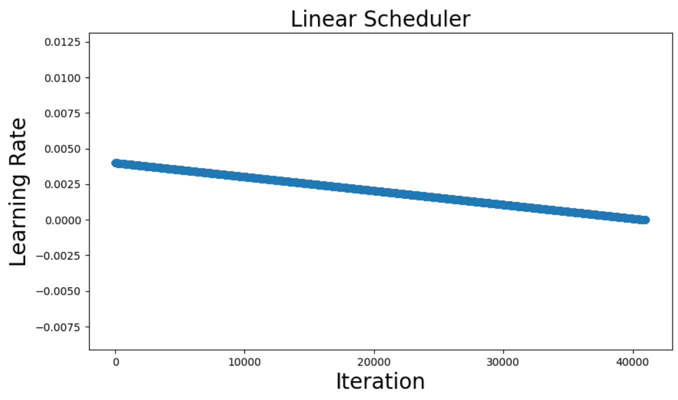
5.5.1 Hyperparameters

When dealing with a deep neural network like *StefanNet*, the key to a successful training phase lies in the hyperparameters. Finding the most suitable values for the hyperparameters is a task of critical importance for obtaining a high performance trained neural network model. *StefanNet* uses the Stochastic Gradient Descent as the main training optimization algorithm. Due to the hardware constraints, the batch size utilized for the training process is configured to 8. Furthermore, during the training phase of *StefanNet*, normalization of the RGB channels of all input images is applied. As a result, the corresponding hyperparameters used for the RGB channel normalization *mean-r*, *mean-g* and *mean-b* are set to 123, 117 and 104 respectively.

The hyperparameter of most crucial importance for the behavior of the training process and the Stochastic Gradient Descent algorithm is the *learning rate*. This is because the learning rate controls the speed of learning and with that it controls the velocity of convergence and the ultimate performance of the network [DBL17]. There exist several strategies for dealing with the learning rate parameter [CZJL15]. One of the simplest possibilities includes having a fixed learning rate throughout the whole training process [CZJL15]. However, changing the learning rate during the training process leads to a better performance and is thus recommended [CZJL15]. In order to leverage the possibility of changing the learning rate, MXNet offers schedulers [MXN19a]. Schedulers are used to define the way of changing the global learning rate and are generally specified per epoch or batch [MXN19a]. For the training of *StefanNet* we employ two kinds of learning rate schedulers: a MultiFactor Scheduler and a Linear Scheduler [MXN19a]. The MultiFactor Scheduler follows a so-called stepwise decay schedule where the learning



(a) MultiFactor Scheduler



(b) Linear Scheduler

Figure 5.6: Schedulers used for the decay of the learning rate during the training of *StefanNet*

rate is decreased by a certain factor at (not necessarily equally spaced) specified intervals [MXN19a]. *StefanNet* is trained for 140 epochs. The initial learning rate parameter is set to 0.004. With the help of the MultiFactor Scheduler, the learning rate is decreased by a factor of 0.1 in epoch 80. In contrast to the sharp decrease of learning rate the MultiFactor Scheduler performs, the Linear Scheduler allows for a much smoother and continuous decay of the learning rate [MXN19a]. From the initial learning rate of 0.004 in the first epoch, the Linear Scheduler smoothly reaches a learning rate of 0 in the last 140th epoch during the training process of *StefanNet*. Figure 5.6a and Figure 5.6b depict the learning rate decay per iteration performed by the MultiFactor Scheduler and the Linear Scheduler.

Another important parameter that has an effect on the intensity of the training of a deep neural network model is *weight decay*. The weight decay is a regularization term, that causes the weights of the network to exponentially decay to zero [HP89][KH92]. In this way the weight decay parameter prevents the weights of the network from getting too large, and thus helps to reduce overfitting in general (see Section 2.3.4 for more details). For the training of *StefanNet* the value of weight decay is set to 0.0005. Furthermore, for the training process of *StefanNet* we also apply *momentum* with a value of 0.9. The momentum helps the Stochastic Gradient Descent algorithm and the training process in general as it helps to accelerate the SGD in the relevant directions and decrease the number of oscillations [Rud16]. This is achieved by taking into consideration the alignment of the directions of the past gradients to better estimate the next best direction (momentum-based approach). The momentum acceleration is maximal when all past gradients are aligned to the same direction [Rud16].

In order to control the evaluation of *StefanNet* during training, we also utilized *non-maximum suppression*. Basically, the non-maximum suppression technique has the task of discovering and dealing with the detections with a confidence score lower than some predefined threshold [HBS17]. The value for the non-maximum suppression hyperparameter is set to 0.45. This means detections with a confidence score lower than 45% are discarded, as they will only add noise to the training (learning) process of *StefanNet*. Lastly, we employ an *overlap* (IoU) parameter with a value of 0.5 (50%) for the training phase.

5.5.2 Training process

The training process starts by loading the record files which will be used for the training of the deep neural neural network model *StefanNet*. In order to read a record file, an iterator is defined with the help of the Data Loading API from MXNet. The iterator is responsible for iterating over the images that are found in the record files for training and validation. Furthermore, the symbol (see Symbol API in Section 5.4) i.e. the structure of the *StefanNet* model is loaded and an executor is defined. The executor sets the hardware device (such as CPU or GPU) that will be used for the training process of the network and defines the structure of the data and the labels. Additionally, a module (see Module API in Section 5.4) for *StefanNet* is created. Essentially, the module of *StefanNet* is used for enabling the training of the network. The most important function that initiates the training process of the network is the *fit()* function from the Module API. The *fit()* function takes the hyperparameters explained in Section 5.5.1 as input. These hyperparameters are also known as optimization parameters, as they serve the purpose of optimizing the performance of *StefanNet* and the training process in general. Lastly, the *Xavier* initializer [MXN19f] is used for initializing the weights of *StefanNet*. The *Xavier* initializer [MXN19f] initiates the weights with random numbers in the range of $[-c, c]$, where c is calculated according to Equation 5.1 [MXN19f]. The n_{in} parameter in Equation 5.1 represents the number of neurons feeding into the weights whereas the n_{out} parameter denotes the number of neurons the result is fed into. The flowchart in

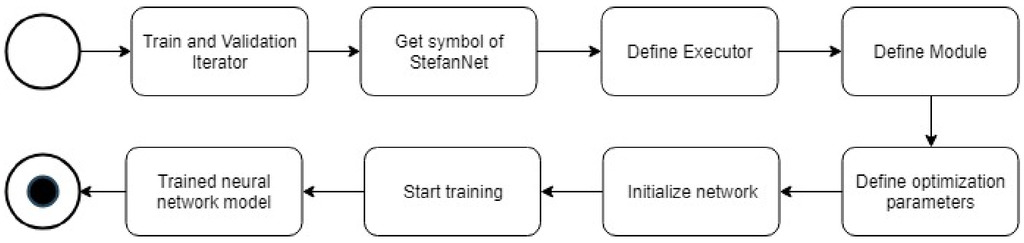

Train and Validation Iterator
Get symbol of StefanNet
Define Executor
Define Module
Define optimization parameters
Initialize network
Start training
Trained neural network model

Figure 5.7: Training process of *StefanNet*

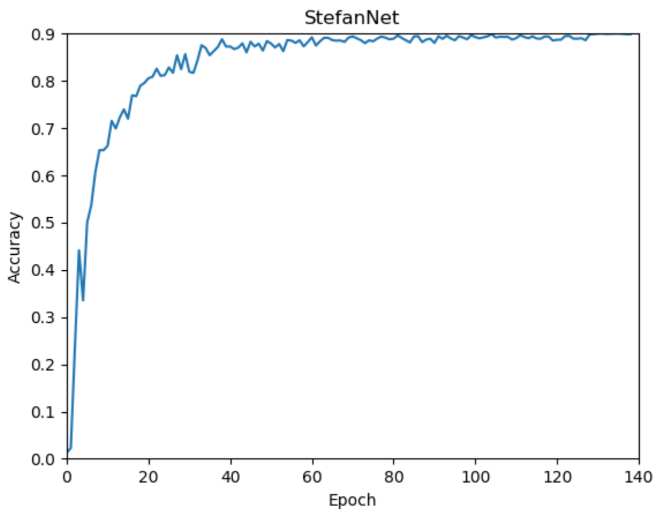


Figure 5.8: Training curve of *StefanNet*

Figure 5.7 depicts the training process of *StefanNet*.

$$c = \sqrt{\frac{3}{0.5 * (n_{\text{in}} + n_{\text{out}})}} \tag{5.1}$$

The resulting model of *StefanNet* consist of two files. The first file is a symbol file which defines the structure of the layers of *StefanNet* in a JSON format. The second file is a parameter file which specifies the weights for each of the layers. The extension of this file is .param. The training process of *StefanNet* given the provided hardware and software specifications (see Section 5.2) lasts circa 48 hours. Figure 5.8 shows the training curve of the training process of *StefanNet*. The curve demonstrates that the training of *StefanNet* runs very smoothly i.e. without drastic oscillations.

5.6 Server side detection

For testing the performance of the *StefanNet* model on video sequences a python script was implemented. The script takes a video file and a neural network model together with its parameters as input and outputs the same video file with the integrated bounding boxes of the detected billboards. The pseudocode of the algorithm is shown in the Algorithm 5.1. The algorithm begins by creating a detector for the *StefanNet* model in line 1. It proceeds by loading the video file in line 2. The video sequence is then evaluated on a frame-wise basis. Hence, each frame is first resized to match the input size of the *StefanNet* model (300x300 pixels). Afterwards, detection is performed on the given resized frame (line 5). If a billboard object is detected in the given frame, the four coordinates of the bounding box are stored in the *bbs* variable. Lastly, the frame together with the bounding box (if any) are presented to the user (line 6). This iterative process loops over all frames of the video.

Algorithm 5.1: Detect video

```
Input: $video, network, prefix, epoch, data-shape, thresh, num-class,$
       $mean-pixels$
Output: Video sequence with detections of billboards

1 $detector \leftarrow get\_detector(network, prefix, epoch, data\_shape,$
          $mean\_pixels, mx.gpu(0), num\_classes, thresh, True)$
2 $video \leftarrow$ Load video sequence
3 for $frame$ to video.end() do
4     $image \leftarrow resize(data\_shape);$
5     $bbs \leftarrow detect(detector, image, thresh);$
6     $vizulize\_detection(image, bbs);$
7 end
```

5.7 Quantization

In order to further optimize and improve the performance of the *StefanNet* neural network model, compression based quantization was applied. The application of the quantization method was performed after the training phase of the *StefanNet* model was completed. Basically, the method of network quantization helps to compress the neural network model by decreasing the number of bits (or the bit-width) necessary for representing each weight of the network [HMD15]. As a result, quantization forces multiple neural network connections to share the same weight and with that, limits the number of effective weights which require storing [HMD15]. At the end of the quantization process, the shared weights are fine-tuned, without any loss of accuracy.

For *StefanNet*, we use quantization to decrease the number of bits for representing the weights from 32-bit float to 16-bit float. The flowchart in Figure 5.9 represents the

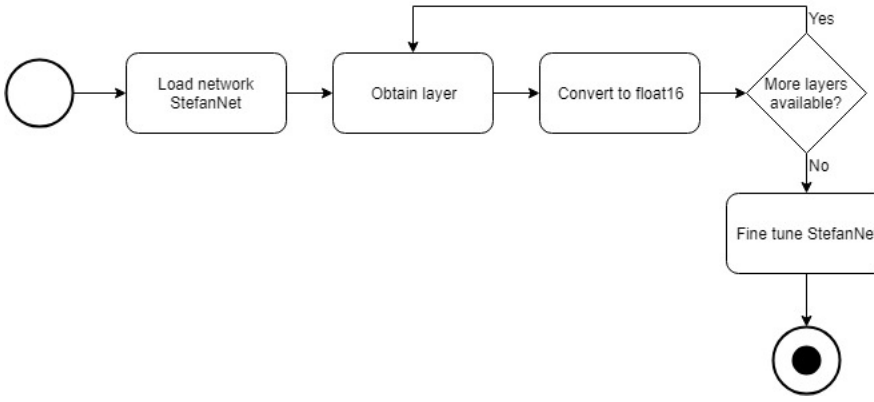


Figure 5.9: Quantization process for *StefanNet*

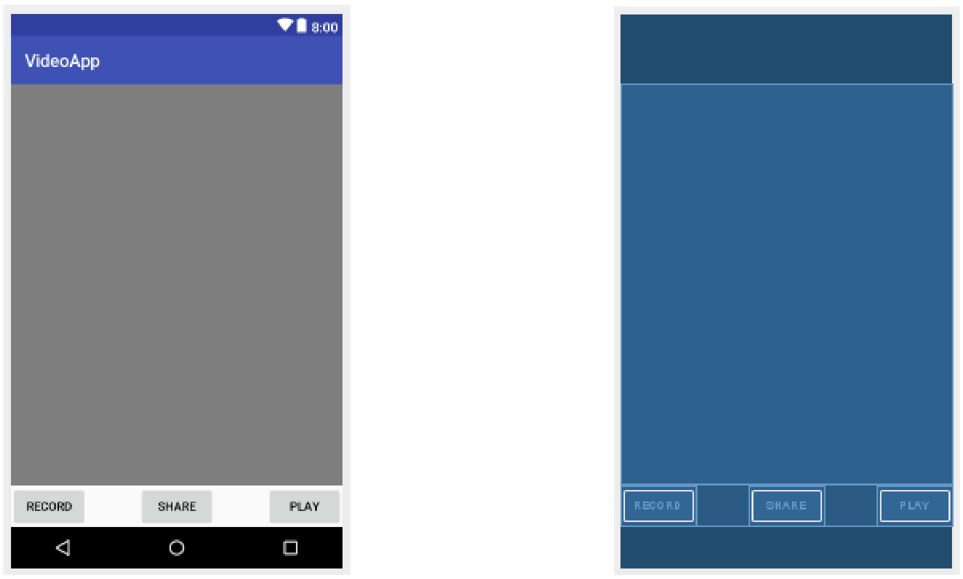
application process of the quantization for *StefanNet*. The process begins by loading the trained *StefanNet* model. Then it iteratively takes each layer of the model and performs a conversion of the data type (bit-width) to float16 in the given layer. Once the decrease of bit-width was performed for every layer of the *StefanNet* model, the shared weights are fine tuned. The fine tuning helps the model to regain its original accuracy. The application of the quantization method resulted in a reduction of size as well as memory overhead of *StefanNet*. Additionally, it provided a speed up of the computation process. This enabled not only faster training of *StefanNet*, but also a significant increase in its inference rate. Further details about the effects and benefits of the quantization can be found in Chapter 6.

5.8 Android application

The Android application is implemented in Android Studio [Goo19] which is the state-of-the-art IDE for developing and implementing Android applications. The programing language used in Android Studio is Java or Kotlin. It offers a Gradle based build support as well as comprehensive layout editor. The layout editor offers multiple layout preview options such as Design and Blueprint (see Figure 5.10) [And19a]. Additionally it provides a drag-and-drop feature that allows the users to easily select and position UI components [And19a].

5.8.1 Components of an Android application

The two main components of an Android application are the layout and the activity [And19d] [And19e]. Generally the layout defines the structure of the user interface of the application [And19e]. The layout can be created in two ways. The first alternative includes self-defining the UI components in an XML file or using the Android Studio's Layout Editor to automatically build the XML file with the help of the drag-and-drop interface [And19e]. The second alternative for creating the layout of an Android application



(a) Design layout (b) Blueprint layout

Figure 5.10: Android studio design layout

involves programmatically instantiating the UI components at runtime [And19e]. The main advantage of the first alternative is that the it separates the presentation layer from the activity code that controls it.

The elements of the layout are composed of objects that belong to either the View or the ViewGroup object classes [And19e]. The objects belonging to the View class usually draw something that the end user of the application can see or interact with [And19e]. Furthermore, the View objects are also known as widgets. Examples of subclasses of the View class include the classes Button, TextView and VideoView. In contrast to the View, the ViewGroup objects represent invisible containers that define and set the layout structure for the View objects [And19e]. Typically, the ViewGroup objects are called layouts. There exist several types of ViewGroup objects each of which provide a different layout structure. Examples of ViewGroup objects include the layout classes LinearLayout, ConstraintLayout and RelativeLayout.

The activity, on the other hand, serves as an entry point to the application's interaction with the end user [And19d]. An Android application typically has one main activity which represents the first screen that appears when the end user opens the application [And19d]. An Android application can have multiple activities and each activity can start another activity. Essentially, the activity classes define the logic and the control code for the UI components introduced in the layout [And19d]. Each activity has a certain number of methods that are called in a very specific order, on specific user actions [And19g]. The methods can be overridden in order to change the way the application reacts to specific conditions. When an activity is created, the method *onCreate()* is called

[And19g]. This is where the layout for the activity is set with the help of the function *setContentView(R.layout.nameOfTheXMLLayoutFile)*. Additionally, in this method, the initialization of UI components such as buttons takes place. If one activity interrupts the flow of another activity, the method *onPause()* is invoked. When the interrupting activity finishes, the method *onResume()* is invoked. Furthermore, there also exist methods like *onStop()*, *onStart()* and *onDestroy()* whose names are considered self-explanatory [And19g].

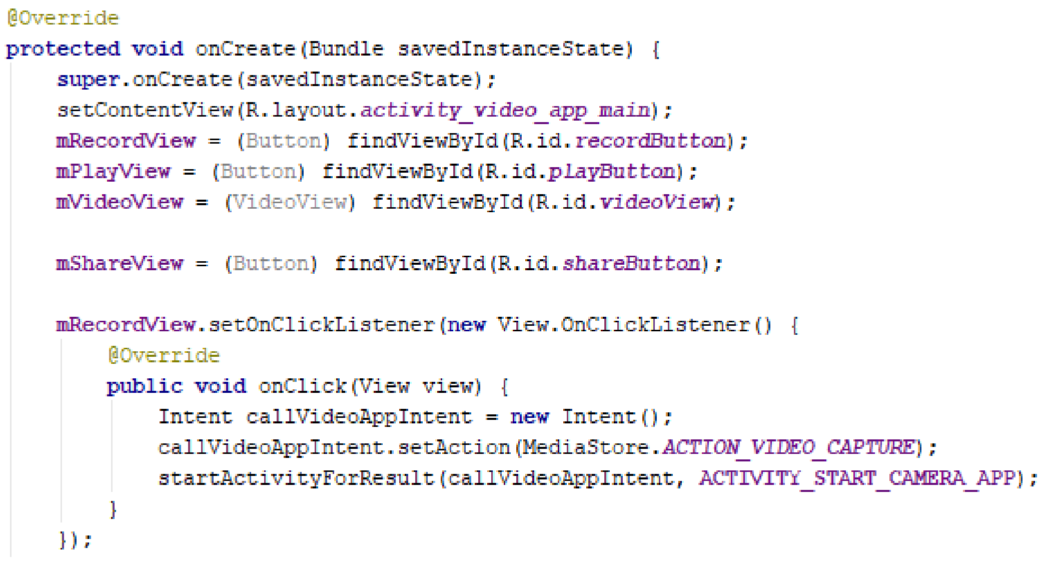
In order to be available and used by the Android application, all activities together with a couple of their most important attributes must be declared in the XML file manifest called *AndroidManifest.xml* [And19d] [And19b]. Apart from the list of all activities, the AndroidManifest file also contains specifications about the permissions, Java packages and other linked libraries used by the application [And19b].

5.8.2 Video Application

The video application represents the client side of the client server system described in Chapter 4.1. The video application is employed by the end-users in order to record videos containing billboard objects. The videos are then transferred to the server side (whose implementation is described above) for billboard object detection. In this section we focus our attention on the implementation details of the video application and its main building components.

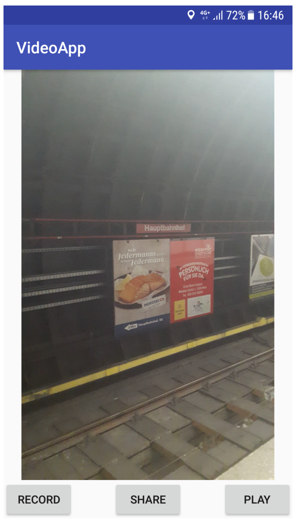
The layout of the video application is fairly simple and very user-friendly. It contains three buttons (View objects or widgets) for recording, playing and sharing a video file. Furthermore, it contains a VideoView object which is used for displaying a video file. The ViewGroup object or the type of layout utilized for the video application is RelativeLayout. The RelativeLayout represents a ViewGroup object that displays child View objects in relative positions [And19f]. This means that the position of each View object of the layout (such as the buttons and the VideoView) is specified either relative to its sibling objects (such as to the left or below View objects) or relative to the parent area (such as aligned with the bottom, center, etc.) [And19f].

The video application has one main activity that defines the actions that the View objects need to execute under certain conditions. The *onCreate()* method of the main activity of the video application is provided in Figure 5.11. For each of the three buttons an individual OnClickListener is defined with the help of the method *setOnClickListener(OnClickListener)* and the method *onClick()* is overridden. As a result, the Android application executes the code (actions) defined in the*onClick()* method whenever the end user clicks on the buttons. When the record button is clicked, an Intent is created with the action set to “MediaStore.ACTION_VIDEO_CAPTURE”. Basically, an Intent is a messaging object used to request an action from another application [And19c]. In this case, a request is sent to the MediaStore for the action of capturing a video. Figure 5.12b depicts the *onClick()* action of the record button. When the play button is clicked, the video content from the VideoView object is started (played). Lastly, when the share

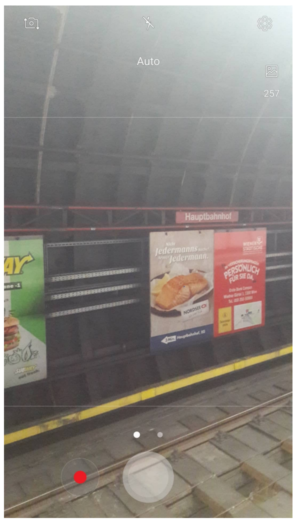


```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_video_app_main);
    mRecordView = (Button) findViewById(R.id.recordButton);
    mPlayView = (Button) findViewById(R.id.playButton);
    mVideoView = (VideoView) findViewById(R.id.videoView);

    mShareView = (Button) findViewById(R.id.shareButton);

    mRecordView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent callVideoAppIntent = new Intent();
            callVideoAppIntent.setAction(MediaStore.ACTION_VIDEO_CAPTURE);
            startActivityForResult(callVideoAppIntent, ACTIVITY_START_CAMERA_APP);
        }
    });
```

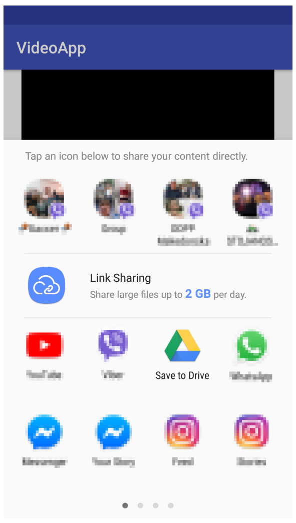
Figure 5.11: Android studio onCreate



(a) Play



(b) Record


VideoApp
Tap an icon below to share your content directly.
Link Sharing
Share large files up to 2 GB per day.
Save to Drive

(c) Share

Figure 5.12: Video application

button is clicked an Intent for sharing the video content is created. The effect of clicking the share button can be seen in Figure 5.12c.

In this section we described the implementation process and details of both the server side and the client side of our proposed system for automatic billboard detection. We reviewed the training process of *StefanNet* and described the application process of the compression based method quantization. In the next section we present the results our system obtained on the task of billboard detection and compare them to our benchmarking results.

CHAPTER 6

Evaluation and Results

Based on the created datasets presented in Chapter 4 and the implementation process described in Chapter 5, in this chapter we provide a detailed overview of the evaluation and achieved results of our proposed system for automatic billboard detection. We first describe the methodology used for testing, including the different configurations of *StefanNet* as well as the configurations of the state-of-the-art neural network models we employ as benchmarks. We proceed by presenting the test and validation results which include the evaluation of both frontal and side view detection of billboard images. Lastly, we finish this chapter by providing an evaluation of the effects and results of the application of the quantization method.

6.1 Test methodology

For the testing purposes of our proposed system for automatic billboard detection three different configuration models of *StefanNet* were created. Table 6.1 provides an overview of the configurations of *StefanNet*. We distinguish between configuration models A, B and C which mainly differ in the dataset they were trained on and in the type of learning rate schedule they employed. The configuration model *StefanNet-A* was trained on the *BillboardDataset* using a MultiFactor Scheduler. In contrast to configuration model A, the configuration model *StefanNet - B* was trained on the augmented dataset *BillboardDatasetPlus* and the learning rate was scheduled by a MultiFactor Scheduler. Finally, the configuration model *StefanNet - C* was also trained on the *BillboardDatasetPlus*, but a Linear Scheduler was employed.

In order to be able to compare the results of *StefanNet* with the performance of the current state-of-the-art neural networks, we chose four different neural network models as benchmarks: ResNet [HZRS16], MobileNet [HZC$^+$17], VGG [SZ14b] and Inception [AHYT17]. All of the chosen benchmark neural network models have been integrated with SSD as their object detector. The integration of SSD in these networks has been

Model	Dataset	Scheduler
StefanNet - A	BillboardDataset	MultiFactor
StefanNet - B	BillboardDatasetPlus	MultiFactor
StefanNet - C	BillboardDatasetPlus	Linear

Table 6.1: Configuration models of *StefanNet*

performed in a similar fashion to the integration process of SSD in *StefanNet* explained in Section 5.3.3. The benchmark neural networks have been trained on the *BillboardDataset* and have the same configuration of the training hyperparameters as *StefanNet*. The Stochastic Gradient Descent with an initial learning rate of 0.004, weight decay of 0.0005 and momentum of 0.9 have been used. Furthermore, a stepwise learning rate decay with the help of a MultiFactor Scheduler has been employed. The values of the rest of the hyperparameters can be found in Section 5.5.1. Important to note is that the achieved results of the state-of-the-art neural networks and the *StefanNet* models presented below, are provided in terms of the mean average precision (mAp) metric, the calculation of which is explained in Section 2.5.2.

6.2 Results

The results presented in Table 6.2 show the overall test performance of the three configurations of *StefanNet* and the state-of-the-art neural networks. We begin with evaluating the performance of the three individual *StefanNet* configurations on the specified test data explained in Section 4.2. As it can be seen from Table 6.2, the *StefanNet* configurations B and C which were trained on the *BillboardDatasetPlus*, outperform configuration A which was trained on the standard *BillboardDataset*. Furthermore, it can be observed that for the test dataset the type of scheduler does not influence the overall mAp score as both configurations B and C achieved 91% mAp despite their difference in the scheduler employed.

The results in Table 6.2 also indicate that our two deep neural network model configurations B and C outperform all state-of-the-art neural network models in terms of the percentage of mAp achieved. With 91% mAp, the *StefanNet* models outperform the best-performing state-of-the-art neural network ResNet by 0.3%. However, the *StefanNet* models demonstrate a significantly faster inference rate of 40 frames per second, in contrast to the ResNet's 26 frames per second. This is remarkable considering that the *StefanNet* models contain only 23 convolutional layers - significantly less than the 152 convolutional layers employed by ResNet. The best inference rate of 67 frames per second was achieved by the state-of-the-art neural network MobileNet300 with an mAp score of 85.5%. We note that this inference speed is due to the depthwise convolutions that only MobileNet implements.

Figure 6.1 demonstrates the Precision-Recall curves of the benchmark neural network

Model	Input shape	# Conv layers	mAp %	FPS
Inception	512	22	88	22.5
ResNet	512	152	90.7	26
MobileNet300	300	28	85.5	67
MobileNet512	512	28	88.2	44
VGG16	300	16	89	45
StefanNet - A	300	23	90	40
StefanNet - B	300	23	**91**	40
StefanNet - C	300	23	**91**	40

Table 6.2: Test results

models and the best configuration of the *StefanNet* model. Both Figure 6.1 and the results presented in Table 6.2 show that our *StefanNet* architecture achieves competitive results and outperforms all benchmark neural network models considered on the task of billboard detection.

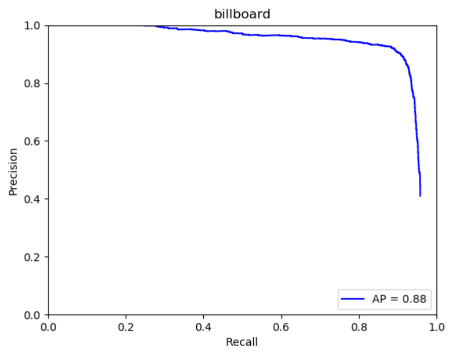
6.3 Validation

The process of validation of the *StefanNet* model is divided into two large tasks: frontal validation and side validation. The frontal validation consists of validating the *StefanNet* as well as the benchmark neural network models on a dataset consisting of images that have a frontal view of billboard objects. Similarly, the side validation task performs validation of all neural network models using a dataset which contains image samples with a side or a profile view of the billboard objects. Furthermore, we want to point out that the image samples (frontal and side validation dataset) used for validation are neither used for training nor for testing purposes.

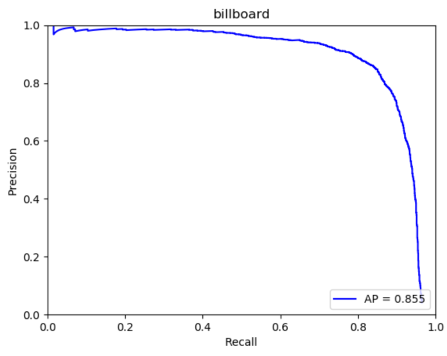
6.3.1 Frontal view validation

Table 6.3 shows the results of the frontal view validation and Figure 6.2 presents the corresponding Precision-Recall curves of all neural network models. As it can be observed, there exists a noticeable difference in the performance considering the three configurations of *StefanNet*. Similarly to the test results, out of the three configurations on the frontal validation dataset, configuration A performed worst with an mAp score of 90.7%. Interestingly, configuration C which was trained on the *BillboardDatasetPlus* using a Linear scheduler performed best with an mAp score of 98%. Configuration C outperformed the *StefanNet* configuration B by 1.3%, which was trained using a MultiFactor scheduler instead.

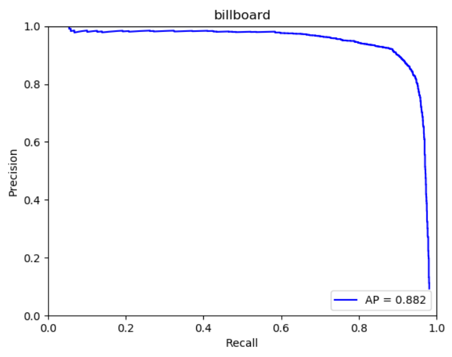
When compared to the state-of-the-art neural networks, both configurations B and C produced superior results. From the results in Table 6.3 it can be observed that the result



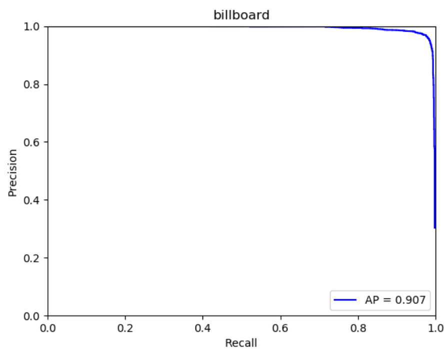
(a) Inception



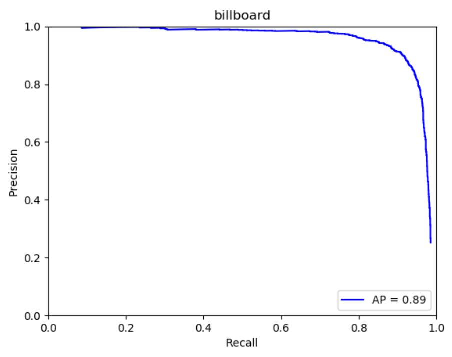
(b) MobileNet300



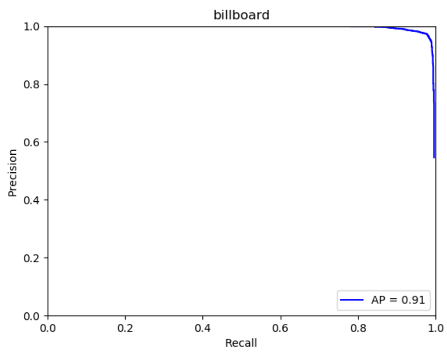
(c) MobileNet512



(d) ResNet



(e) VGG16



(f) StefanNet-C

Figure 6.1: Precision - Recall curves of all models on the test evaluation dataset

Model	Input shape	# Conv layers	mAp %
Inception	512	22	90.5
ResNet	512	152	88.6
MobileNet300	300	28	89.7
MobileNet512	512	28	92.4
VGG16	300	16	96
StefanNet - A	300	23	90.7
StefanNet - B	300	23	**96.7**
StefanNet - C	300	23	**98**

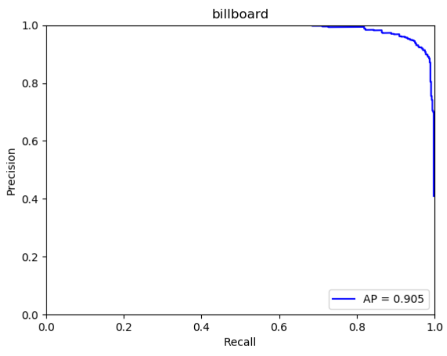
Table 6.3: Frontal validation results

achieved by configuration C (98% mAp) exceeds the results of all benchmark neural network models by at least 2%. On the other hand, the mAp score of 96.7% achieved by configuration B outperforms the results achieved by the benchmark neural network models by at least 0.7%. Out of all benchmark neural network models, VGG16 performed best with an mAp score of 96%. Surprisingly, the benchmark neural network model ResNet achieved only 88.6% mAp on the frontal validation dataset i.e. circa 10% less than our best performing *StefanNet* model. We believe that the outstanding performance of StefanNet - C is due to the employment of the Linear scheduler which smoothly decreases the value of the learning rate during training. Additionally, we note that the increase in training samples significantly contributes to the performance of configurations B and C which outperform configuration A.

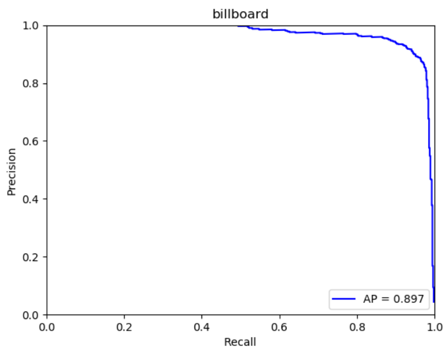
6.3.2 Side view validation

In a similar manner to the frontal view validation, we now present the results of the side validation task. Table 6.4 shows the results of the side validation. As it can be seen by the results in Table 6.4, the mAp score of all neural network models (including the benchmark and the three *StefanNet* configurations) on the side validation dataset is lower. We note that this is due to the increased complexity of the task of detecting billboard objects in images which have a side or profile view of billboard objects as opposed to detecting billboards in images which have a frontal view of billboard objects.

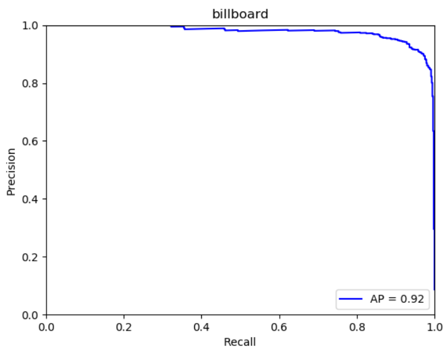
Even though the achieved results are lower in terms of mAp, the same performance pattern can be observed in the three *StefanNet* configurations. The configurations C and B outperformed the configuration A (72.6 % mAp) by achieving an mAp score of 82% and 80.5% respectively. Similarly to the results of the frontal validation, both configurations C and B produced outstanding results outperforming the results of every benchmark neural network model considered. The best-performing benchmark neural network model on the side validation dataset is ResNet with an mAp score of 80%. Figure 6.3 depicts the side validation Precision-Recall curves of all benchmark neural network models including the best configuration model of *StefanNet*. Having configuration C achieve the best results,



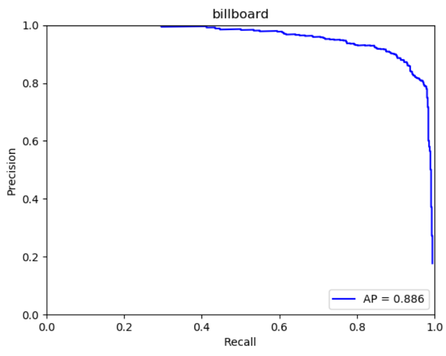
(a) Inception



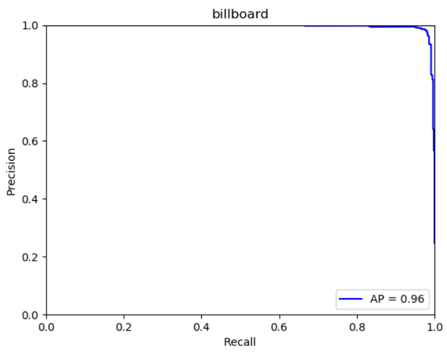
(b) MobileNet300



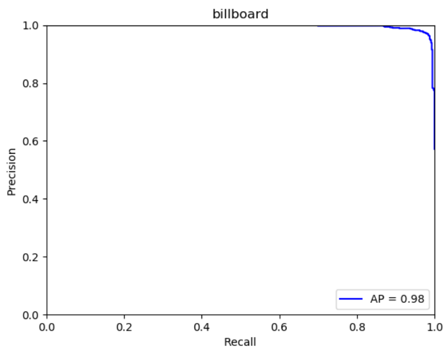
(c) MobileNet512



(d) ResNet



(e) VGG16



(f) StefanNet-C

Figure 6.2: Precision - Recall curves of all models on the frontal validation dataset

Model	Input shape	# Conv layers	mAp %
Inception	512	22	74
ResNet	512	152	80
MobileNet300	300	28	68
MobileNet512	512	28	72
VGG16	300	16	78
StefanNet - A	300	23	72.6
StefanNet - B	300	23	**80.5**
StefanNet - C	300	23	**82**

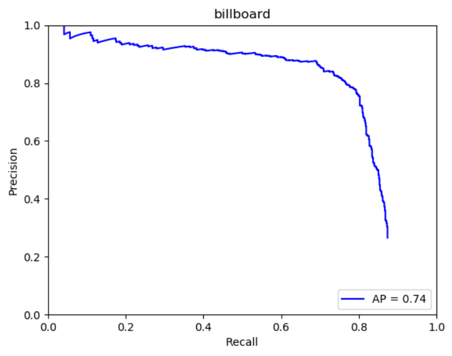
Table 6.4: Side validation results

just proves that the Linear scheduler is more robust and produces better performing models than the MultiFactor scheduler.

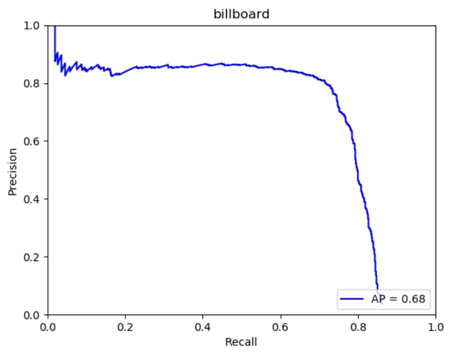
6.4 Quantization results

We applied the compression based quantization technique on our best-performing configuration model of *StefanNet* - configuration C. The details of the application process can be found in Section 5.7. After the application of the quantization method, the bit-width necessary for storing the weights of the *StefanNet* network was reduced by half. This was achieved by converting the type of the parameters from float32 to float16. We also experienced a significant improvement in the training speed, which was increased by a factor of 2. This is due to the fact that we were now able to set the batch size of the quantized *StefanNet* model to 16. The application of the quantization method also resulted in a decrease of the physical size of the *StefanNet* model. The size of the *StefanNet* model was reduced by 30% from 90MB to 60MB. Furthermore, the inference rate of the *StefanNet* model was increased by circa 10%, from 40 frames per second to 45 frames per second. These advantages were all achieved without substantial loss of the high mAp scores of the model.

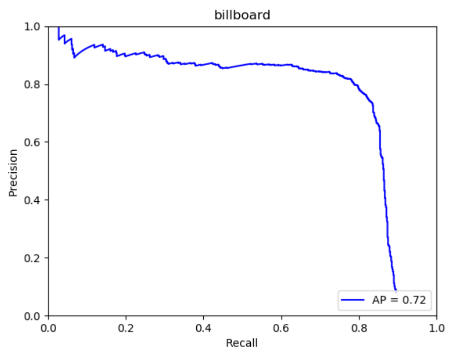
Figure 6.4 represents the Precision-Recall curves of the performance of the quantized *StefanNet* model on the test and validation datasets. Furthermore, Table 6.5 and Table 6.6 present the achieved results of the quantized *StefanNet* model together with the achieved results of the benchmarks considered for comparison. As experienced by the results shown in Table 6.5 the quantized *StefanNet* model still achieved the highest mAp score of 91% on the test dataset. However, its inference rate has been enhanced to 45 frames per second. Moreover, the quantized variant of *StefanNet* achieved an outstanding performance on both validation datasets (see Table 6.6). With an 96% mAp on the frontal validation dataset and an 85% mAp on the side validation dataset, the quantized model of *StefanNet* outperforms the results of all state-of-the-art neural networks taken into consideration. The exceptional performance of *StefanNet* on the



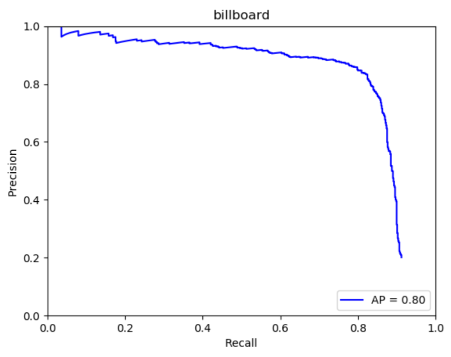
(a) Inception



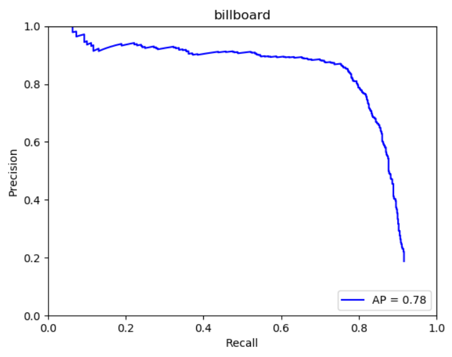
(b) MobileNet300



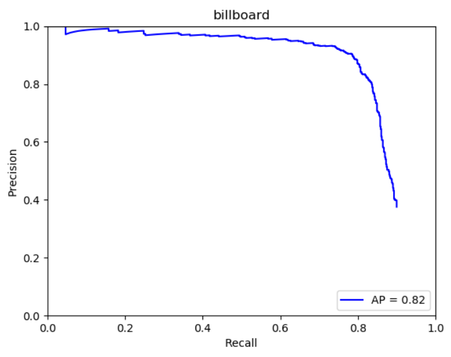
(c) MobileNet512



(d) ResNet



(e) VGG16



(f) StefanNet-C

Figure 6.3: Precision - Recall curves of all models on the side validation dataset

Model	Input shape	# Conv layers	mAp (test) %	FPS
Inception	512	22	88	22.5
ResNet	512	152	90.7	26
MobileNet300	300	28	85.5	67
MobileNet512	512	28	88.2	44
VGG16	300	16	89	45
StefanNet-C (quantized)	300	23	**91**	**45**

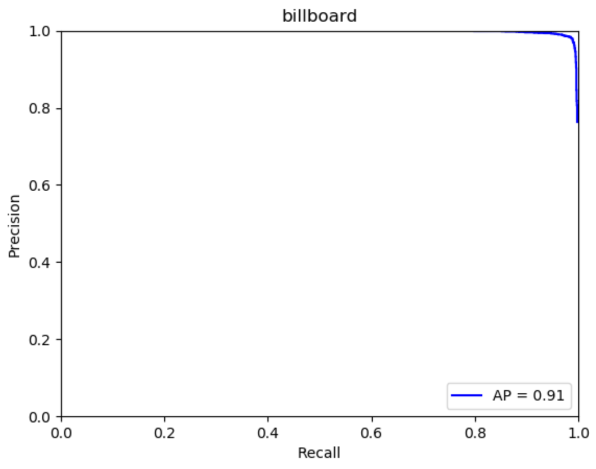
Table 6.5: Quantization results on the test dataset

Model	Input shape	# Conv layers	mAp (frontal view)	mAp (side view)
Inception	512	22	90.5	74
ResNet	512	152	88.6	80
MobileNet300	300	28	89.7	68
MobileNet512	512	28	92.4	72
VGG16	300	16	96	78
StefanNet-C (quantized)	300	23	**96**	**85**

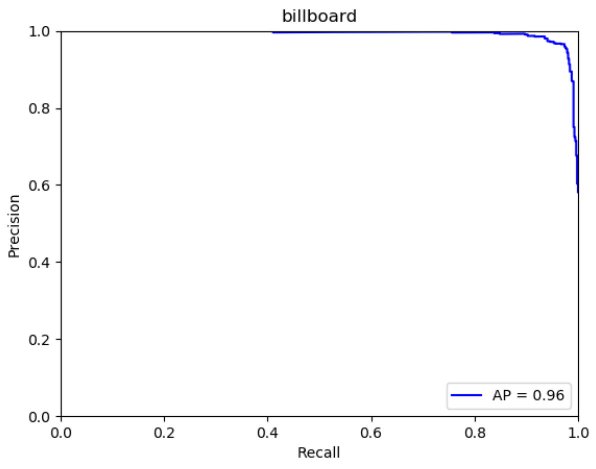
Table 6.6: Quantization results on the validation datasets

test and validation datasets proves that *StefanNet* performs on par with the current state-of-the-art networks. In fact, *StefanNet* outperforms the benchmark networks in every evaluation category which confirms that the *StefanNet* architecture is the most suitable for the task of automatic billboard detection.

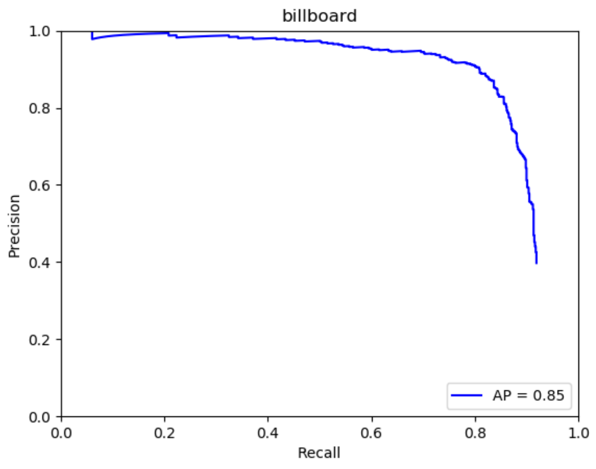
In this section we presented the three different configurations of *StefanNet* and evaluated their performance on the test and validation datasets. Furthermore, we presented an overview of the state-of-the-art neural network models considered as benchmarks and performed a comparison of their performance to the one achieved by *StefanNet*. We also reviewed the benefits and results of the application of the method quantization on the best model of *StefanNet*. In the next section we will summarize and conclude the master thesis by highlighting all of the contributions and accomplishments as well as the future work to come.



(a) test evaluation



(b) frontal validation



(c) side validation

Figure 6.4: Precision - Recall curves of the quantized *StefanNet* model on the test and validation dataset

CHAPTER 7

Conclusion

In this chapter, I present the main contributions and achievements of this master thesis. First, we collected image samples containing billboard objects located throughout all metro stations in Vienna, Austria. We then created two datasets called *BillboardDataset* (4042 image samples) and *BillboardDatasetPlus* (5002 image samples) which represents an artificially augmented extension of the original *BillboardDataset*. We then proposed and implemented a client server system with a deep learning-based architecture for automatic billboard detection. This included the development of the client side which is represented by a lightweight Android application used for gathering video data streams and sharing them to a cloud environment. On the other hand, the implementation of the server side was executed in the MXNet framework and its main task encompassed the design and implementation of *StefanNet*, a self-defined deep neural network model developed specifically for billboard detection. After the training of *StefanNet*, we performed a thorough analysis and evaluation of its performance. For the evaluation, we considered four state-of-the-art deep neural network models as benchmarks which were trained and used for comparison purposes. Lastly, we implemented and evaluated the benefits of the application of the compression-based quantization technique on *StefanNet*.

On the task of billboard detection, the best configuration of our *StefanNet* model achieved 91% mAp on the test dataset. It achieved 98% mAp on the validation dataset containing images with a front view of billboard objects and 82% mAp on the validation dataset containing images with a side view of billboard objects. The inference rate of the *StefanNet* model reached 40 frames per second. The quantized version of this model achieved the same mAp score on the test dataset. However, it achieved 96% mAp on the frontal validation dataset and 85% mAp on the side validation dataset. Additionally, the quantized version of the *StefanNet* model reached an inference rate of 45 frames per second. In comparison to the considered benchmark neural networks ResNet, MobileNet, Inception and VGG16, both the *StefanNet* model and the quantized version of the model produce superior results and outperform the state-of-the-art neural network models on the test and the validation datasets. For the frontal view validation of the models, *StefanNet* with only 23 convolutional layers outperformed the deepest state-of-the-art neural network ResNet-152 by 10%. Compared to the other models *StefanNet* outperformed VGG16 by 2%, Inception by 8% and MobileNet by 6% on the images with frontal view. For the side view evaluation *StefanNet* outperformed ResNet-152 by 2%, VGG16 by 6%, Inception by 9% and MobileNet by 10%. This confirms that the architecture of *StefanNet* is the most suitable for the specific problem domain of automatic billboard detection in video streams.

As the current system for automatic billboard detection focuses only on detecting billboard objects which are located throughout the metro lines, possible future work includes extending the problem domain to billboards which are located throughout the city. This would mean extending the size of the two datasets *BillboardDataset* and *BillboardDatasetPlus* with image samples taken from various billboards and different billboard shapes which can be found throughout Vienna. Furthermore, other data augmentation techniques can be explored to artificially increase the number of image samples. Another possibility includes further improvement of the inference rate of the *StefanNet* model. This could be achieved by integrating depthwise convolutions in the *StefanNet* model instead of the normal convolutions the current model implements. Additionally, the quantization process can be optimized to compress the model even more and reduce the bit-width necessary for storing the weights of the network into int8 values. Moreover, analysis of other compression based methods like pruning can be performed and applied to the *StefanNet* model. Finally, the *StefanNet* model could be used within a greater scope in the area of augmented reality, such as targeted (personalized) advertising.

List of Figures

List of Tables

Bibliography

[ABMM16] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *CoRR*, abs/1611.01491, 2016.

[AHYT17] Md. Zahangir Alom, Mahmudul Hasan, Chris Yakopcic, and Tarek M. Taha. Inception recurrent convolutional neural network for object recognition. *CoRR*, abs/1704.07709, 2017.

[AKAL18] Khaled Almgren, Murali Krishnan, Fatima Aljanobi, and Jeongkyu Lee. Ad or non-ad: A deep learning approach to detect advertisements from magazines. *Entropy*, 20(12), 2018.

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning.* The MIT Press, 2nd edition, 2010.

[Alp16] Ethem Alpaydin. *Machine Learning: The New AI.* MIT Press, 2016.

[And19a] Android Developers Docs. Android studio layout editor. `https://developer.android.com/studio/write/layout-editor.html`, last accessed on: 2019-04-06, 2019.

[And19b] Android Developers Docs. App manifest overview. `https://developer.android.com/guide/topics/manifest/manifest-intro`, last accessed on: 2019-04-06, 2019.

[And19c] Android Developers Docs. Intents and intent filters. `https://developer.android.com/guide/components/intents-filters`, last accessed on: 2019-04-06, 2019.

[And19d] Android Developers Docs. Introduction to activities. `https://developer.android.com/guide/components/activities/intro-activities`, last accessed on: 2019-04-06, 2019.

[And19e] Android Developers Docs. Layouts documentation. `https://developer.android.com/guide/topics/ui/declaring-layout`, last accessed on: 2019-04-06, 2019.

[And19f] Android Developers Docs. Relative layout. https://developer.android.com/guide/topics/ui/layout/relative, last accessed on: 2019-04-06, 2019.

[And19g] Android Developers Docs. Understand the activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle, last accessed on: 2019-04-06, 2019.

[Apa19] Apache Software Foundation. Apache mxnet. http://mxnet.incubator.apache.org, last accessed on: 2019-03-31, 2019.

[ATY+18] Md. Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Mahmudul Hasan, Brian C. Van Esesn, Abdul A. S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164, 2018.

[AV07] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[aws19] Apache mxnet on aws. https://aws.amazon.com/mxnet/, last accessed on: 2019-03-31, 2019.

[BA18] C. Bircanoğlu and N. Arıca. A comparison of activation functions in artificial neural networks. In *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, May 2018.

[BAA16] Sourour Brahimi, Najib Ben Aoun, and Chokri Ben Amar. Very deep recurrent convolutional neural network for object recognition. In *ICMV*, volume 10341 of *SPIE Proceedings*, page 1034107. SPIE, 2016.

[BBMS17] Simone Bianco, Marco Buzzelli, Davide Mazzini, and Raimondo Schettini. Deep learning for logo recognition. *Neurocomputing*, 245:23–30, 2017.

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition.* Information science and statistics. Springer, 2007.

[BL08] Matthew B. Blaschko and Christoph H. Lampert. Learning to localize objects with structured output regression. In David Forsyth, Philip Torr, and Andrew Zisserman, editors, *Computer Vision – ECCV 2008*, pages 2–15, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CAP+12] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek. A brief introduction to opencv. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1725–1730, May 2012.

[CAS16] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198. ACM, 2016.

[CBF06] M. Covell, S. Baluja, and M. Fink. Advertisement detection and replacement using acoustic and visual repetition. In *2006 IEEE Workshop on Multimedia Signal Processing*, pages 461–466, Oct 2006.

[Cho16] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[CLL$^+$15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[CS12] J. Carreira and C. Sminchisescu. Cpmc: Automatic object segmentation using constrained parametric min-cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1312–1328, July 2012.

[CSL$^+$16] Junxuan Chen, Baigui Sun, Hao Li, Hongtao Lu, and Xian-Sheng Hua. Deep CTR prediction in display advertising. *CoRR*, abs/1609.06018, 2016.

[CWHH14] Z. Chen, J. Wang, H. He, and X. Huang. A fast deep learning system using gpu. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1552–1555, June 2014.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *MCSS*, 2(4):303–314, 1989.

[CZJL15] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. A learning-rate schedule for stochastic gradient methods to matrix factorization. In *Advances in Knowledge Discovery and Data Mining - 19th Pacific-Asia Conference, PAKDD 2015, Ho Chi Minh City, Vietnam, May 19-22, 2015, Proceedings, Part I*, pages 442–455, 2015.

[Dab17] Imad Dabbura. Gradient descent algorithm and its variants. `https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3`, last accessed on: 2019-03-23, 2017.

[DBL17] *2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017, Santa Rosa, CA, USA, March 24-31, 2017*. IEEE Computer Society, 2017.

[DDS$^+$09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[Die95] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, September 1995.

[DY14] Li Deng and Dong Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3&#8211;4):197–387, June 2014.

[EEG$^+$15a] Mark Everingham, S. M. Eslami, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *Int. J. Comput. Vision*, 111(1):98–136, January 2015.

[EEG$^+$15b] Mark Everingham, S. M. Ali Eslami, Luc J. Van Gool, Christopher K. I. Williams, John M. Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, 2015.

[EVGW$^+$10] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[Faw06] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`, last accessed on: 2019-03-23.

[GDDM14] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, pages 580–587. IEEE Computer Society, 2014.

[Gir15] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[GLL$^+$18] Jianbo Guo, Yuxi Li, Weiyao Lin, Yurong Chen, and Jianguo Li. Network decoupling: From regular to depthwise separable convolutions. *CoRR*, abs/1808.05517, 2018.

[GM99] Ben Gold and Nelson Morgan. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[Goo19] Google, JetBrains. Android studio. `https://developer.android.com/studio`, last accessed on: 2019-04-06, 2019.

[Hay09] Simon S. Haykin. *Neural networks and learning machines.* Pearson Education, Upper Saddle River, NJ, third edition, 2009.

[HBS17] Jan Hendrik Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. *CoRR*, abs/1705.02950, 2017.

[HDN+18] Murhaf Hossari, Soumyabrata Dev, Matthew Nicholson, Killian McCabe, Atul Nautiyal, Clare Conran, Jian Tang, Wei Xu, and François Pitié. Adnet: A deep network for detecting adverts. *CoRR*, abs/1811.04115, 2018.

[Hin87] Geoffrey E Hinton. Learning translation invariant recognition in a massively parallel networks. In *International Conference on Parallel Architectures and Languages Europe*, pages 1–13. Springer, 1987.

[HMD15] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[HP89] Stephen Jose Hanson and Lorien Y. Pratt. Comparing biases for minimal network construction with back-propagation. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 177–185. Morgan-Kaufmann, 1989.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[HSK+12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[HZC+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[HZZ+17] Zaeem Hussain, Mingda Zhang, Xiaozhong Zhang, Keren Ye, Christopher Thomas, Zuha Agha, Nathan Ong, and Adriana Kovashka. Automatic understanding of image and video advertisements. *CoRR*, abs/1707.03067, 2017.

[IAB17] IAB. Iab internet advertising revenue report. `https://www.iab.com/wp-content/uploads/2017/12/IAB-Internet-Ad-Revenue-Report-Half-Year-2017-REPORT.pdf`, last accessed on: 2019-03-23, 2017.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pages 448–456, 2015.

[JC17] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *CoRR*, abs/1702.05659, 2017.

[JVZ14] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014.

[Kar16] Andrej Karpathy. Convolutional neural networks (cnns / convnets). `http://cs231n.github.io/convolutional-networks/`, last accessed on: 2019-03-23, 2016.

[KH92] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*, pages 950–957. Morgan Kaufmann, 1992.

[KKB18] Leslie Pack Kaelbling Kenji Kawaguchi and Yoshua Bengio. Generalization in deep learning. In *Mathematics of Deep Learning, Cambridge University Press, to appear. Prepint avaliable as: MIT-CSAIL-TR-2018-014, Massachusetts Institute of Technology*, 2018.

[KOLW16] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Object detection from video tubelets with convolutional neural networks. *CoRR*, abs/1604.04053, 2016.

[Kri18] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018.

[KSH12a] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[KSH12b] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1106–1114, 2012.

[KUE11] Ekaterina V Karniouchina, Can Uslay, and Grigori Erenburg. Do marketing media have life cycles? the case of product placement in movies. *Journal of Marketing*, 75(3):27–48, 2011.

[KW52] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[Lab16] LabelMe. Labelme annotation tool. `https://github.com/wkentaro/labelme`, last accessed on: 2019-03-23, 2016.

[LAE$^+$15] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[LBD$^+$89] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[LGEC17] A. L'Heureux, K. Grolinger, H. F. Elyamany, and M. A. M. Capretz. Machine learning with big data: Challenges and approaches. *IEEE Access*, 5:7776–7797, 2017.

[LH15] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[LL15] Hao Li and Hui-Yi Lo. Do you recognize its brand? the effectiveness of online in-stream video advertisements. *Journal of Advertising*, 44(3):208–218, 2015.

[LMB$^+$14] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[LRM04] Keith Lau, Ronald A. Rensink, and Tamara Munzner. Perceptual invariance of nonlinear focus+context transformations. In *Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization*, APGV '04, pages 65–72, New York, NY, USA, 2004. ACM.

[Mit06] Tom Mitchell. The discipline of machine learning. Technical Report CMU ML-06 108, 2006.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, USA, 1969.

[MRSM06] Marvin L. Minsky, Nathaniel Rochester, Claude E. Shannon, and John McCarthy. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. 27(4):12, 2006.

[MXN19a] Apache MXNet. Learning rate schedules. `https://mxnet.incubator.apache.org/versions/master/tutorials/gluon/learning_rate_schedules.html`, last accessed on: 2019-03-31, 2019.

[MXN19b] Apache MXNet. Mxnet data loading api documentation. `http://mxnet.incubator.apache.org/test/api/python/io.html`, last accessed on: 2019-03-31, 2019.

[MXN19c] Apache MXNet. Mxnet module api documentation. `https://mxnet.apache.org/api/python/module/module.html`, last accessed on: 2019-03-31, 2019.

[MXN19d] Apache MXNet. Mxnet ndarray api documentation. `https://mxnet.apache.org/api/python/ndarray/ndarray.html`, last accessed on: 2019-03-31, 2019.

[MXN19e] Apache MXNet. Mxnet symbol api documentation. `https://mxnet.apache.org/api/python/symbol/symbol.html`, last accessed on: 2019-03-31, 2019.

[MXN19f] Apache MXNet. Xavier initialization for weights. `https://mxnet.apache.org/api/python/optimization/optimization.html#mxnet.initializer.Xavier`, last accessed on: 2019-03-31, 2019.

[OHL$^+$08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[Ped18] Dabal Pedamonti. Comparison of non-linear activation functions for deep neural networks on MNIST classification task. *CoRR*, abs/1804.02763, 2018.

[PG17] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach.* O'Reilly, Beijing, 2017.

[PMG97] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach.* Oxford University Press, Inc., New York, NY, USA, 1997.

[Pow11] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[PW17] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[RDGF15] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[RDS$^+$15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[RDVC$^+$04] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are loss functions all the same? *Neural Comput.*, 16(5):1063–1076, May 2004.

[Red16] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, , last accessed on: 2019-03-23, 2013–2016.

[RF16] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[RHGS15] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[RHM86] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter A General Framework for Parallel Distributed Processing, pages 45–76. MIT Press, Cambridge, MA, USA, 1986.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[RHW88] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[RM86] David E Rumelhart and James L McClelland. Parallel distributed processing: explorations in the microstructure of cognition. volume 1. foundations. 1986.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[RPLvZ11] Stefan Romberg, Lluis Garcia Pueyo, Rainer Lienhart, and Roelof van Zwol. Scalable logo recognition in real-world images. In *ICMR*, 2011.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. cite arxiv:1609.04747Comment: 12 pages, 6 figures.

[Sam59] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[SC08] Ingo Steinwart and Andreas Christmann. *Support Vector Machines*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[SEZ$^+$] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann Lecun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *http://arxiv.org/abs/1312.6229*.

[SEZ$^+$13] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[SHA17] SAGAR SHARMA. Activation functions: Neural networks. `http://cs231n.github.io/convolutional-networks/`, last accessed on: 2019-03-23, 2017.

[SHK$^+$14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[SLJ$^+$14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[SLJ$^+$15] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 00, pages 1–9, June 2015.

[SSM$^+$16] Suraj Srinivas, Ravi Kiran Sarvadevabhatla, Konda Reddy Mopuri, Nikita Prabhu, Srinivas S. S. Kruthiventi, and R. Venkatesh Babu. A taxonomy of deep convolutional neural nets for computer vision. *CoRR*, abs/1601.06615, 2016.

[SZ14a] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[SZ14b] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[TXWE15] Cheng Tai, Tong Xiao, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization. *CoRR*, abs/1511.06067, 2015.

[TYRW14] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 1701–1708, 2014.

[USGS13] J. R. Uijlings, K. E. Sande, T. Gevers, and A. W. Smeulders. Selective search for object recognition. *Int. J. Comput. Vision*, 104(2):154–171, September 2013.

[vdSUGS11] K. E. A. van de Sande, J. R. R. Uijlings, T. Gevers, and A. W. M. Smeulders. Segmentation as selective search for object recognition. In *IEEE International Conference on Computer Vision*, 2011.

[Wil86] R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter The Logic of Activation Functions, pages 423–443. MIT Press, Cambridge, MA, USA, 1986.

[WM00] D Randall Wilson and Tony R Martinez. The inefficiency of batch training for large training sets. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 2, pages 113–117. IEEE, 2000.

[WZZ$^+$13] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[Zaw18] Jan Zawadzki. Convolutional neural networks for all | part ii. `https://medium.com/machine-learning-world/convolutional-neural-networks-for-all-part-ii-b4cb41d424fd?fbclid=IwAR3yX5fxtahPC1Gmry9Osj5YWNpQQQ8K9FTzCttcZCkMZSmkYA_sm899Ji0`, last accessed on: 2019-03-23, 2018.

[ZCHC17] H. Zhang, X. Cao, J. K. L. Ho, and T. W. S. Chow. Object-level video advertising: An optimization framework. *IEEE Transactions on Industrial Informatics*, 13(2):520–531, April 2017.

[Zha16] Joshua Z. Zhang. mxnetssd, implementation of ssd detector in mxnet. `https://github.com/zhreshold/mxnet-ssd`, last accessed on: 2019-03-23, 2016.

[Zuc03] Jean-Daniel Zucker. A grounded theory of abstraction in artificial intelligence. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 358 1435:1293–309, 2003.

[ZZXW18] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *CoRR*, abs/1807.05511, 2018.