DIPLOMA THESIS

Convolutional Neural Networks for Combined Classification and Detection of Distorted Traffic Signs

A Resource- and Performance-Efficient Approach Towards Hardware Implementations

Submitted at the Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Dr. Sai Manoj Pudukotai Dinakarrao Ph.D.

Institut of Computertechnik Technology (E384)
Vienna University of Technology

by

Martin Lechner BSc.
Matr.Nr. e1026059
Bienweg 1, 1220 Wien

Vienna, February 15, 2018

Abstract

Precise classification and detection of traffic signs in real-time is one of the non-trivial requirements for safe autonomous driving. Convolutional neural networks (CNNs) are a widely accepted concept for various kinds of image processing and machine vision tasks. Thus, it seems appropriate to use a CNN for detecting and classifying traffic signs. However, state-of-the-art CNNs for traffic sign detection though accurate are resource hungry due to their inherent structure with millions of parameters, and thus, not feasible to fit on low-end FPGA platforms.

This work shows that a much smaller, resource- and performance-efficient architecture can achieve a classification accuracy of 98.34% on real-world images. Therefore, the developed CNN has a simple, straight-forward architecture with only 60,000 weights and without using expensive computations like *Batch Normalization* or *Exponential Linear Units* (ELU). To reduce the time required for training such a network, step-size control is used as a way of managing the weight updates. Furthermore, this work shows that *Dropout* can improve the accuracy of a small-scale network, but the effect is smaller compared to larger architectures. Moving to binary weights increases the resource efficiency and allows an implementation on a ZedBoard with an accuracy of 96.53%. However, the simple network architecture in combination with the "detection by classification" approach limits the detection accuracy to approximately 80%.

The results demonstrate that it is not necessary to use a CNN with 1e6 and more weights to classify traffic signs with high accuracy. On the other hand, a reliable detection requires a more advanced network architecture. However, this work can be used as a starting point for developing efficient and reliable detection networks with a small hardware footprint to fit on low-cost FPGA platforms.

Kurzfassung

Eine zuverlässige Erkennung von Verkehrszeichen in Echtzeit ist eine der Grundvoraussetzungen für sicheres, autonomes Fahren. Künstliche neuronale Netze, sogenannte *Convolutional Neural Networks* (CNNs), werden erfolgreich für unterschiedlichste Aufgaben in den Bereichen Bildverarbeitung und Objekterkennung eingesetzt. Daher ist es naheliegend, ein CNN auch zur Erkennung von Verkehrszeichen einzusetzen. Aktuelle Umsetzungen erreichen zwar eine hohe Genauigkeit, können aber durch ihre schiere Größe mit mehreren Millionen Parametern nicht auf günstige FGPA Plattformen portiert werden.

In dieser Arbeit wird gezeigt, dass auch ein deutlich kleineres und in Bezug auf die benötigen Ressourcen effizienteres Netzwerk 98.34% der Verkehrszeichen richtig zuordnen kann. Das dazu entwickelte CNN hat einen einfachen Aufbau ohne Verzweigungen oder Schleifen und benötigt nur etwa 60,000 Parameter. Zusätzlich wird auf aufwendige Methoden wie *Batch Normalization* und exponentionelle Nichtlinearitäten (ELU) verzichtet. Um die nötige Trainingszeit des neuronalen Netzes zu reduzieren, werden die einzelnen Updateschritte überwacht und, falls notwendig, angepasst. Außerdem wird in dieser Arbeit der Einfluss von *Dropout* auf die erreichbare Klassifizierungsgenauigkeit kleiner Netzwerke analysiert. Es zeigt sich, dass ein positiver Effekt zwar vorhanden, die erzielbare Verbesserung aber schwächer ausgeprägt ist als bei größeren CNNs. Durch den Wechsel von Gleitkommaparametern zu binären Parametern kann das entwickelte neuronale Netz auch auf einem ZedBoard implementiert werden. Trotz der beträchtlich gesteigerten Ressourceneffizienz sinkt dieKlassifizierungsgenauigkeit nur um 1.81 Prozentpunkte auf 96.53%. Ein Nachteil des einfachen Aufbaus des Netzwerkes und des verwendeten „Erkennung durch Klassifizierung“ Ansatzes ist, dass in kompletten Bildern nur etwa 80% der Verkehrszeichen erkannt werden.

Die Ergebnisse zeigen, dass zur Klassifizierung von Verkehrszeichen mit hoher Genauigkeit kein Netzwerk mit über einer Million Parametern nötig ist. Andererseits zeigt sich auch, dass eine sichere und zuverlässige Erkennung eine komplexere Architektur oder deutlich mehr Parameter benötigt. Dennoch stellt diese Arbeit eine gute Basis für die zukünftige Entwicklung von zuverlässigen und gleichzeitig ressourceneffizienten neuronalen Netzen dar, welche auch auf kostengünstigen FPGA Plattformen implementiert werden können.

Table of Contents

Abbreviations

ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assistance Systems
AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
BLIS	Blind Spot Information System
BRAM	Block RAM
CNN	Convolutional Neural Network
DSP	Digital Signal Processing
ELU	Exponential Linear Unit
FF	Flip Flop
FLOPs	Floating Point Operations
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
HLS	High Level Synthesis
IoU	Intersection over Union
IP	Intellectual Property
LDW	Lane Departure Warning
LIDAR	Light Detection And Ranging
LUT	Look Up Table
mAP	mean Average Precision
MCDNN	Multi-Column Deep Neural Network
NAG	Nesterov Accelerated Gradient
PCA	Principal Component Analysis
PReLU	Parametric Rectified Linear Unit
QVGA	Quarter Video Graphics Array (Image/Video with a resolution of 320×240)
RADAR	Radio Detection And Ranging
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RPN	Region Proposal Network
RReLU	Randomized Rectified Linear Unit
SGD	Stochastic Gradient Decent
SoC	System on Chip
STE	Straight Through Estimator
SVM	Support Vector Machine
VGA	Video Graphics Array (Image/Video with a resolution of 640×480)

1 Introduction

We live in an era where the way we drive cars is about to change. Since the first day of power-driven vehicles, humans were the only decision-making instance in a car until a few years ago, the first advanced driver assistance systems (ADAS) had been developed. Well-known examples are light and rain sensors. In contrast to current developments, they were more comfort than safety oriented. However, they still take control away from the driver. More recent systems like adaptive cruise control (ACC), blind spot information system (BLIS), lane departure warning system (LDW), driver monitoring system, collision avoidance system, and intersection assistant help the driver in potentially dangerous situations. According to various statistics [16, 22], the market for ADAS is growing rapidly. Thus, the cars are taking more and more control eventually leading to fully autonomous vehicles. With Tesla in the leading position, many manufacturers invest in such self-driving cars. To be aware of the surrounding environment, a lot of different sensors like LIDAR, RADAR, and cameras are utilized. In combination with advanced image processing techniques, cameras can detect traffic signs or recognize pedestrians around the car. Traffic signs play a critical role in our transportation systems informing the driver (or the car) about upcoming limitations and dangers. For safe autonomous driving, the reliable detection and classification of traffic signs plays an important role to avoid, for example, dangerous situations near intersections.

On the other hand, artificial intelligence (AI) is currently gaining a lot of attention. Regarding machine vision tasks, neural networks, especially convolutional neural networks (CNNs), play an important role. The history of artificial neural networks goes back to the experiments of Hubel and Wiesel in 1959 [HW59]. However, it took 39 years before LeCun presented LeNet, a rather small CNN for handwritten character recognition [LBBH98]. But convolutional neural networks were still not accepted. It took another 14 years until AlexNet [KSH12] won the ImageNet classification challenge, a major competition for visual computing. Since then, CNNs were used very successfully for different tasks in visual computing. Thus, it seems to be logical to apply convolutional neural networks for recognizing and classifying traffic signs in context with autonomous driving.

1.1 Motivation

Enhanced capabilities of current hardware platforms, the hardware-software co-design and the data processing techniques in embedded systems led to an increased interest in the design of intelligent real-time systems. Advanced driver assistance systems is one of the fields where an

intelligent processing of data within a short time and a high accuracy is crucial. When using cameras, object detection and classification targeting, among others, traffic signs [CCWY16, CMMS12, SGH15, SL11, ZYZ+17], road surface signs [QLY+16], and pedestrians [AKV+15, RMNM16] is gaining attention in such systems to ensure safe driving of autonomous cars.

Traffic sign detection and classification is a challenging computer vision problem typically performed based on the canonical structural features like size, shape, and color [CWHW10, KGEU08, LYH+13]. The main problems addressed are perspective changes and illumination variations such as differing lighting conditions (cloudy weather, night, heavy light, and so on). Furthermore, distortions due to human-made or natural activities (e.g., tilted, dirty or bleached boards) make the problem of traffic sign detection and classification even more complex. Another problem is that most of the used features are hand crafted. Thus, adding more traffic signs requires extra engineering work to develop the new features required for detecting and classifying the additional signs.

Nowadays, it is common to use deep learning approaches like convolutional neural networks for traffic sign detection [CMMS12, ZYZ+17]. Once, a network is designed, retraining with a dataset containing new traffic signs is enough to adapt it to the additional sign. In the worst case of a couple of new traffic signs, it may be required to increase the size of the network. However, both tasks do not require any costly re-engineering.

Current software implementations of neural networks either lack in performance [SEZ+13] or need a lot of computational resources [CCWY16, RF16] leading to a higher power consumption and a large packaging. Typically, general purpose object detection frameworks can reach between 5 frames per second (fps) [RHGS15] and 45 fps [RDGF15] on high-end GPUs. For example, a typical computing GPU, the Nvidia Tesla K20, measures $26.7cm \times 11cm \times 3.9cm$ and requires 225W. It does not require additional explanations that such configurations are not practical nor efficient to use for real-time situations in embedded systems with limited resources.

Specialized hardware accelerators allow massive parallelization and ensure that real-time identification and classification of (distorted) traffic signs is possible. Among various hardware platforms, FPGAs suit the best because of its parallelization, reconfigurability, and low power utilization properties [FBCS12, AMA13]. Also, they are much cheaper, require less design time, and less workforce than custom ASIC designs. As FPGAs have limited resources (area, processing capabilities, and memory), current state-of-the-art classification [HZRS15a, SZ14, SLJ+14] and detection [RF16, RHGS15] networks with millions of full-precision parameters and complex architectures do not fit low-cost FPGA platforms. Implementing a convolutional neural network a cheap FPGA requires a simple, straightforward, and efficient architecture with a small number of parameters.

1.2 Problem Statement

The goal of this work is to develop a combined traffic sign classification and detection algorithm based on state-of-the-art machine learning methods, namely convolutional neural networks. Being aware of a future FPGA implementation, the network has to be organized in a simple feed-forward way without any shortcut connections. Due to different latencies of parallel paths, additional buffers would be required limiting the overall performance of the system. If possible, the final architecture should not rely on computational expensive functions and full-precision datatypes. Both require a lot of extra hardware resources and increase the runtime of the system. This eventually leads to a larger design and a lower framerate. FPGAs are also limited in terms of

on-chip memory. Thus, the neural network has to contain fewer weights than standard GPU based implementations. Another way to save memory is using binary weights and eventually also binary activations.

Drastically reducing the size of a network introduces various challenges. Some state-of-the-art methods like *Batch Normalization* require computing multiple square roots and other complex functions which is against the objective of keeping the network simple. Other improvements like *Dropout* reduce the active size of a CNN which can be a problem when using small-scale networks.

Most current object detection architectures are designed for winning different competitions. Thus, the main focus of such networks is predicting a bounding-box (a bounding box is a good metric to compare different solutions) around the detected object as accurate as possible. Detecting traffic signs in the context of autonomous driving does not require the exact position of a traffic sign inside an image. A self-driving car only needs to know what traffic signs are present in the image, not where they are. However, in some special scenarios were a sign of a nearby road is visible from the current location, the position inside the image matters. Since excluding such signs requires more advanced methods and further image processing, these rare scenarios are not addressed in this work.

The process of developing hardware-based neural networks can be divided into three subtasks

- **Develop a hardware-friendly architecture:** This includes reducing the number of parameters, eliminating the use of computational expensive functions and organizing the network in a strict feed-forward way without any shortcuts.
- **Minimize the required memory:** The necessary memory can be reduced by limiting the accuracy of the datatypes down to one bit. This results can in binary weight networks or in full binary networks, i.e., networks with binary weights and binary activations.
- **Implement the CNN on an FPGA**

The objective of this work is developing a simple architecture with a small memory footprint similar to the subtasks one and two. After that, each part of the network should be simulated and synthesized using High-Level Synthesis (HLS) to obtain an approximation for an implementation on an FPGA.

1.3 Main Contributions

The main contributions of this work can be outlined as follows:

- Development of a resource- and performance-efficient CNN-based Traffic Sign Detection with only ≈60,000 weights with a classification accuracy above 98%
- Analysis of the efficiency of *Dropout* in small-scale CNNs.
- Implementation of step-size control as a way to improve the training process.
- Simulation of the CNN in hardware using High-Level Synthesis and generalization of the synthesis results (hardware footprint) for arbitrary sized layers.

2 State-of-the-Art

2.1 Convolutional Neural Networks

Convolutional neural networks are widely used for different types of image processing. Two of the most famous application cases are image classification and object detection. However, this section starts with an introduction into CNNs and their main building blocks together with an overview of recent developments. This section will also present various methods to improve the training efficiency and some of the most important datasets.

2.1.1 Main Building Blocks

The design of convolutional neural networks typically follows a layer-wise structure. The following sections give a brief overview of the most important layers to present current developments an to introduce some naming conventions used in this thesis. Each layer accepts an input volume of $W_{in} \times H_{in} \times D_{in}$ in and produces an output volume of $W_{out} \times H_{out} \times D_{out}$ where W and H represent the spatial size, and D represents the depth of a volume. In case of an input image, W and H are the width and the height and D is the number of color channels, e.g., three for RGB color images and one for gray-scale images.

2.1.1.1 Convolutional Layers

Convolutional layers are the fundament of any CNN. They produce an output volume by convolving K 3-dimensional filters of size $F \times F \times D_{in}$ spatially over the input volume computing dot products. The stride S controls the amount of pixels the filters are moved at a time. Finally, zero-padding with a width of P can be used to prevent filters from smoothing the border sections too much and to control the size of the output volume. It is convenient to use zero-padding in a way that the output volume has the same spatial extent as the input volume having only pooling layers in the network changing the spatial size during the forward pass. Figure 2.1 and Equation 2.1 illustrate the relationship of the hyperparameters stride, filter size and pad.

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1, \qquad H_{out} = \frac{H_{in} - F + 2P}{S} + 1, \qquad D_{out} = K \tag{2.1}$$

It is important to note that specific set of hyperparameters in considered to be invalid if the result of one of the equations above is not an integer.

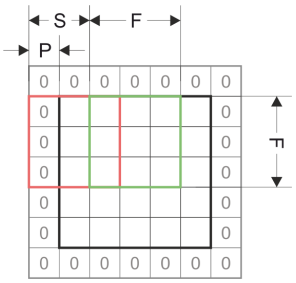


Figure 2.1: Single convolutional layer with convolutional parameters filter size F, stride S, and pad P

2.1.1.2 Fully-Connected Layers

Fully-connected layers, also called affine layers, are the core building block of any standard neural network, and they are widely used in convolutional networks too. Usually, a series of such layers follows after the convolutional layers to do the classification job based on the features computed by the conv-layers. As the name states, the neurons in fc-layers are connected to all activations coming from the previous layer. Some of the most recent papers, however, replaced them with convolutional layers for increased flexibility. See section 2.1.1.5 for more details.

2.1.1.3 Activation Functions

Activation functions add a non-linearity to a network. A full linear network without non-linear functions and with one output can be reduced to a single node being not able to express a complicated function. Figure 2.2 shows a simple, two-layer network and a single neuron both computing the same function as described by Equation 2.2. It is easy to prove, that such a transformation exists for every linear network independent of the size of the network.

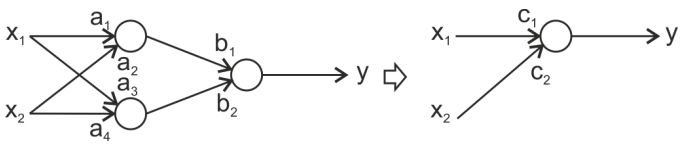


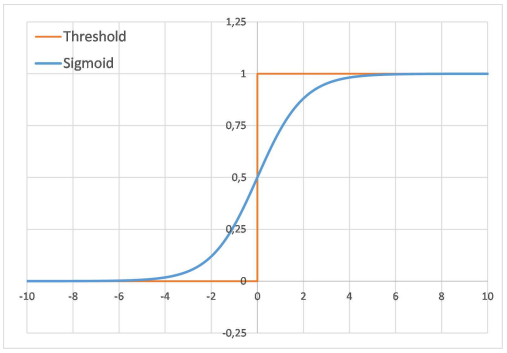
Figure 2.2: Simple, two layer network with three neurons and an equal single neuron

$$y = b_1(a_1x_1 + a_2x_2) + b_2(a_3x_1 + a_4x_2) = \underbrace{(b_1a_1 + b_2a_3)}_{c_1}x_1 + \underbrace{(b_1a_2 + b_2a_4)}_{c_2}x_2 \tag{2.2}$$

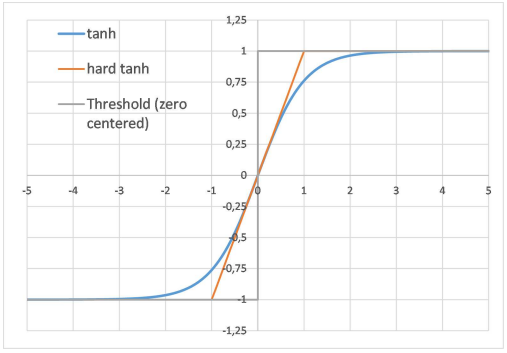
An activation function acts on each element of a volume individually computing a fixed mathematical relation. Historically, the sigmoid function was a common choice, but it is barely used anymore. The main reasons are saturation for very large and small input values (which kills the gradient) and the fact that the output is not zero-centered (which can lead to problems during training). A more detailed explanation on the drawbacks is presented in [10]. The hyperbolic tangent function resolves the latter issue but still saturates for increasing input values. Currently, the Rectified Linear Unit (ReLU) is the most popular activation function accelerating the convergence during training. Also, this function does not need expensive computations like the exponential function. However, ReLUs have the undesirable property that they can die. That happens when the weights of a previous convolutional or fully-connected layer update during

training in a way that the output of the neuron is zero for all inputs. Again, [10] gives more details on that.

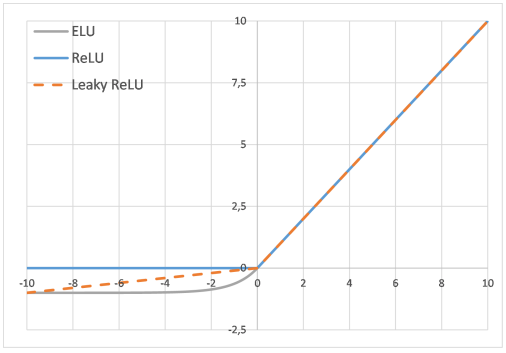
There are also some improved versions claiming to fix the dying ReLU problem and further accelerating the convergence. Leaky ReLU has a small negative slope for values $x < 0$ instead of being always zero and in Parametric ReLU (PReLU), the steepness of the negative slope is a trainable parameter. Another recent variant of the ReLU activation is the Exponential Linear Unit where the kink at $x = 0$ is smoothed using the exponential function for $x < 0$ [CUH15]. Xu et al. discuss different versions of the ReLU function in [XWCL15]. They also compare the convergence curves while training on the CIFAR dataset (see section 2.1.3.2). Finding the best activation function is still a very active area of research. In another paper, Xu et al. describe modifications to improve the performance of sigmoid and hyperbolic tangent activation functions [XHL16]. Figure 2.3 gives an overview of some of the most important activation functions, and Table 2.1 shows the corresponding mathematical expressions.



(a) Sigmoid and simple threshold



(b) tanh, hard tanh, and zero-centered threshold



(c) ReLU, leaky ReLU and ELU

Figure 2.3: Plots of commonly used activation functions

There is also another problem with some of the activation functions. For backpropagation, it is necessary to know the gradient of a non-linearity over the full range of possible input values. Some functions are not differentiable at every point (all functions with a kink) while others have a gradient of zero over the entire range except one point (threshold functions). To solve the first issue, it is enough to define the derivative at those points. A simple solution for the second problem is presented in [CB16]. They use the sign function without changes during the forward pass and a hard-tanh shaped gradient during the backward pass.

$\sigma(x) = \frac{1}{1+e^{-x}}$	(Sigmoid)
$\tanh(x) = 2\sigma(2x) - 1$	(tanh)
$\max(-1, \min(1, x))$	(hard tanh)
$\max(0, x)$	(ReLU)
$\max(cx, x) / \max(\alpha x, x)$	(Leaky ReLU / PReLU)
$c(e^x - 1)$ if $x < 0$, $\quad x$ otherwise	(ELU)

Table 2.1: Different activation functions with c being a constant with $c > 0$ and α being a trainable parameter

2.1.1.4 Pooling Layers

Pooling is a common technique to reduce the special size (W and H) of volumes in CNNs helping to trim the number of parameters. Many state-of-the-art networks use pooling layers in regular intervals between convolutional layers. Pooling layers operate on each depth slice individually in a sliding window fashion. The most common versions are maximum pooling and average pooling where the output is the maximum value or the average value of a window, respectively. Because pooling is very aggressive in discarding information, it is common to use only small sizes in practice. Thus, 2x2 maximum pooling is the most widespread one. Table 2.2 gives an overview of different sizes and the amount of discarded information. It is also possible to use a stride smaller than the pooling size which is sometimes referred to as overlapped pooling.

Pooling size	Stride	Discarded activations
2x2	2	75%
3x3	3	88.89%
4x4	4	93.75%

Table 2.2: Amount of discarded activations depending on the pooling size

The output size of any pooling layer can be calculated using Equation 2.3 with the hyperparameters S (stride) and F (spatial extent). Again, pooling is only possible if the results are all integers.

$$W_{out} = \frac{W_{in} - F}{S} + 1, \qquad H_{out} = \frac{H_{in} - F}{S} + 1, \qquad D_{out} = K \tag{2.3}$$

2.1.1.5 Conversions Between Fully-Connected and Convolutional Layers

Both, convolutional layers and fully-connected layers are based on computing dot products. The only difference is the scope of the filter, i.e., the weights. While the neurons in convolutional layers look at local regions of an input image, the neurons in fully-connected layers look at

the whole image at once. As a consequence, any convolutional layer can be represented by an equivalent fully-connected layer and any fully-connected layer by a corresponding convolutional layer. Figure 2.4 illustrates the first type of conversion by a small convolution layer with a single filter of size $F = 3$. Even in this small example the amount of weights increases by a factor of 25 (225 weights instead of 9). More generally, the necessary number of weights is $N_{conv} = F^2$ for a convolutional layer compared to $N_{fc} = F^2 \cdot H \cdot W$ for an equivalent fully-connected layer. Therefore, this conversion is not useful in practice. This example also demonstrates the increased (memory-) efficiency of convolutional layers due to parameter sharing and local connectivity. The corresponding weight matrix of the fully-connected layer consist of a lot of zeros and every line contains only the same 9 weights at various positions.

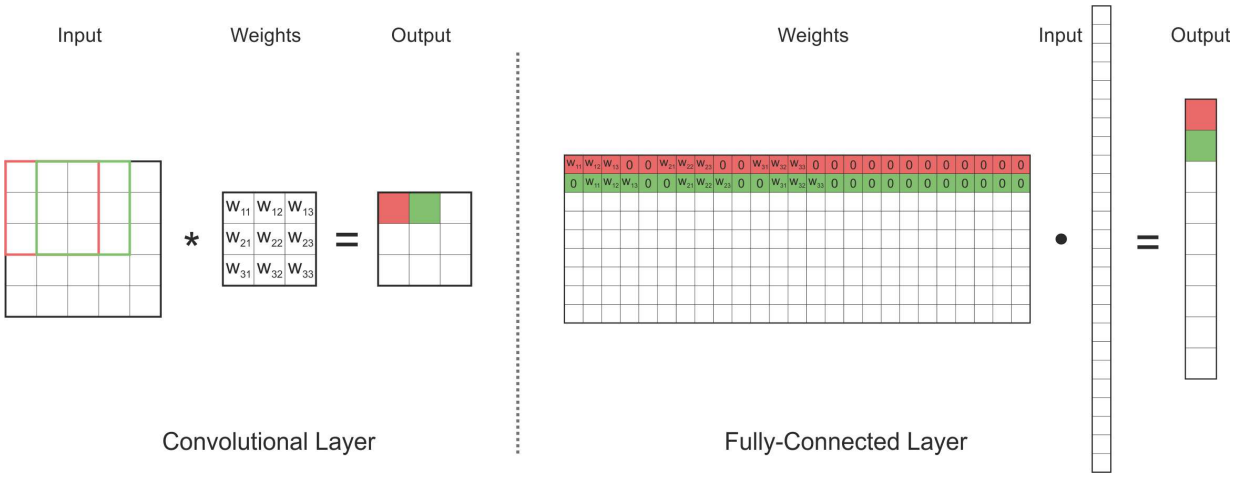


Figure 2.4: Conversion from a convolutional-layer to a fully connected layer

However, the conversion in the other direction has significance in some applications. It can be used to increase the efficiency of the sliding window technique for multi-position classification and localization as described in section 2.2.2. The fully-connected layer in the right section of Figure 2.4 is equal to the convolutional layer sketched in Figure 2.5. This representation is different to the one in the left sub-image of Figure 2.4 because it is universally valid for any set of weights. The conversion does not involve a change in the number of weights. It requires only some simple reshaping operations performed on the weights as well as on the input and the output volume. To convert any fully-connected layer into a convolutional layer, the convolutional parameters have to be set according to the following rules:

- A filter size F equal to the size of the input. Therefore, the input has to be square for most network architectures.
- A number of filters K equal to the number of neurons in the fully-connected layer.
- No zero padding, i.e., $P = 0$
- A stride S of 1 (in fact, choosing a different stride should not change anything as the weights of the filter are connected to the full input).

Such conversions are also possible for fully-connected layers with one-dimensional inputs when using, for example, the output of Figure 2.5 (1x1x9) as input to the next layer which then performs a spatial 1x1 convolution. A comprehensive explanation of this transformation can be found in [8].

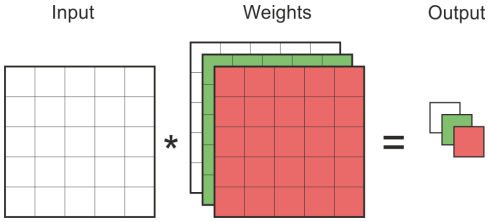


Figure 2.5: Convolutional layer equivalent to the fully-connected layer in Figure 2.4 (right sub-image) using the same coloring scheme

2.1.1.6 Loss Functions

Loss functions are a crucial part of a neural network. Simply speaking, they measure the dissatisfaction of a network with its results, e.g. the class scores s. There are different loss functions available, depending on the purpose of a network. Because this work mainly focuses on classification and classification based detection, this section presents loss functions for classification only. The second big group, loss functions for regression as used for example for predicting bounding boxes, are not covered.

In general, the total loss L of a network can be calculated using Equation 2.4 with the number of training examples N, the loss of each sample L_i, the regularization loss $R(W)$, and the hyperparameter regularization strength λ. The first term is often called data loss.

$$L = \frac{1}{N}\sum_{i=1}^{N}(L_i) + \lambda R(W) \tag{2.4}$$

Next to the L1 and the L2 classifier (using the L1 and the L2 loss, respectively), the most common classifiers used in practice are the SVM classifier (using the hinge loss) and the Softmax classifier (using the cross-entropy loss). The different loss functions are summarized in Table 2.3.

$$L_i = \sum_j |s_j - s_{y_i}| \qquad \text{(L1 loss)}$$

$$L_i = \sum_j (s_j - s_{y_i})^2 \qquad \text{(L2 loss)}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \qquad \text{(hinge loss)}$$

$$L_i = -log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \qquad \text{(cross-entropy loss)}$$

Table 2.3: Different loss functions with the score s_j of class j, the correct class y_i, the score of the correct class s_{y_i}, and $j = 1, ..., C$ for C classes

The Softmax function, as shown in Equation 2.5, takes a real-valued, K-dimensional vector z and squashes it to a K-dimensional vector $f(z)$ with values in the range $[0, 1]$, so that they sum

up to 1. From a probabilistic point of view, the Softmax function interprets the values of z as unnormalized log probabilities and transforms them to normalized class probabilities. This characteristic is not only useful for computing the loss but also helps to interpret the class scores at test time.

$$f_j(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{with} \quad j = 1, ..., K \tag{2.5}$$

An interesting and very recent paper on loss functions for classification can be found in [JC17]. The authors compare how different loss functions affect the learning dynamics of neural networks and the robustness of the classifiers to diverse effects. For more details and intuitions on loss functions, especially the SVM and the Softmax classifier, and the regularization loss, refer to [11] and [9].

2.1.2 Techniques to Improve Learning

Training (convolutional) neural networks can be a tricky task and involves tuning several parameters to converge the loss successfully. Even with an assembly of state of the art graphics cards, it can take up to several weeks depending on the size of the network, the activation function, and the dataset.

Another problem is overfitting. The network fits the noise in the training data or precisely the training data instead of generalizing the underlying pattern. When a network achieves a high training accuracy together with a comparable low validation accuracy, this is a clear indicator that overfitting occurred. Thus, several techniques have been proposed to accelerate the training process and to prevent overfitting. The following sections present some of the most important and widely accepted techniques for that purpose.

2.1.2.1 Update Rules

Update rules define how to update the weights during backpropagation based on the gradient. The most straightforward update rule is Stochastic Gradient Descent (SGD) where the weights are changed in the negative gradient direction multiplied by a learning rate η. Table 2.4 gives an overview of different versions of SGD. Note that the equations in this section are written in a way how they would be implemented in practice. A more formal notation together with more in-depth explanations of many different update rules can be found in [Rud16]. SGD has some problems when the gradient is very steep in one direction and shallow in the other one. A high learning rate will cause an overshoot in the steep direction, and with a low learning rate, the network will barely learn in the shallow direction. Adding a momentum term ($\mu \cdot v$) to SGD can reduce this issue and increases the converge rate. The velocity v builds up in shallow directions regularized by the momentum parameter μ, which is often set to values around 0.9. In literature, this approach is simply called SGD with momentum. A further improvement is the so-called Nesterov Accelerated Gradient (NAG). The idea behind this is that the update step caused by the momentum term is made anyway independent of the actual gradient. Hence it should be better to calculate the gradient starting from the end-point of the momentum step. Figure 2.6 (figure redrawn from [12]) illustrates this intuition.

$$W = W - \eta \cdot dW \qquad \text{(SGD)}$$

$$\begin{aligned} v &= \mu \cdot v - \eta \cdot dW \\ W &= W + v \end{aligned} \qquad \text{(SGD with momentum)}$$

$$\begin{aligned} v_{old} &= v \\ v &= \mu \cdot v - \eta \cdot dW \\ W &= -\mu \cdot v_{old} + (1 + \mu) \cdot v \end{aligned} \qquad \text{(NAG)}$$

Table 2.4: Different versions of stochastic gradient descent with weights W and a gradient dW

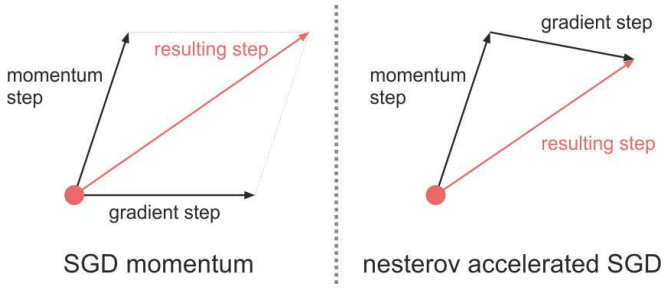


Figure 2.6: Intuition behind Nesterov Accelerated SGD

All these versions of SGD have in common that they use the same learning rate for all parameters. Adaptive methods like RMSprop [TH12] can tune the learning rate for each parameter individually (Equation 2.6). The cache variable accumulates past gradients in a leaky way (decay rate γ). This ensures that the effective learning rate is reduced for high gradients, e.g., steep gradients but avoids the cache variable to reach high values which would reduce the effective learning rate to zero over time . Kingma et al. present an improved version called ADAM adding a momentum term in [KB14].

$$\begin{aligned} cache &= \gamma \cdot cache + (1 - \gamma) \cdot dW^2 \\ W &= W - \frac{\eta \cdot dW}{(\sqrt{cache} + \epsilon)} \end{aligned} \tag{2.6}$$

Of course, the list given above is not exhausting. Second order methods, for example, use the previous gradient to estimate the curvature of the loss surface for faster convergence. However, these methods require a lot of memory during training. It is also very common to decay the learning rate during training after a fixed number of epochs or when the training progress stops, i.e. the loss function reaches a plateau.

2.1.2.2 Dropout

Dropout, as presented in [SHK+14], is a simple and effective technique to prevent, particularly wide and deep, neural networks from overfitting. It also increases the classification accuracy and leads to sparse activations. The idea is to drop neurons with a probability p on every forward pass during training and to keep all neurons active while testing (Figure 2.7). This reduces the size and consequently the capacity of the network which lowers the chance of overfitting.

For high accuracy networks, it is common to train an ensemble of equal CNNs with randomly initialized weights and to combine them for testing. This often increases the accuracy by one or two percent. However, training multiple networks can be very time-consuming, especially if the network and/or the dataset are big. Because dropout samples a new set of neurons for each forward pass, the final network can be interpreted as an ensemble of a lot of smaller networks. Another way to explain the increased accuracy is the fact that the features have to be more robust, i.e., the classification cannot rely on specific features. As a drawback, the layers have to be more extensive, that is they have to contain more neurons so that the resulting network can still fit the training data.

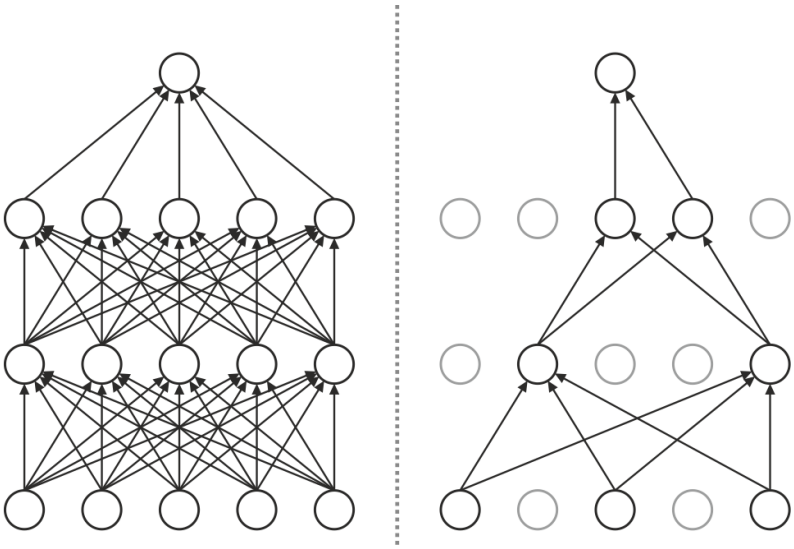


Figure 2.7: Effect of dropout on a network with two hidden layers. Network without dropout (left) and with a dropout of $p = 0.5$ (right). Dropped neurons are displayed in light gray. (Figure redrawn from [SHK+14]

At test time, the activations have to be scaled down according to p to ensure that the expected output and the actual output are the same. In case of dropout with $p = 0.5$, all activations have to be divided by a factor of 2. A simple example using a single neuron with two inputs (x_1, x_2), two weights (w_1, w_2) and one output (y) helps to explain the reasons behind rescaling. Without dropout, the neuron would calculate the output as written in Equation 2.7. This is also true at test time since dropout is only applied at training time.

$$y = x_1 w_1 + x_2 w_2 \tag{2.7}$$

At training time, however, the expected value of y is calculated using Equation 2.8 having four different cases: Both inputs get dropped, only x_1 gets dropped, only x_2 gets dropped, and no input gets dropped.

$$\begin{aligned} E(y) &= \frac{1}{4}\left[0w_1 + 0w_2 + 0w_1 + x_2 w_2 + x_1 w_1 + 0w_2 + x_1 w_1 + x_2 w_2\right] \\ &= \frac{1}{2}\left[x_1 w_1 + x_2 w_2\right] \end{aligned} \tag{2.8}$$

Because y and $E(y)$ are different, the activation (which is equal to the output of the neuron y) has to be downscaled to ensure $y = E(y)$. In practice, it is more convenient to leave the

activations untouched at test time and to rescale them during training instead, i.e., to increase the activations by $1/p$. This technique is called inverse dropout.

2.1.2.3 Weight Initialization

A proper initialization of neural networks is vital for the loss function to converge. While a standard initialization with small random numbers sampled from a normal distribution with zero mean and a variance of 1 works well for shallow networks, deeper networks hardly train. As shown in [GB10], the activations tend to get smaller after each layer, i.e., the variance of the distribution of the activations is shrinking. During the forward pass, this is not a big problem, but during the backward pass, the gradient also gets multiplied with the weights eventually leading to tiny gradients for the first layers. Thus, the network will barely learn. In [GB10], the authors came up with the idea of scaling the variance of the normal distribution for weight initialization according to the number of inputs and outputs of the layer. With a single layer l computing $y^l = \sum_{i=1}^{n}(x_i^l w_i^l) + b^l$ with a weight vector w^l and a bias $b^l = 0$, a better initialization strategy can be derived as following:

$$\begin{aligned} Var(y^l) &= Var(\sum_{i=1}^{n^l} x_i^l w_i^l) = \sum_{i=1}^{n^l} Var(x_i^l w_i^l) \\ &= \sum_{i=1}^{n^l} \underbrace{\Big([E(x_i^l)]^2 Var(w_i^l) + [E(w_i^l)]^2 Var(x_i^l) + Var(x_i^l)Var(w_i^l)\Big)}_{\text{assuming zero mean of } x\text{, which is not true for ReLU-shaped inputs}} \\ &= \sum_{i=1}^{n^l} Var(x_i^l)Var(w_i^l) = n^l Var(w^l)Var(x^l) \end{aligned} \tag{2.9}$$

To ensure that $Var(y^l) = Var(x^l)$, $Var(w^l)$ has to be set to $1/n^l$. Thus, when sampling the initial weights from a unit Gaussian, they have to be scaled by $\sqrt{1/n^l}$.

With taking the forward and the backward pass into account, Glorot et al. came to a similar result scaling the weights by $\sqrt{2/(n^l + n^{l+1})}$ [GB10]. In literature, this strategy is often referred to as *Xavier initialization*. However, He et al. discovered that this strategy does not work well with ReLU non-linearities [HZRS15b]. In the end, they came up with an condition for setting the variance for a layer l with ReLU-shaped inputs (Equation 2.10) leading to a scaling factor of $\sqrt{2/n^l}$.

$$\frac{1}{2} n^l Var(w^l) = 1, \qquad \forall l \tag{2.10}$$

Other methods like [KDDD15] try to find an optimal initialization using an iterative approach. First, they draw weights from a unit Gaussian, and then they tune the weights based on the outputs of the corresponding layer. However, such approaches are more complex and currently not widely accepted.

2.1.2.4 Batch Normalization

Batch Normalization [IS15] is a method to normalize the activations after each affine or convolutional layer. Inspired by the different weight initializations methods trying to retain Gaussian-like inputs to all layers, the basic idea is to simply force the activations to be Gaussian. For that purpose, they added Batch Normalization layers computing the function in Equation 2.11 where $\hat{x}^{l,(k)}$ is the normalized version of $x^{l,(k)}$ along each dimension (k) of the layer l. Because a simple normalization can constrain the following non-linearity too much, they added a set of learnable parameters $\gamma^{l,(k)}, \beta^{l,(k)}$. The final transformation is given in Equation 2.12. With those parameters, the network can learn to undo the normalization by setting $\gamma^{l,(k)} = \sqrt{Var(x^{l,(k)})}$ and $\beta^{l,(k)} = E(x^{l,(k)})$. As a drawback, Batch Normalization adds additional complexity to the algorithm in both, the forward and the backward pass, thus lowering the performance at test time as well as the required training time.

$$\hat{x}^{l,(k)} = \frac{x^{l,(k)} - E(x^{l,(k)})}{\sqrt{Var(x^{l,(k)})}} \tag{2.11}$$

$$y^{l,(k)} = \gamma^{l,(k)} \hat{x}^{l,(k)} + \beta^{l,(k)} \tag{2.12}$$

2.1.3 Popular Datasets

This section presents some of the most valuable datasets for evaluating convolutional neural networks and other image-based machine learning algorithms in different disciplines like classification, localization, and detection. This list is by far not exhaustive, but it includes the majority of datasets used by the papers mentioned in this thesis.

2.1.3.1 ImageNet

ImageNet [21] is a significant and widely used dataset for image classification, object localization, and object detection containing more than 1.2 million images. It is organized according to the WordNet hierarchy which is a large database of English words (nouns, verbs, adjectives, and adverbs) grouped into sets of cognitive synonyms called synsets [19]. ImageNet aims to have an average of 1,000 images for each synset. They also host an annual challenge, the Image Net Large Scale Visual Recognition Challenge [RDS+15]. The size of the datasets for those challenges varies depending on the task and the year, but this year the detection training set contains 50,000 images sampled from a set of 1,000 classes [7].

2.1.3.2 CIFAR

The two CIFAR datasets are subsets of the larger "80 million tiny images" dataset [13] [14] created for image classification only. The CIFAR-10 dataset consists of 60,000 color images (50,000 training and 10,000 test images) with a resolution of 32x32 pixels in 10 classes. Following this structure, the CIFAR-100 dataset contains the same amount of images equally split up in 100 categories.

2.1.3.3 Pascal

The Pascal Visual Object Classes (VOC) dataset [17] is an image dataset built for classification and object detection tasks. Depending on the year of release, there are also different challenges like segmentation, person layout (individual labeling parts of the body – head, hand, and feet) and action classification (jumping, walking, and many more) available. The latest version features 20 classes with a total number of 11,530 training images together with 27,450 annotated objects and 6,929 segmentations. Everingham et al. give a deeper insight into this dataset providing more details about the creation process in [EVGW+10].

2.1.3.4 COCO

COCO, short for Common Object in Context, is a large dataset mainly for object detection, segmentation and captioning developed by Microsoft Research [4]. It features more than 200,000 labeled images with 80 object categories. Like others, they host annual challenges in different visual tasks. There is also a paper [LMB+14] available giving an in-depth description of the dataset.

2.1.3.5 Traffic Sign Datasets

The two most relevant traffic sign databases in central Europe are the German Traffic Sign Benchmark [20] and the BelgiumTS Dataset [23]. The German Traffic Sign Benchmark features two datasets, one for classification, the German Traffic Sign Recognition Benchmark (GTSRB) [SSSI12], and one for detection, the German Traffic Sign Detection Benchmark (GTSDB) [HSS+13]. The GTSRB contains over 50,000 images taken in the wild and therefore offering a broad range from nearly perfectly visible signs to heavily disturbed, tilted and dirty signboards. The GTSDB includes 900 high-resolution images with a variety of different scenes and locations like highways, cities and country roads. Similarly, the BelgiumTS dataset also consists of images recorded in the wild offering traffic sings in various angles and lighting conditions.

2.2 Convolutional Neural Networks for Classification, Localization, and Detection

The main building blocks presented above can be combined in various ways to form fully functional convolutional neural networks for a wide range of application cases. It is important to differentiate between different tasks in computer vision as described in [SEZ+13]. Classification is the simplest one describing the process of assigning a class label (out of a discrete set of classes) to an input image. Localization is an extension of classification with an additional output describing the position of the classified object inside the image with a bounding box. Usually, images in localization datasets contain exactly one object but a greater amount is also possible. However, such networks can only produce a fixed number of guesses (class label and position) no matter how many objects are present in a specific scene. Object detection removes this limitation being, in theory, able to detect any amount of objects in an image, including zero objects. Taking these concepts one step further leads to object segmentation where more accurate outlines replace

bounding boxes. A nice overview of the most important network architectures for classification and detection is given in [5].

This section starts with presenting different architectures of convolutional neural networks for classification, localization and detection (Sections 2.2.1 to 2.2.3). After that, a short overview of binary neural networks and FPGA-based implementations is given in Sections 2.2.4 and 2.2.5. For convenience, the network details like the number of weights and layers are summarized in Section 2.4.

2.2.1 CNNs for Classification

In 2012, A. Krizhevsky et al. developed the first CNN winning a major image classification challenge, the ILSVRC 2012 [KSH12] [2]. Since then this network is often referred to as AlexNet, and still, a couple of state-of-the-art algorithms use it as an underlying structure. The network features a total of 11 layers (five convolutional, three pooling and three fully-connected layers). They also the exchanged the tanh activations with ReLUs for faster training and applied dropout to prevent overfitting. In the following years, CNNs started to dominate all classification challenges. One year later, Zeiler et al. released a paper describing how to visualize and understand neural networks based on AlexNet by using so-called deconvolution networks [ZF13]. This approach allows to pass feature maps from inside the network back to the input pixel space and to visualize them in a for humans understandable way. The authors also tweaked AlexNet a bit to boost its performance. They changed the filter size in the input layer and added more filters to the third, fourth and fifth convolutional layer. This network is often called ZF-Net. By removing some layers and testing the performance of the remaining network, they found out, that the overall depth of a network matters a lot. With this in mind, Simonyan et al. developed VGG-Net, a much deeper but simpler convolutional network [SZ14]. The best performing network, VGG-16, features 16 convolutional layers using only filters of size 3x3 and 2x2 maximum pooling. AlexNet, for example, uses different filter sizes (11x11, 5x5 and 3x3) and overlapped pooling (3x3 with a stride of 2).

All three networks presented above share the same underlying architecture. They consist of a series of convolutional layers (followed by an activation function) with pooling layer in-between at irregular intervals and some fully-connected layers at the end to compute class scores. With GoogLeNet, Szegedy et al. introduced a new type of architecture with several convolutional layers in parallel [SLJ+14]. They call such clusters of convolutional and pooling layers inception module. Figure 2.8 shows the naïve version in the left sub-image. GoogLeNet consists of several stacked inception modules with an overall depth of 27 layers (22 convolutional layers) and a total amount of about 100 individual layers. In 2015, He et al. presented a much deeper network with a different architecture called ResNet [HZRS15a]. However, they found out that deeper networks can have a higher training error than shallower networks due to the increased complexity. It seems that current optimization methods cannot handle very deep networks. To overcome this issue, the paper introduces shortcut connections. In this work, a shortcut is a simple identity mapping as illustrated in the right section of Figure 2.8. The authors assume that such an architecture is easier to optimize than a traditional one. The best performing network, ResNet-152, has 152 weight layers with a computational complexity lower than VGG16 (11.3 billion FLOPs for ResNet-152 vs. 15.3 FLOPs for VGG16), therefore being very efficient. It is also possible to combine inceptions modules and ResNets. Szegedy et al. presented the latest version, Inception-v4, in [SIV16].

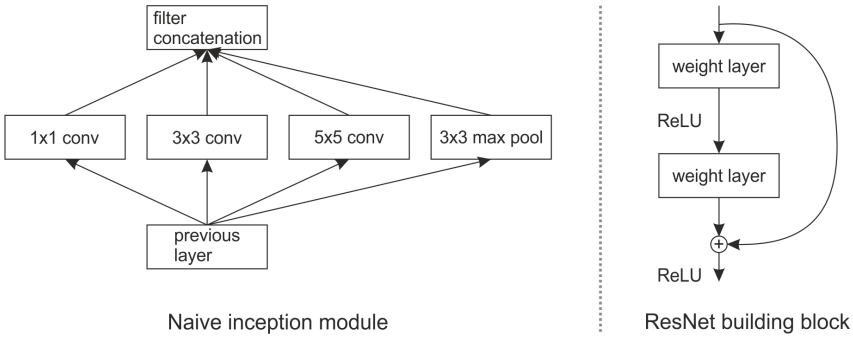


Figure 2.8: Naive inception module (left, redrawn from [SLJ+14]) and simplified ResNet building block (right, redrawn from [HZRS15a])

2.2.2 CNNs for Object Localization

Common convolutional neural networks as described above are designed to classify full images, i.e., to assign an image to the correct class. The next logical step is to localize one object (or any other fixed number of objects) in a higher resolution image. A common and rather simple approach is to extend an existing and pre-trained classification network with an additional regression head forming a two-headed network as shown in Figure 2.9. There are two slightly different ways to run such a network: On the entire image as a whole or multiple times on different subregions of the image (see Overfeat). In both cases, the regression head consists of several fully connected layers being very similar to the classification head. The only major difference is that the output is a 4-tuple of real numbers instead of a class label. The 4-tuple represents a bounding box around the localized object, and different papers tend to use different representations. The most common ones are (x_0, y_0, w, h) with (x_0, y_0) being the coordinates of the top left corner and w and h being the width and the height of the bounding box and (x_0, y_0, x_1, y_1) where the 2-tuple (x_1, y_1) are the coordinates of the bottom right corner.As a consequence of having a 4-tuple of real numbers, the loss function has to change. Popular choices are the Euclidean loss (Equation 2.13) and the Jaccard index, better known as Intersection over Union (IoU, Equation 2.14). Now the regression head can be trained using one of those loss functions (or any other suitable loss function) together with a ground truth bounding box provided in conjunction with the training set. At test time, both heads are attached to the CNN simultaneously.

$$L = \frac{1}{2N}\sum_{i=1}^{N}\|x_i^1 - x_i^2\|_2^2, \qquad [3] \tag{2.13}$$

$$IoU = J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{Area\ of\ Overlap}{Area\ of\ Union}, \qquad [1] \tag{2.14}$$

In [SEZ+13], this technique is modified to increase the localization accuracy by running such a two-headed convolutional neural net in several locations and several scales of the image using sliding windows. This approach is called Overfeat, and it won the ImageNet Large Scale Visual Recognition Competition (ILSVRC [2]) localization challenge in 2013. They apply the trained network in every possible location, hence with a stride of 1, on a higher resolution image. In a standard sliding window approach where every crop is processed separately one after another,

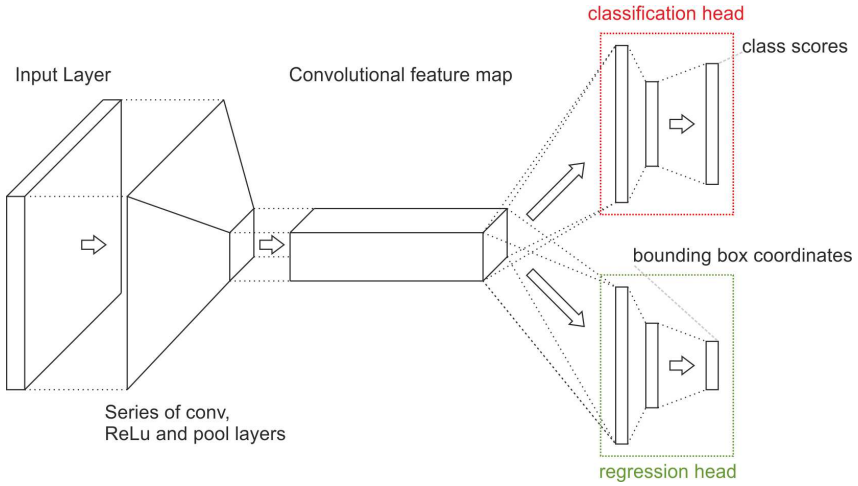


Figure 2.9: CNN with two heads attached to the feature map - one regression head to compute bounding boxed and one classification head.

this would cause a huge computational overhead. To overcome this problem, they took advantage of the structure of convolutional networks as they share computations in overlapping regions by default. This requires a network free of fully-connected layers. Having section 2.1.1.5 in mind, the restriction to use only convolutional layers has no impact on the performance of a network. Figure 2.10 illustrates the benefit of using efficient sliding windows by an example of a neural network with a conv-pool-conv-conv structure trained on inputs of 14x14. For reasons of simplicity, the nonlinear activation layers in between are not shown. In the upper image sequence, the network operates on an input of the same size as the training image resulting in a single output (in the context of classification and localization, this output consist of a pair of a class confidence and a bounding box). The lower sequence shows the same network running on a slightly larger input. It is important to note that each layer is applied successively on the output of the previous layer instead of using the complete stack of layers on every possible location. The red shaded areas in figure Figure 2.10 show the extra computation required for larger scale inputs. In this example, the result is a 2-by-2 map of pairs of outputs containing class confidences and bounding boxes for four different locations. More generally, the size of the output map and the resolution in one direction are defined by equations (2.15) and (2.16).

$$outputMapSize = \frac{InputSize - networkInputSize}{\sum poolSubsampling} \tag{2.15}$$

$$resolution = \sum poolSubsampling \tag{2.16}$$

After running such an efficient sliding window network on multiple scales of the input image, the authors of [SEZ+13] obtained a high number of pairs of outputs. Then they merged all of these outputs to get a single class label together with one bounding box.

2.2.3 CNNs for Object Detection

The simplest CNN based object detector classifies individual crops of the input image taken from every possible position at different scales using sliding windows. In contrast to localization, a

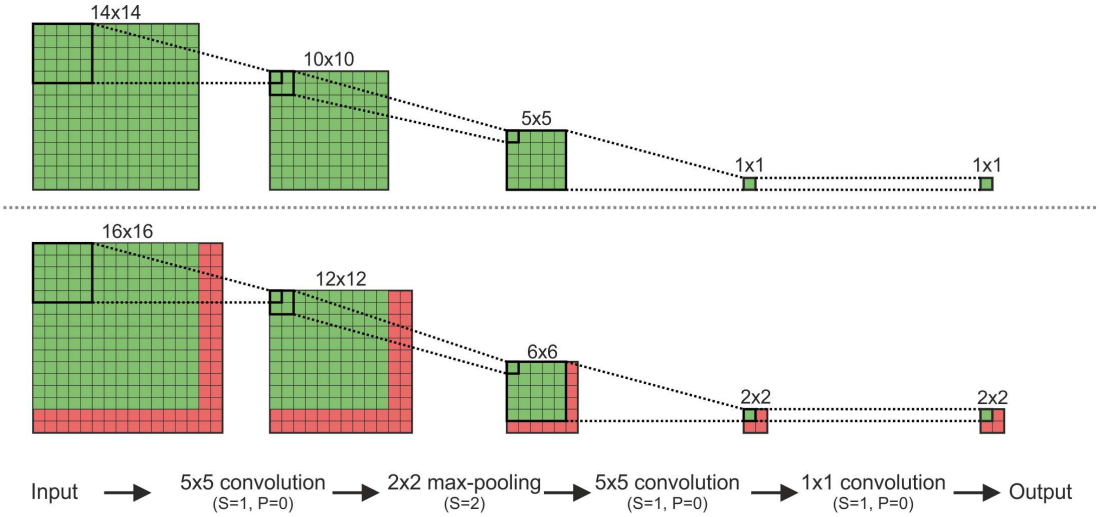


Figure 2.10: Efficient sliding window example with a 4-layer network (5x5conv-2x2maxpool-5x5conv-1x1conv). The network is trained on a 14x14 input image (upper section). At test time, the network is applied on a larger image taking advantage of shared computations (lower section). Only the red shaded areas require additional computations. The output is a 2x2 map of class scores.

regression head is omitted. Knowing the position and the extent of the current crop, the bounding box is simply the outline of this crop. Of course, it is possible to combine an overlapping set of bounding boxes to create a more accurate one. However, this approach requires a very fast classifier as there are many windows to process. Using the efficient sliding window technique explained above reduces the computational costs by a large factor.

2.2.3.1 R-CNN

In [GDDM14], Girshick et al. presented a new method for object detection called R-CNN by combining traditional CNNs with region proposals. Applying a neural net on promising areas only allows to either use a larger network to get a higher accuracy or to boost the performance while maintaining the accuracy. In their work, Girshick et al. use selective search to compute about 2,000 region proposals per image. Next to the selective search algorithm, there are many options how to implement the selection of promising areas, and R-CNN is not limited to a specific algorithm. A nice and probably nearly exhausting overview is given in [HBDS16]. After extracting promising areas, the crops are warped to match the input size of the following convolutional neural network to compute feature vectors. The features are then fed into a linear SVM for classification and a bounding box regression module, one of each per class. The left part of Figure 2.11 illustrates this structure. The training pipeline of such a detection network can be split up into four major steps:

1. **Supervised pre-training:** Download of any pre-trained classification network or like AlexNet, ZF net and VGG net. Of course, from-scratch-training is also possible. In [GDDM14], pre-training is done using the ILSVRC2012 classification dataset (1000 classes).

2. **Domain-specific fine-tuning:** Replacement of the last fully-connected layer (1000 class classifier) with a smaller, domain-specific classification layer. In case of the Pascal dataset,

the new fully-connected layer has 21 categories (20 classes from the dataset plus one class for the background). Then, the network is trained on the warped proposal windows.

3. **Feature extraction:** Removal of all fully-connected layers of the fine-tuned network and computation of the feature vectors by forwarding region proposals through the remaining network. Saving all features can require a lot of disk space.

4. **Detector training:** Training of a linear SVM together with a linear bounding box regression model using the extracted features.

Despite the high test-time accuracy, R-CNNs have three major drawbacks. First, the training pipeline is long and complex, and because detector training is independent of CNN training, there is no backpropagation. Hence the CNN features are not updated during detector training. Second, the algorithm is slow at training time and needs much memory to save the features. According to [15], it takes about 200GB to store the features for the Pascal dataset. Third, R-CNNs are rather slow at test time (see Section 2.4 for some numbers). To overcome some of the problems, Girshick released a follow-up paper presenting an improved architecture called Fast R-CNN [Gir15]. Figure 2.11 shows the Fast R-CNN architecture on the right side. While the accuracy is only slightly higher compared to R-CNN, this version achieves a higher frame rate due to swapping the convolutional network with the region proposals. It also consumes less memory at training time because of the joint training of the whole system. Applying the convolutional network on the entire image enables to sharing computation across the image (efficient sliding window). The ROIs selected from the feature map are warped to a fixed size to use them as input for the fully-connected layer with a special version of maximum pooling, called RoI pooling. See [Gir15] for more details on this pooling method.

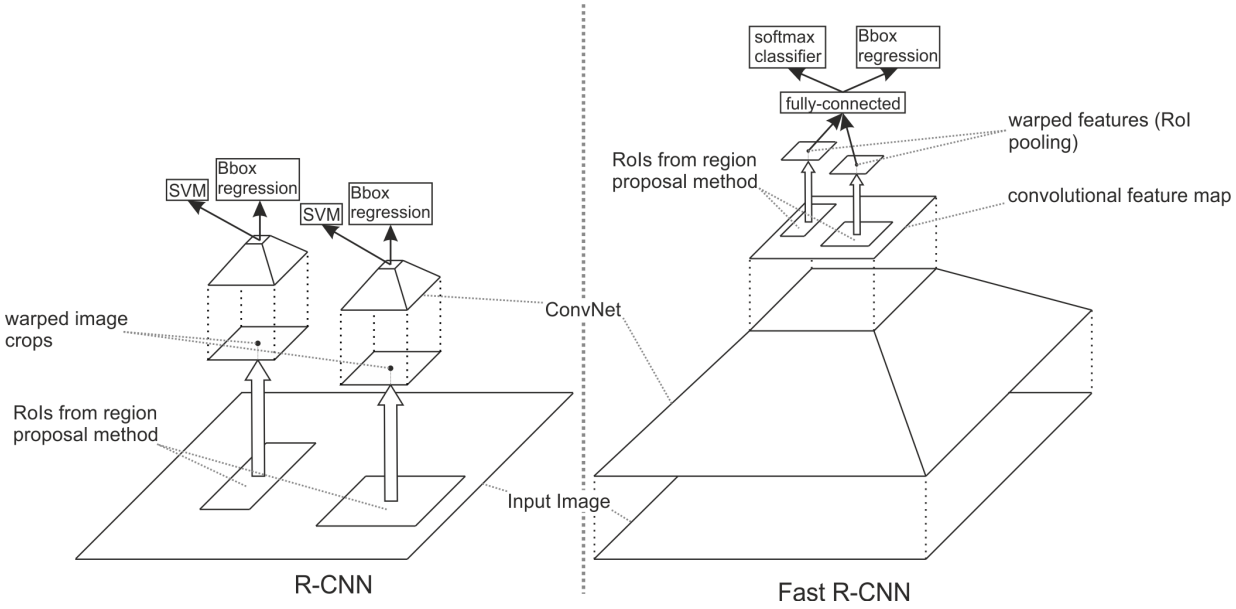


Figure 2.11: Structure of the R-CNN detection method (left) and of Fast R-CNN (right).

While R-CNN spends the majority of the computation time on the CNN (47s compared to 2s for region proposals), Fast R-CNN suffers from the selective search algorithm to create the region proposals (0.32s CNN and still 2s for region proposals. Another follow-up work introducing Faster R-CNN resolves this issue [RHGS15]. The architecture is still the same as Fast R-CNN,

but instead of the selective search algorithm, they use a small convolutional network trained to generate region proposals. This network is called Region Proposal Network (RPN). Table 2.10 shows the improvement in performance achieved with this approach.

2.2.3.2 YOLO

YOLO is an acronym for "You Only Look Once" which is a CNN based object detection system focusing on speed as presented in [RDGF15]. Compared to other, classifier based detection frameworks, YOLO runs on the whole high-resolution image at training time as well as at test time. This decreases the running time and enables the network to be aware of contextual background information. As a downside, the authors say that the accuracy is behind other state-of-the-art methods having localization errors as their main error source. In contrast to [SEZ+13] and other localization networks, YOLO relies on a single-headed network with a 5-tuple output (x,y,w,h,confidence).

Redmon et al. also published a paper with two new version called YOLOv2 and YOLO9000 [RF16]. YOLOv2 is an improved version of YOLO being faster and more accurate at the same time. To achieve this, they added some state of the art techniques like batch normalization and anchor boxes together with novel ideas. YOLO9000 is a joint model for detection and classification. It is trained on a combined dataset of ImageNet (classification) and COCO (detection) and evaluated on the ImageNet detection dataset. ImageNet contains 1000 different classes and shares only 44 with COCO meaning that the trained network has only seen classification data (and no detection data) for most of the test images. However, YOLO9000 can achieve a decent accuracy while still running in real time. Section 2.4 lists some numbers for performance and accuracy.

2.2.4 Binary Networks

Especially but not exclusive when targeting hardware (FPGAs) it is necessary to reduce the size of a neural network. Limiting the precision of the used datatypes lowers the required memory and helps to improve the efficiency. With SqeezeNet [IMA+16], the authors present a method called deep compression. Among other improvements, the limit the precision of the datatypes to 6bit while maintaining an AlexNet classification accuracy. There are also a lot of additional papers covering that topic, but such compression methods are outside the scope of this work.

In terms of hardware efficiency, binary networks are more interesting. Binary Neural Networks (BNN) can be divided into two classes, binary weight networks where only the weights are binary and full binary networks with binary weights and binary activations. Two full binary networks are presented in [CB16] and [RORF16]. Both implementations use $\{-1, 1\}$ for binary representation instead of the more hardware friendly set with $\{0, 1\}$. One reason is that using $\{-1, 1\}$ allows to use the same architecture as with standard networks as multiplications equal the XNOR operation. Another advantage is that the outputs of the layers are still approximately zero centered whereas $\{0, 1\}$ leads to all positive activations.

When moving to binary weights, there are different strategies of binarization. In [CB16], the authors compare deterministic (Equation 2.17) and stochastic binarization (Equation 2.18) where x^b is the binarized version of x and $\sigma(x)$ is the hard sigmoid function. They conclude that stochastic

binarization is more appealing, but because it requires the generation of random numbers, deterministic binarization is more efficient and convenient. Full binary networks also require to change the order of the layers as illustrated in [RORF16]. The pooling layer has to me moved straight after the convolutional / fully-connected layers as pooling binary activations would cause a huge loss of information.

$$x^b = Sign(x) = \begin{cases} +1 \text{ if } x \geq 0 \\ -1 \text{ otherwise} \end{cases} \tag{2.17}$$

$$x^b = \begin{cases} +1 \text{ with probability } p = \sigma(x) \\ -1 \text{ with probability } 1 - p \end{cases} \tag{2.18}$$

2.2.5 Hardware-Based Neural Network Architectures

In the last few years, FPGAs have found their way into the (convolutional) neural network research community. Because CNNs benefit a lot form parallel computing, GPUs are currently the number one platform to implement them. However, FPGAs also provide a massive amount of parallel units, but the limited amount of memory is often the biggest bottleneck. Recent high-end devices now feature enough on-chip memory to allow the implementation of high accuracy networks.

A high-performance classification system based on a Xilinx VC709 is presented in [LFJ+16]. The Virtex VC709 is an expensive high-end FPGA platform. For more details on this specific FPGA refer to [25]. They implemented the full AlexNet structure by optimizing each type of layer for parallel and pipelined processing and limiting the precision to 16-bit fixed point datatypes across the whole network. YodaNN [ACRB16] is a hardware-based implementation (ASIC) of a binary weight network with fixed-point activations. To increase the efficiency, they remap the binary representation of the weights from $\{-1, 1\}$ to $\{0, 1\}$ according to Equation 2.19, where w_i^{b*} is the remapped version of the weight w_i^b. Next to other improvements regarding energy efficiency, they used latch-based standard cell memory to store the images during the computations.

$$w_i^{b*} = \begin{cases} 0 \text{ if } w_i^b = -1 \\ 1 \text{ if } w_i^b = 1 \end{cases} \tag{2.19}$$

Another interesting work is the Caffe-to-Zynq project [18] from Xilinx giving an API which provides a direct connection between the Caffe deep learning framework [3] and the Zynq SoC (System on Chip) platform. The API takes the output file of Caffe (containing the network structure and the trained weights) and applies it on a pre-optimized CNN engine running on the programmable logic of the Zynq SoC.

2.3 Traffic Sign Classification and Detection

Traffic signs are a special application case for general purpose CNNs. Some implementations rely on the architectures presented in section 2.2 optimized for traffic signs while other approaches use rather different types of neural networks.

2.3.1 Classification

A traffic sign classification network consisting of multiple individual networks is presented in [CMMS12]. This approach is called Multi-Column Deep Neural Network (MCDNN). An MCDNN is a combination of multiple CNNs where the final prediction is the average of all individual predictions of each CNN. In this work, they used 25 rather small neural networks (five weight layers – three convolutional and two fully-connected) grouped in ensembles of five where each one uses a differently preprocessed version of the input image. During preprocessing, the contrast is normalized to increase the classification rate. They use the following methods to obtain the five variants of the input image:

- **Unchanged image**
- **Image Adjustment:** Increase contrast so that 1% of the pixels are saturated
- **Histogram Equalization:** Transform pixel intensities to have a roughly uniform histogram.
- **Adaptive Histogram Equalization:** Tile the image into 8 non-overlapping regions and transform pixel intensities to have a roughly uniform histogram in each tile.
- **Contrast Normalization:** Edge enhancement using 5x5 difference of Gaussians filters

On the GTSRB the reached an impressive accuracy of 99.46% with this approach. Sermanet and LeCun presented a small sized architecture in [SL11]. They use a convolutional neural network with four weight layers (three convolutional and one fully-connected) with an additional bypass. The features computed in the first convolutional layer are not only fed into the second layer but also forwarded to the classifier as high-resolution features. The basic idea behind this approach is to combine global features (second conv layer) and more detailed local ones (first conv layer). Zang et al. present a traffic sign classification method called *Reformative CNN* using different machine learning strategies (SVM, CNN, AdaBoost) in [ZZZ+16].

2.3.2 Detection

Current state-of-the-art traffic sign detection networks often rely on the different versions of R-CNN. In [CCWY16] the authors focus on detecting Chinese traffic signs building their self-developed dataset. Similarly, [ZYZ+17] presents an R-CNN to detect Chinese traffic signs but they also treat traffic lights as traffic signs. However, the final network based on VGG16 has problems on small signs and produces some overlapping bounding boxes. The authors claim that these issues lead to the rather low accuracy of 34.5 mAP. A different approach using a combination of fixed and trainable filters is presented in [WLL+13]. They also used a method called bootstrap where wrongly classified images in the validation set are moved to the training set. Then they trained the network a second time resulting in an increase of the accuracy by about 1%.

A high-performance FPGA implementation (Xilinx ZC706) for traffic sign detection following a different approach is presented in [SLYO17]. Because current CNNs are too complex for hardware platforms, they use haar-like features similar to the Viola-Jones face detection method [VJ01]. The final detector can run at 126 frames per second on high-definition images (1080p). As a downside, this implementation can only detect a single class. In their work, Shi et al. used

stop signs. Another FPGA based detection algorithm is introduced in [SGH15]. In contrast to the previous method, no kind of training is involved, and the used platform is much cheaper (Xilinx Spartan 6). The detection relies on unique, handcrafted features which are mainly based on colors and color transitions. Although the presented architecture is restricted to speed limit signs, Schwiegelshohn et al. claim that an extension to more classes is no problem. Unfortunately, they give no clues on the accuracy nor the used dataset.

2.4 Comparison of Different Network Architectures

To finish of the chapter on the State of the Art, this section shows a brief comparison of the networks mentioned in Section 2.2 split up into classification networks (Section 2.4.1) and detection networks (Section 2.4.2).

2.4.1 Classification Networks

Table 2.5 and Table 2.7 give an overview of current general purpose classification networks regrading network structure (number of layers, weights) and performance. Table 2.6 and Table 2.8 list some neural network architectures specialized on traffic sign classification again split up in two tables, one for the network structure and one showing different performance metrics.

2.4.1.1 Network Structure

Architecture	Layers	Weights	Source
AlexNet	11 (5 conv, 3 pool, 3 fc)	≈6.2e7	[KSH12]
ZF-Net	11 (5 conv, 3 pool, 3 fc)	≈1e8	[ZF13]
VGG16	21 (13 conv, 5 pool, 3 fc)	≈1.4e8	[SZ14]
GoogLeNet	27 (22 conv, 5 pool, 3 fc)[1]	≈6.8e6	[SLJ+14]
ResNet	154 (151 conv, 2 pool, 1 fc)	≈1.7e6	[HZRS15a]

Table 2.5: Layer-structures of different classification networks

Architecture	Layers	Weights	Source
MCDNN	8 (3 conv, 3 pool, 2 fc)	≈1.5e6	[CMMS12]
CNN + bypass	6 (3 conv, 2 pool, 1 fc)	≈3.7e6	[SL11]

Table 2.6: Layer-structures of different traffic sign classification networks

[1] GoogLeNet is 27 layers deep with a total number of about 100 layers

2.4.1.2 Performance

Architecture	Platform	Top-5 error	Test set	Classes	Source
AlexNet	2x GTX 580	15.3%	ILSVRC 2012	1,000	[KSH12]
ZF-Net	GTX 580	14.8%	ILSVRC 2012	1,000	[ZF13]
VGG16	4x Titan Black	7.2%	ILSVRC 2012	1,000	[SZ14]
GoogLeNet	-	6.7%	ILSVRC 2014	1,000	[SLJ+14]
ResNet	8x GPU	3.57%	ILSVRC 2012	1,000	[HZRS15a]
Inception-v4 + Incept. ResNet v2	Kepler	3.08%	ILSVRC 2012	1,000	[SIV16]

Table 2.7: Classification accuracy and used platforms for different classification networks

Architecture	Platform	Accuracy	Test set	Classes	Source
MCDNN	4x GTX 580	99.46%	GTSRB	43	[CMMS12]
CNN + bypass	-	99.17%	GTSRB	43	[SL11]
Reformative CNN	Intel Core2Duo	98.09%	GTSRB	3[2]	[ZZZ+16]

Table 2.8: Classification accuracy and used platforms for different traffic sign classification networks

2.4.2 Detection Networks

Similar to the section on classification networks above, Table 2.9 gives an overview of the structure of current state-of-the-art object detection networks. Table 2.10 and Table 2.11 show a comparison of the performance of some networks regarding accuracy (mAP) and speed (FPS).

2.4.2.1 Network Structure

Architecture	Layers	Weights	Source
OverFeat	11 (5 conv, 3 pool, 3 fc)	≈1.5e8	[SEZ+13]
YOLO	30 (24 conv, 4 pool, 2 fc)	≈2.7e8	[RDGF15]
YOLOv2	25 (19 conv, 6 pool)	≈2e7	[RF16]
R-CNN (AlexNet)	11 (5 conv, 3 pool, 3 fc)[3]	≈6.2e7	[GDDM14, KSH12]
R-CNN (VGG16)	21 (13 conv, 5 pool, 3 fc)[3]	≈1.4e8	[GDDM14, SZ14]

Table 2.9: Layer-structures of different detection networks

[2] Prohibitory, mandatory and danger signs

2.4.2.2 Performance

Architecture	Platform	mAP	FPS	Test set	Classes	Source
OverFeat	K 20x	24.3	0.5	ILSVRC 2013	200	[SEZ+13]
YOLO	Titan X	63.4	45	Pascal VOC 2007	20	[RDGF15]
YOLO+Fast R-CNN	K 40	75.0	≈3	Pascal VOC 2007	20	[RDGF15]
Fast YOLO	Titan X	52.7	155	Pascal VOC 2007	20	[RDGF15]
R-CNN (Alex-Net)	K 20	58.5	0.056	Pascal VOC 2007	20	[GDDM14]
R-CNN (Alex-Net)	K 20	31.4	0.056	ILSVRC 2013	200	[GDDM14]
R-CNN (VGG16)	K 20	66.0	0.02	Pascal VOC 2007	20	[GDDM14]
Fast R-CNN (VGG16)	K 40	66.9	0.43	Pascal VOC 2007	20	[Gir15]
Faster R-CNN (VGG16)	K 40	69.9	5	Pascal VOC 2007	20	[RHGS15]
Faster R-CNN (VGG16)	K 40	78.8	5	Pascal VOC 2007[4]	20	[RHGS15]
Faster R-CNN (ResNet-101)	8x GPU	76.4	-	Pascal VOC 2007	20	[HZRS15a]
Faster R-CNN (ResNet-101)	8x GPU	85.6	-	Pascal VOC 2007[4]	20	[HZRS15a]
GoogLeNet	-	43.9	-	ILSVRC 2014	200	[SLJ+14]

Table 2.10: Detection accuracy using the mean Average Precision (mAP) metric, speed, and used platforms for different general purpose detection networks

Architecture	Platform	mAP	FPS	Test set	Classes	Source
Faster R-CNN (VGG16)	GTX980Ti	90.9	6.25	self created Chinese	219	[CCWY16]
Faster R-CNN (ZF-Net)	GTX980Ti	90.9	16.67	self created Chinese	219	[CCWY16]
Faster R-CNN (VGG16)	-	31.5	-	-	-	[ZYZ+17]
Cascaded Classifiers	ZC706	99.8[5]	126	German+Belgium	1	[SLYO17]

Table 2.11: Detection accuracy using the mean Average Precision (mAP) metric, speed, and used platforms for different traffic sign detection networks

[3]Network architecture / size only given for underlying network
[4]Trained on a joint dataset of VOC 2007, VOC 2012 and COCO
[5]Training time accuracy

3 Project Description

3.1 Dataset Creation

Having a good dataset is vital to train a neural network successfully. Thus, a suitable dataset for solving a specific problem has to be selected or created with care. For traffic sign classification and detection in the European region, the GTSRB (classification), the GTSDB (detection), and the BelgiumTS datasets are available (see Section 2.1.3.5). However, all of them are not ideal for a combined classification and detection network. This Section will explain the modifications made to the mentioned datasets in order to use them in this project.

3.1.1 Background Information

Both, the GTSRB and the BelgiumTS dataset, are designed for traffic sign classifications containing only images with traffic signs. Training a neural network to also distinguish between traffic signs and background information requires the dataset to imply negative samples as well. The GTSDB contains high-resolution images (1360×800) of different real-world scenes with annotated traffic signs (Figure 3.1). Those images are perfectly suitable to generate tiles with background

Figure 3.1: Example images of the GTSDB dataset. The annotated traffic signs are highlighted in green and purple. The bounding box of the stop sign (purple) has a size of 32×32 pixels. Note: The GTSDB contains only annotations for prohibitory (including speed limits), mandatory and warning signs as well as stop and give way signs. Thus, the blue pedestrian crossing sign in the left image is not annotated.

information. Thus, they are divided into equal sized tiles with a resolution of 32×32. The reasons for choosing this resolution will be explained in the next section. Because 1360 is not a multiple of 32, the size of the images have to be adjusted in width to simplify the process of tiling. Some of the images are also surrounded by a tiny, white border. Although this should not be an issue, all images are cropped in both, width and height, to the closest multiple of 32 before tiling. Tiles with traffic signs (or parts of traffic signs) are removed.

3.1.2 Training-, Validation-, and Test-Set

The dataset used in this work is a combination of the GTSRB and the BelgiumTS dataset to increase the generalization during training by having more variations of each sign in the dataset. As the focus of this work is to detect and classify different classes of traffic signs, not each sign individually. Therefore, a second set of annotations is added to all images in the training, validation, and test set grouping all signs in ten classes. In Table 3.1, the used class IDs are shown together with some examples related to the classes, and Table A.1 (Appendix A) lists all traffics signs present in the dataset including the individual sign IDs and the corresponding class IDs. The training set is the GTSRB with background images and images added from the BelgiumTS training set, while the test set is the GTSRB test set with additional background images.

Class ID	Name	Examples
0	Warning Signs	Uneven Road, Road Works
1	Prohibitory Signs	No Overtaking, No Entry
2	Mandatory Signs	Turn Left, Straight Only
3	Informational Signs	Pedestrian Crossing
4	End-of Signs	End of Overtaking Restricions
5	Speed Limits	Speed Limit (50)
6	Give Way	Give Way
7	Stop	Stop
8	Priority Road	(End of) Priority Road
9	Background	-

Table 3.1: Traffic Sign Classes

Usually, the validation set is subsampled from the training set at random. However, the GTSRB training dataset contains a sequence of 30 images of the same traffic sign recorded when getting closer (Figure 3.2). Similarly, the BelgiumTS consist of multiple three-image-consecutions taken from different camera angles (eight cameras are mounted around the recording van) of each traffic sign. Thus, random subsampling would cause the validation set to be not independent of the training set. To overcome this issue, the validation set is created by hand selecting sequences from the training images.

Figure 3.2 also shows, that the images in the GTSRB dataset are of different size, which is true for the BelgiumTS dataset as well. However, training convolutional neural networks with a dataset of equal sized images is more efficient. Hence, all images in the dataset are rescaled to a resolution of 32×32 pixels. During detection, this resolution defines the smallest size of detectable traffic signs. The purple bounding box in Figure 3.1 (right image) has a spatial extent of 32×32 pixels giving an idea of the maximal detectable distance of a traffic sign.

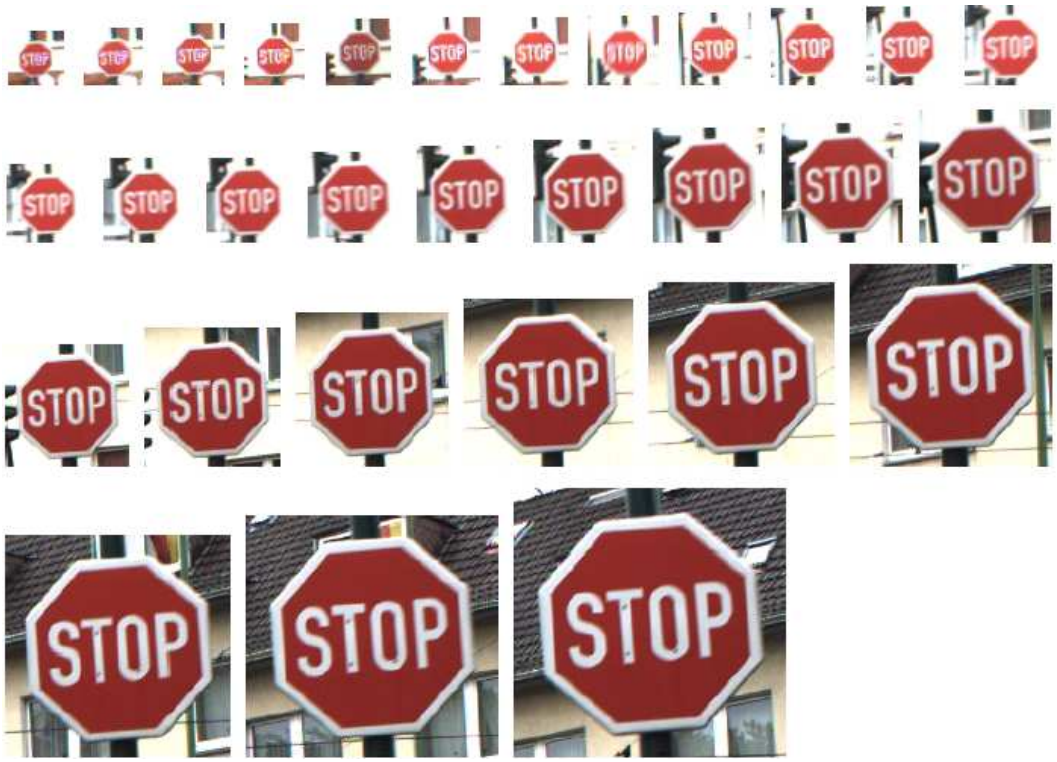


Figure 3.2: Typical image sequence of the GTSRB recored while approaching to a traffic sign

3.1.3 Artificial Distortions

Adding distortions to the training images of a dataset, as described in [SL11], is a common technique to increase the robustness and generalization of a network. Thus, the entire dataset is doubled in size, and the following distortions are added to one half:

- Rotations between $-20°$ and $20°$ (α_r)
- Brightness shifts by adding numbers between -40 and 40 to all three channels equally to take different lighting conditions into account (α_b)
- Color shifts by adding numbers between -30 and 30 to all three channels individually to consider different illumination conditions, i.e. more blueish or reddish light ($\alpha_c^1, \alpha_c^2, \alpha_c^3$)
- Gaussian noise with zero mean and a maximum variance of $1.25e^{-3}$ (α_n)

The integer values for rotation, color shift, and brightness shift, are randomly drawn from a uniform distribution and the Gaussian noise from a normal distribution. To avoid that all random modifications are applied with their maximum values, the strengths are normalized as shown in Equations 3.1 to 3.3. This ensures a constant total distortion.

$$\begin{aligned} \alpha_r &= \gamma_r \cdot X_r && \text{with } X_r \backsim \mathcal{U}\{-20, 20\} \\ \alpha_b &= \gamma_b \cdot X_b && \text{with } X_b \backsim \mathcal{U}\{-40, 40\} \\ \alpha_c^i &= \gamma_c \cdot X_c && \text{with } X_c \backsim \mathcal{U}\{-30, 30\} \text{ for } i \in \{1, 2, 3\} \\ \alpha_n &= \gamma_n \cdot X_n && \text{with } X_n \backsim \mathcal{N}\{0, 1.25 \cdot 10^{-3}\} \end{aligned} \tag{3.1}$$

with

$$\gamma_r = \frac{X_1}{X_{sum}}, \gamma_b = \frac{X_2}{X_{sum}}, \gamma_c = \frac{X_3}{X_{sum}}, \gamma_n = \frac{X_4}{X_{sum}} \tag{3.2}$$

and

$$X_{sum} = \sum_{k=1}^{4} X_k \qquad \text{with } X_k \backsim \mathcal{U}\{1, 1000\} \tag{3.3}$$

so that $\gamma_r + \gamma_b + \gamma_c + \gamma_n = 1$. In the equations above, $\mathcal{U}$ denotes a uniform distribution and $\mathcal{N}$ a normal distribution. Before applying the modifications to the images, α_r, α_b, and α_c^i are rounded to the closest integer. It is also important to make sure that the results are in the range $\{0, 255\}$ to avoid an over- or underflow.

Figure 3.3 shows how the distortions affect the training images with the original images in the first row and the distorted tiles in the second row. The color of the first image is slightly shifted to the blue, the second last image is rotated a bit more reddish, and the last tile is marginally brighter. However, the added noise is not visible as the amount is rather low and the images are tiny.

Figure 3.3: Effect of the artificial distortions on some images of the training set

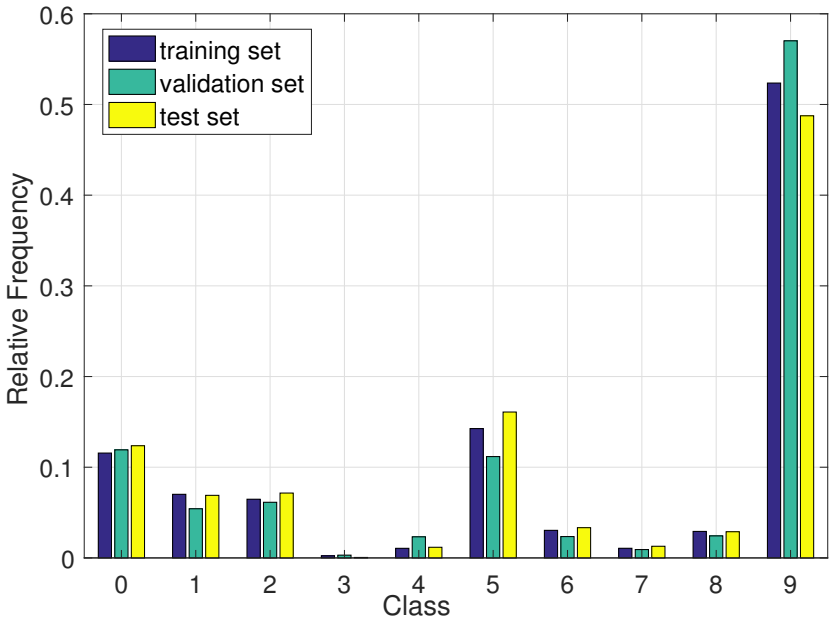


Figure 3.4: Distribution of the traffic sign classes in the used dataset for the training, validation and test set

3.1.4 Final Dataset

With added background information and artificial distortions, the full dataset consists of 140,360 training images of which 73,128 images contain background information, 7,822 validation images (4,460 background images) and 20,631 test images (10,137 background images). Figure 3.4 shows how the classes are distributed over the dataset. Ideally, the distribution should be equal for all three subsets which is approximately fulfilled. To satisfy the traffic sign detection requirement, about the half of each set contains background information. Thus, the traffic sign images should be equally distributed over the remaining 50%. However, the classes 3, 4 and 7 are highly underrepresented while the classes 0 and 5 are overrepresented. This issue can be addressed in future work when revising the dataset.

3.2 Small-Scale Convolutional Neural Networks for Combined Traffic Sign Classification and Detection

The focus of this work is developing a small-scale convolutional neural network for combined traffic sign classification and detection, as outlined in the introduction. In this context, small-scale means not only a low number of weights but also includes a simple straight-forward architecture without computationally expensive functions like the exponential function. This section presents the architecture designed for that purpose and gives insights on the training methods used.

3.2.1 Architecture

In the previous chapter, different architectures for object and traffic sign classification and detection have been presented. The simplest approach for combined classification and detection would be applying any classification network (like AlexNet) at different locations and scales of an image. The primary disadvantage is that this approach requires a lot of computational power. Very similar, but much more efficient is the efficient sliding window technique used in the Overfeat architecture. The only limitation compared to the straightforward implementation is that the classification network has to be built with convolutional layers only. However, this is not really an issue as any fully connected layer can be converted to a convolutional layer. More advanced methods like YOLO and the R-CNN family focus on accurate bounding boxes. Since knowing the exact position of a traffic sign inside an image is nice to have but not necessary for autonomous driving, those architectures do not add useful information compared to simpler methods. For that reason, an architecture based on the efficient sliding window technique similar to the Overfeat network is utilized in this work.

The following sections first describe the developed architecture from a classification viewpoint. Then the same architecture is analyzed considering the traffic sign detection task.

3.2.1.1 Classification Network

As described in Section 3.1.2 above, the images used for training and traffic sign classification are RGB color images with a resolution of 32x32 pixels. Since the dataset is divided into ten classes, the network produces an array with 10 class scores. However, for traffic sign detection,

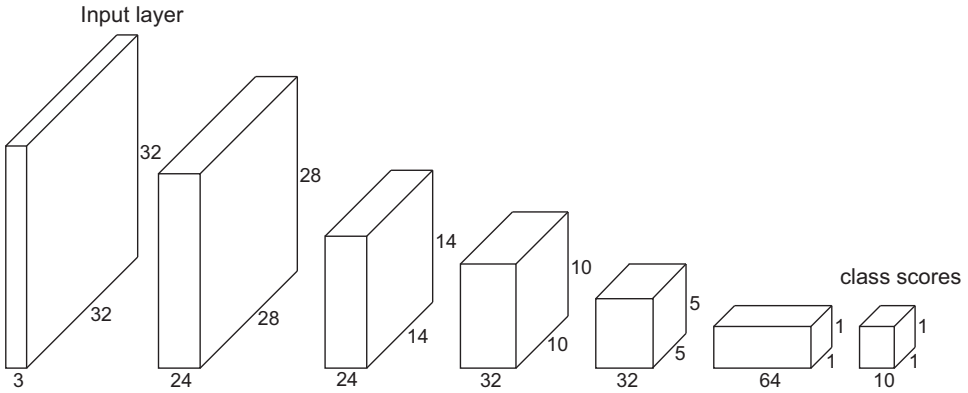
the system should be able to process images of arbitrary size. One way to fulfill this requirement is taking advantage of the efficient sliding window technique (Section 2.1.1.5). As a consequence, the resulting network has to be fully convolutional, i.e., all necessary affine layers have to be converted to convolutional layers (Section 2.2.2).

Under these constraints a simply base convolutional architecture with 5×5 convolutional layers as shown in Figure 3.5a together with the layer details in Figure 3.5b is utilized: $(5 \times 5\ \textit{conv} \Rightarrow \textit{ReLU} \Rightarrow 2 \times 2\ \textit{max-pool}) \cdot 2 \Rightarrow 5 \times 5\ \textit{conv} \Rightarrow \textit{ReLU} \Rightarrow 1 \times 1\ \textit{conv}$. Converting the affine layer to a five-by-five and the output layer to a one-by-one convolutional layer leads to a full-convolutional network. For clarity, those layers will still be referred to as affine and out in this work, although they compute convolutions. As described in Section 2.1.1.5, all filters operate with a stride of one, and no zero padding is applied. A ReLU non-linearity directly follows each weight-layer (except the out layer). Even if Exponential Linear Units (ELUs) and other more recent activation functions can improve the training process, they are contrary to the objective keeping the network less complex. For details on how the different activation functions are defined, please refer to Section 2.1.1.3.

Assuming a stride of one, two stacked 3×3 convolutional layers have the same receptive field as a single 5×5 layer, which can be easily proven using Equation 2.1. To increase the efficiency and decrease the size of the network at the same time, the second convolutional layer (conv2) and the affine layer are replaced by such a block of two consecutive three-by-three layers. Figure 3.5c shows the adapted layer structure. Using multiple smaller layers back-to-back has several advantages. First, they add additional non-linearities which allow the network to learn more complex features. Second, 3x3 layers reduce the number of necessary memory accesses when it comes down to FPGA implementations. A single 3×3 layer also requires 64% fewer computations compared to a 5×5 layer, and in pipelined streaming hardware, the latency is 64% smaller too.

Some recent papers like [IMA+16] and [RDGF15] use 1×1 compression layers in front of the primary convolutional layers to greatly reduce the number of weights. This strategy is also applied in this work in order to address the problem of eventually fitting the traffic sign classification and detection network on low-end FPGAs. Figure 3.5d shows the updated layer structure including four of those compression layers. The squeezing layers add another four non-linearities making the final network ten layers deep. Reducing the depth of the input matrices does not necessarily have a negative impact on a networks accuracy. One way to explain this behavioral is that compression layers combine the incoming features to more complex ones which is similar to the idea of Boosting [24]. To keep the network simple, batch normalization is not used in the final architecture. Though it can enhance the accuracy, this technique has an unfavorable effect on the performance during the forward pass. The reason is that batch norm requires computing the square root of the variance and the expectation of several vectors (Section 2.1.2.4) which is not ideal for running on FPGAs.

Table 3.2 compares the base structure, the modified structure with consecutive three-by-three layers and the final layer structure with 1×1 compression layers regarding the size of the network. For the latter ones, Table 3.2 additionally gives the size of each sublayer too. It is interesting to note that while two stacked three-by-three layers decrease the number of parameters for conv2, they increase the size of the affine layer. This happens because the depth of the input and output volume of the second convolutional layer does not change so that both sublayers act on input volumes with equal depth. However, the increase from 32 to 64 activation maps in the affine layer requires a $3 \times 3 \times 64$ layer with 64 input maps leading to a high number of weights inside. Overall, the 3×3 version saves about 4% weights, but, more important, it prepares the architecture for



(a)

conv1	pool1	conv2	pool2	affine	out
5x5x32	2x2 maxpool	5x5x32	2x2 maxpool	5x5x64	1x1x10

(b)

conv1	pool1	conv2	pool2	affine	out
5x5x32	2x2 maxpool	3x3x32 3x3x32	2x2 maxpool	3x3x64 3x3x64	1x1x10

(c)

conv1	pool1	conv2	pool2	affine	out
5x5x32	2x2 maxpool	1x1x16 3x3x32 1x1x20 3x3x32	2x2 maxpool	1x1x24 3x3x64 1x1x32 3x3x64	1x1x10

(d)

Figure 3.5: Architecture of the used CNN (a) together with different implementations of the layers conv2 and affine: Simple base structure (b), improved version using 3×3 convolutional layers (c), and final structure with additional 1×1squeezing layers (d). All convolutional layers operate at a stride of one without zero-padding and the pooling layers have a stride of two.

employing squeezing layers. Thus, the final structure reduces the total number of weights by 38%.

3.2.1.2 Detection Network

Because the network is fully-convolutional, it can be directly applied on input images with arbitrary size (efficient sliding window). This is illustrated in Figure 3.6. The CNN takes an RGB color image with $W_{in} \times H_{in} \times C_{in}$ and produces a score map of $W_{scores} \times H_{scores} \times C_{scores}$ with $C_{scores} = 10$ for ten classes. The class predictor, shown on top of the score map in Figure 3.6, is responsible for interpreting the class scores to get the final predictions. There are a couple of algorithms that the prediction function can be chosen from. The possibilities include a simple max-function picking the maximum score, thresholding at a minimum confidence level using the softmax function (Equation 2.5), and more advanced methods taking a small neighborhood and multiple scales into account. A short evaluation of some potential choices is given in Section 4.2. In the end, the class predictor produces a classification map of $W_{scores} \times H_{scores} \times 1$ containing the predicted labels. In this case, a number between zero and nine for the ten traffic sign classes.

Layer	weights (base)	weights (3x3)		weights (final)	
		total	sublayers	total	sublayers
conv 1	2,400	2,400	-	2400	-
conv 2	25,600	18,432	9,216 9,216	11,520	512 4,608 640 5,760
affine	51,200	55,296	18,432 36,864	35,072	768 13,824 2048 18,432
out	640	640	-	640	-
total	79,840	76,768 (-3.8%)		49,632 (-37.8%)	

Table 3.2: Comparison of the number of weights for the base-architecture, when using only 3×3 layers (except in conv 1), and for the final, optimized architecture

The width and height of the score map (and the classification map) can be calculated using Equations 3.4, where P is the total sub-sampling caused by pooling, C is the number of convolutional layers, and F_k is the spatial convolution size of filter k with sub-sampling of P_k behind. In practice, the size of the input image is not completely arbitrary, because the size of the score map has to be whole-numbered. Alternatively, whole-numbered map sizes can also be achieved by chopping off the fractional digits and proper sizing of the array size. However, such a strategy requires some structural changes in the neural net framework, and therefore, it is rarely used in practice.

$$W_{scores} = \frac{W_{in}}{P} - \sum_{k=1}^{C} \frac{F_k - 1}{P_k}, \quad H_{scores} = \frac{H_{in}}{P} - \sum_{k=1}^{C} \frac{F_k - 1}{P_k} \tag{3.4}$$

To know which resolutions of images are eligible as input to the neural network is essential for actually using it for traffic sign detection. For convenience, the parameter settings of the 5×5 base structure are used since there is no difference between the various versions regarding width and height of the score map. Thus, the following settings for the parameters of Equation 3.4 apply:

$$\begin{aligned} &P = 4, \quad C = 4 \\ &P_1 = 4, \quad P_2 = 2, \quad P_3 = P_4 = 1 \\ &F_1 = F_2 = F_3 = 5, \quad F_4 = 1 \end{aligned} \tag{3.5}$$

Then, the equation for W_{scores} then turns into

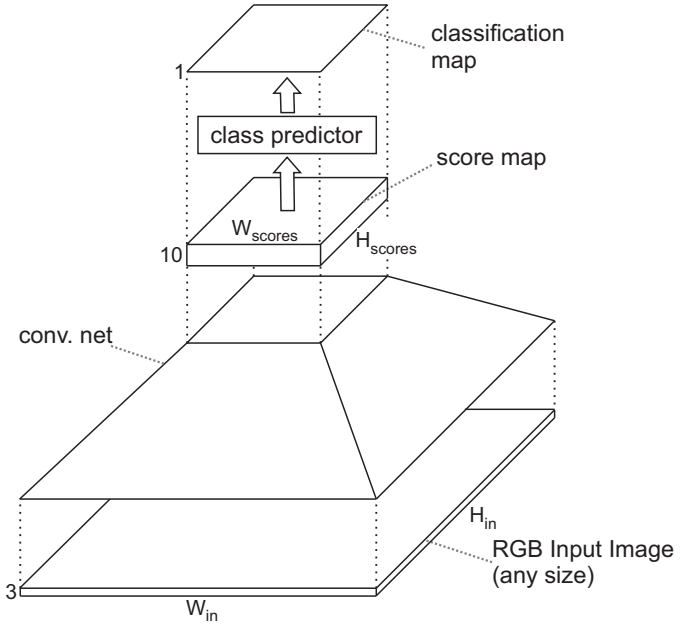


Figure 3.6: Setup for traffic sign detection

$$\begin{aligned} W_{scores} &= \frac{W_{in}}{P} - \sum_{k=1}^{C} \frac{F_k - 1}{P_k} \\ &= \frac{W_{in}}{4} - \left[\frac{5-1}{4} + \frac{5-1}{2} + \frac{5-1}{1} + \frac{1-1}{1} \right] = \frac{W_{in}}{4} - 7 \end{aligned} \tag{3.6}$$

where the sum simplifies to a constant. In order to obtain an integer W_{scores}, the remainder of $W_{in}/4$ has to be zero. This is the first condition for W_{in}. Naturally, the score map has to contain at least one set of class scores. The inequation $W_{in}/4 - 7 \geq 1$ results in the second condition for W_{in}. Repeating this calculation for H_{scores} leads to an elementary set of conditions:

$$\begin{aligned} W_{in} \,\%\, 4 = 0 \quad &\text{and} \quad W_{in} \geq 32 \\ H_{in} \,\%\, 4 = 0 \quad &\text{and} \quad H_{in} \geq 32 \end{aligned} \tag{3.7}$$

with % being the modulo operator. It does not need to be explained further that those conditions are easy to fulfill. Table 3.3 summarizes the sizes of the classification maps for different input images and the total number of class labels. The corresponding score maps contain ten times as much values since the dataset has ten classes. As expected, a 32×32 training image results in a 1-by-1 classification map.

Another point to take care about is the resolution of the scores. Taking a second look at Equation 3.6 and neglecting the subtracted constant reveals that the resolution of is about four pixels, independent of the size of the input. Examining Figure 3.2 and Figure 3.3 indicates that a shift of four pixels should not affect the classification task because it would only change a small part of the background. Even in the worst case, when the traffic sign fills out the 32×32 tile completely, the training network should be robust enough to ignore such a distortion.

Image Type	Resolution	Classification map size	Total labels
Training image	32×32	1×1	1
QVGA	320×240	73×53	3,869
VGA	640×480	153×113	17,289
HD ready (720p)	1280×720	313×173	54,149
GTSDB (cropped)	1344×768	329×185	60,865
GTSDB (original)	1360×800	333×193	64,269
Full HD (1080p)	1920×1080	473×263	124,399

Table 3.3: Resulting size of the classification map and total number of labels for different sizes of input images when using the developed architecture

3.2.2 Training Methods

Having a good dataset and a proper architecture are only two of the three ingredients required for a good performing convolutional neural network. The third factor is training the network. This includes not only the training process itself, but also, among others, weight initialization, hyperparameter tuning, and managing the learning rate. This section covers those topics and gives details on the training methods used.

3.2.2.1 Step Size Control

It is common to adjust the size of an update step during training by modifying the learning rate, e.g., with some decaying function. Popular choices are, among others, the exponential decay, where the learning rate is multiplied with a factor smaller one after each training epoch and a training dependent step decay, where the learning rate is reduced when the training process stops. A simple way to detect a stalling training process is monitoring the loss over time. When the loss reaches a plateau, the learning rate should be decreased. This often results in a global learning rate for all layers. Some methods like [IH00] use individual learning rates for each individual weight. However, such adaptive gradient-based algorithms seem to be fallen a bit out of favor for training neural networks. A possible reason could be that they require much more memory than current methods. For each weight matrix, an equal sized matrix of learning rates is needed and, in case of weight backtracking, an additional matrix holding the previous weight updates.

The learning rate, as well as the strength and the implementation of the decay, are tunable hyperparameters. Setting them correctly is vital for successful training. If the learning rate is too low, the network would barely learn, and a large learning rate would lead to a diverging loss. In this context, it is advantageous to monitor not only the loss but also the size of the update steps. Because the values of the individual weights can span across several orders of magnitude, the absolute step size is not a good metric. The ratio of the updates, the magnitude of the updates relative to the magnitude of the weight values provides more useful information as it eliminates the influence of the actual weight values.

This relative step size can be defined as the quotient of the normed weight matrix and the normed update step matrix, i.e., the output of the update rule. Equation 3.8 shows the Frobenius norm of a multi-dimensional weight matrix W with individual weights w_{ij} and the relative step size r is

defined in Equation 3.9. According to [6], the relative size of an update step should be somewhere around $1e-3$.

$$\|W\|_F = \sqrt{\sum_{w \in W} |w_{ij}|^2} \tag{3.8}$$

$$r = \frac{\|W\|_F}{\|step\|_F} \tag{3.9}$$

While tracking the ratio of the updates gives hints on how to set the learning rate, it still requires a lot of testing and tuning. In the presented work, this idea is taken one step further. Instead of just monitoring the relative step size, it is used to modify the learning rate. In other words, large steps lead to a reduced and tiny steps to an increased learning rate. Therefore, the following rule applies:

$$\eta = \begin{cases} \eta \cdot \eta_-, & \text{if } r > \alpha_u \\ \eta \cdot \eta_+, & \text{if } r < \alpha_l \\ \eta, & \text{otherwise} \end{cases} \tag{3.10}$$

where η_- and η_+ are the factors to increase or decrease the learning rate with $0 < \eta_- < 1 < \eta_+$, and α_u and α_l being the upper and the lower threshold, respectively. At first glance, this idea seems a bit counterproductive. The two hyperparameters *learning rate* and *decay rate* are replaced by four new parameters to tune. Nevertheless, the learning rate and the decay rate are more critical to set. It turns out, that this method is very robust to variations of the parameters as long as the upper threshold is not too big and the thresholds are separated by a factor of at least five. Additionally, both threshold values can be decreased over time similar to the learning rate. Depending on the size of the mini batches during training, the initial setting of the thresholds can vary. In general, bigger mini batches allow a higher learning rate. Thus, the upper threshold can also be higher. For a mini batch size of 50, $\alpha_u = 2.5e{-}3$ to $5e{-}3$ and $\alpha_l = 5e{-}4$ seems to work well and for 100 samples in each batch, $\alpha_u = 1e{-}2$ and $\alpha_l = 5e{-}3$ leads to a good and stable training process. For the decrease and increase factors, $\eta_- = 0.75$ to 0.95 and $\eta_+ = 1.1$ to 1.25 are reasonable default choices.

For training the network architecture described above, the step size control is applied to each layer individually. This results in ten different learning rates maintaining the advantages of having multiple, data dependend learning rates without the downside of requiring more memory during training. Figure 3.7 shows a comparison between standard training and training with activated step size control. In both cases the *rmsprop* update rule with an equal initial learning rate is used. Even though the loss functions converges a bit faster with standard training in the first third, step size control makes the update steps more consistent eventually ending up with a lower loss.

3.2.2.2 Spatial Dropout in Small-Scale Networks

Dropout, as described in Section 2.1.2.2, is a standard technique to avoid overfitting and to increase the accuracy of a neural network. However, overfitting is usually not a big concern

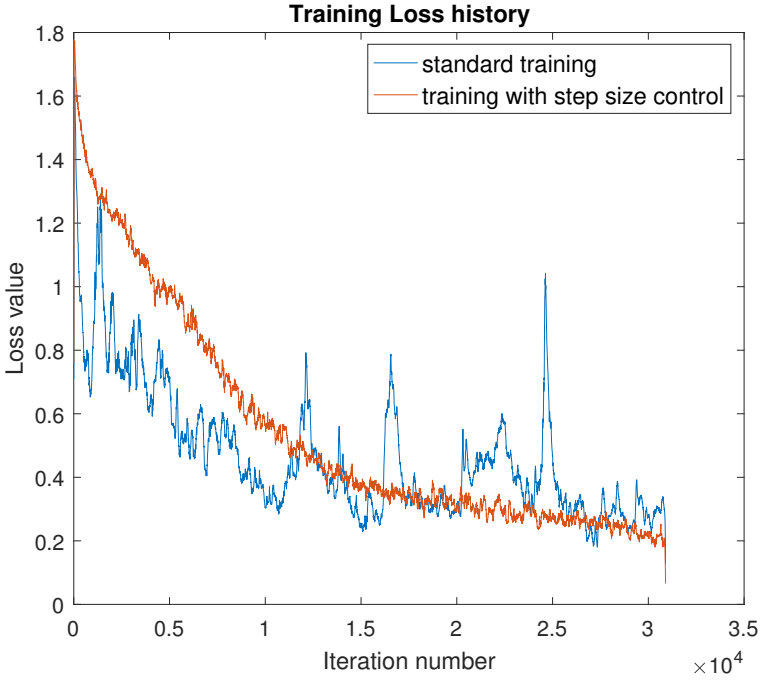


Figure 3.7: Effect of step size control on the loss function

in small-scale neural networks, but the possibility of increasing the accuracy is motivating the following experiments. Since the network is built only using convolutional layers, it is more meaningful to drop complete feature vectors instead of single activations. In fact, this matches the basic idea of dropout as convolutional layers can be interpreted as applying a single neuron at different locations. Thus, the activation map contains multiple outputs of a single neuron. This is the idea behind spatial dropout as presented in [TGJ+14].

Depending on the number of dropped neurons, the size of the active CNN and consequently its capacity gets reduced. That being said, dropout can have a negative impact on a networks accuracy if the remaining network does not have enough capacity to learn a specific dataset. This can especially a problem in networks with small layers. In the following experiments, dropout with varying strength is applied on different layers of the CNN.

Section 2.1.2.2 explains that the weights have to be rescaled at test time in order to have the outputs of the neurons at test time equal to their expected output at training time. However, this is just an approximation which is only valid for wide layers. The error of this approximation drastically increases for small layers. A short example illustrates the problem using inverted dropout. Let y be a vector of activations with five elements sampled from a normal distribution $\mathcal{N}\{10, 1\}$: $y = [10, 9.67, 9.88, 10.03, 9.91]$. The mean of the activations at test time is

$$\bar{y}_{test} = \frac{1}{5}\sum_{i=1}^{5} y_i = 9.898 \tag{3.11}$$

A dropout with $p = 0.5$ could then lead to a binary dropout mask $d_b = [0, 1, 1, 0, 0]$. For convenience, the obligatory correction factor is directly applied in this mask resulting in $d = d_b/p = [0, 2, 2, 0, 0]$. Then, the mean of the activations during training is

$$\bar{y}_{train,1} = \frac{1}{5}(y \cdot d) = 7.82 \tag{3.12}$$

The difference between y_{test} and $y_{train,1}$ is quite big because the real dropout ratio is rather different $p_{real} = 0.4$. This leads to the new dropout mask $d_{real} = d_b/p_{real} = [0, 2.5, 2.5, 0, 0]$ and a much better approximation:

$$\bar{y}_{train,2} = \frac{1}{5}(y \cdot d_{real}) = 9.775 \tag{3.13}$$

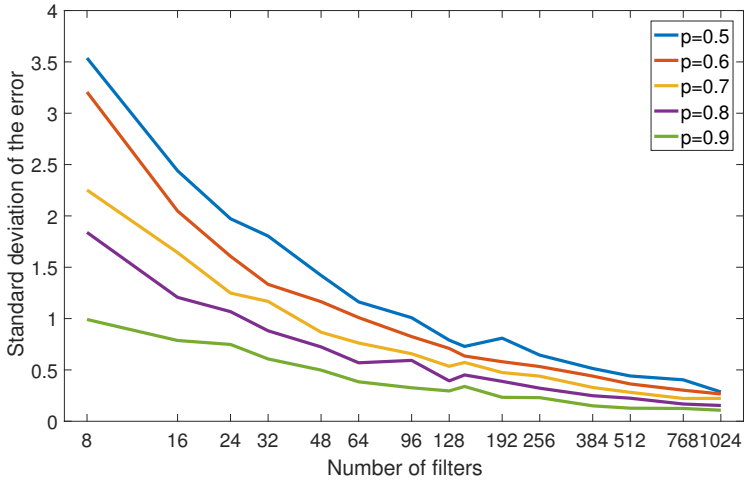
Thus, the real dropout ratio is used in the presented work to scale the activation. This requires additional computations slowing down the training process as the dropout mask has to be analyzed every time. However, small layers do not suffer too much from that penalty and the better approximation is worth the overhead. In Figure 3.8a, the error of the standard approximation method is plotted for different layer sizes and different dropout ratios, and Figure 3.8b shows the result using the real dropout ratio for layers with less than 500 filters. Both figures show the standard deviation of the absolute error between the activations at test and at training time averaged over 100 forward passes, where the dropout mask is re-sampled for each forward pass. Comparing both figures shows, that the error in the range relevant for this work is reduced by a factor of five.

3.2.2.3 Training Details

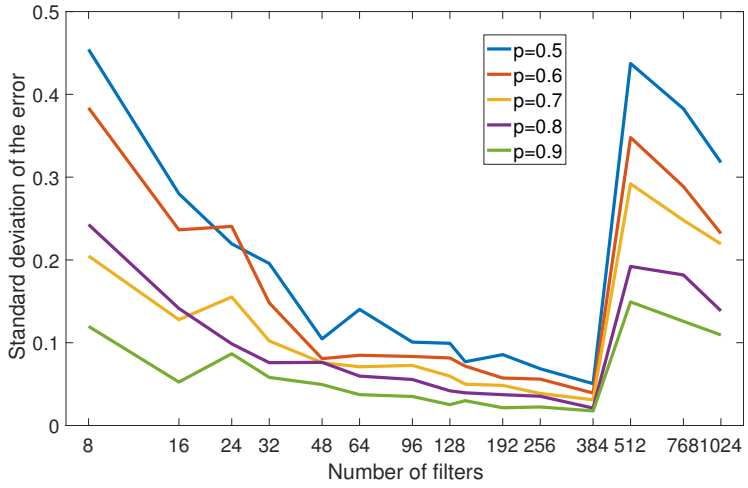
Besides step-size control and dropout, there are also a couple of other things to take care of. Because the network has a total depth of ten layers and no batch normalization is used to control the variance of the layer outputs, a proper weight initialization strategy is key. As described in Section 2.1.2.3, it is recommended to sample the weights from a normal distribution with $\mathcal{N}\{0, 1/\sqrt{n/2}\}$ when using ReLU activations. However, it turned out that using this strategy sometimes leads to a diverging loss. Rescaling all weights with an additional factor of 0.75 resolved this issue. The bias vectors do not require a particular initialization strategy and are therefore set to zero.

As an update rule, Rmsprop is used. The slightly more advanced ADAM method shows no advantage regarding accuracy or convergence rate of the loss. One possible explanation is the fact that controlling the size of an update step can be interpreted as a way of building up a momentum. Small update steps directly lead to an increased learning rate and large steps to a reduced learning rate. Pure Rmsprop has only the capability of reducing the effective learning rate of each weight individually for high gradients due to building up a cache. Step size control adds a second layer of adapting the learning rate on top. The more simple, SGD-based update rules performed a bit worse regarding the convergence rate of the loss. As a consequence, Rmsprop is the best choice to go with for the experiments in the next chapter.

The last thing to mention are the settings of the learning rate and its decay. The initial value of the learning rate is arbitrary in a wide range. Due to controlling the step size, it will be automatically increased or decreased to a proper value. However, a very large setting can lead to a large gradient and together with ReLU non-linearities, this could become a problem. A large gradient can cause the weights to be updated in a way that the ReLUs will never be active again (dead ReLU). An initial setting between $1e{-}4$ and $1e{-}2$ should be safe in most of the



(a) Dropout without calculating the real ratio



(b) Dropout with calculating the real ratio for layers with less than 500 filters

Figure 3.8: Comparison of the common approximation for rescaling the activations (a) with a version with improved accuracy for small layers (b)

cases. Decaying the learning rate directly is not necessary when using *step size control* but it is meaningful to decrease the step size thresholds over time. Although an automatic decay mechanism is easy to implement, the upper and the lower threshold are adjusted manually by inspecting the loss. As soon as the loss stops decreasing, the thresholds are scaled down by a factor of 2 oder 2.5. Starting from $\alpha_u = 1e{-}2$ and $\alpha_l = 1e{-}3$, they eventually ended up with $\alpha_u = 5e{-}4$ and $\alpha_l = 5e{-}5$ in the following experiments.

The last point to note is preprocessing of the data, the inputs. Depending on the type of data, people use different methods like mean subtraction, normalization, and principal component analysis (PCA). When it comes down to images as input data, compression methods like PCA are rarely used. In contrast to that, zero-centering the data by subtracting the mean is very common and essential. A zero-centered input data together with randomly initialized weights leads to an approximately zero-centered output of a convolutional or affine layer. In case of ReLU activations (and other zero-centered activation functions), this helps to improve the training process. For

subtracting the mean, the necessary RGB mean image I_{mean} is calculated over the entire training of the dataset described in Section 3.1 set using

$$I_{mean} = \frac{1}{N_{train}} \sum_{n=1}^{N_{train}} I_n \tag{3.14}$$

The mean image is shown in Figure 3.9. It is extremely important to note that the mean image has to be calculated on the training set only. Then, the mean image has to be saved for subtracting it from all validation and test images. This especially plays a role in a future FPGA implementation as I_{mean} requires some additional memory. To keep preprocessing as simple as possible, no normalization is used because it would need additional memory and also logic to apply the normalization on every incoming image.

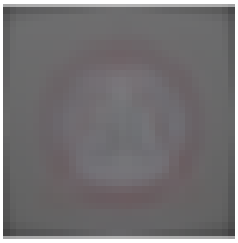


Figure 3.9: Mean image I_{mean} calculated over the entire set of training images

3.3 Binary Network Architectures

Moving from full precision to binary networks can save not only a lot of memory but also simplifies the necessary computations. This section will first explain how to modify the architecture to get a binary weight network. In a second step, the activations will also be binarized leading to a full binary network.

3.3.1 Binary Weight Network

Moving to a binary weight network is a rather straightforward task. When using the binary representation $\{-1, 1\}$, the network structure does not has to be changed. The weights are still stored as full precision variables to allow a gradient-based weight optimization, but for the forward and backward pass, the weights are binarized using Equation 3.15.

$$w^b = Sign(w) = \begin{cases} +1 \text{ if } w \geq 0 \\ -1 \text{ otherwise} \end{cases} \tag{3.15}$$

However, binarized weights introduce one major problem. After subtracting the mean, the input to the first layer can, in theory, contain values between -255 and 255. In practice, values in the range of $\{-127, 127\}$ is much more likely. The weights in full precision networks are smaller than 1 which ensures that the outputs of a layer s scaled down. Figure 3.10 shows the distribution of the weight of the conv1 layer. Since the weights in binary weight networks are equal to 1 (or

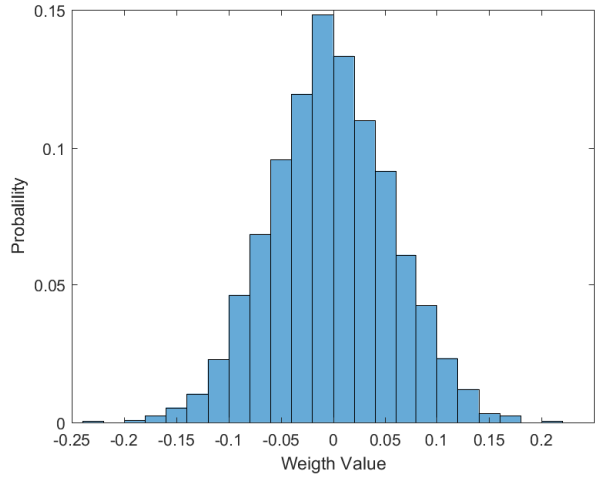


Figure 3.10: Distribution of the weights in the conv1 layer

-1), no rescaling happened. This leads in increasing layer outputs. As a result, training is very inefficient.

The best way to overcome this issue is rescaling the activations after each layer. Among other methods, the half the mean of the squared weights performed best during testing as it is a good measure for the scale of the weights. Alternatively, the mean of the absolute values also works well. The standard mean is not an option as the distribution of the weights is approximately symmetric around zero. In Table 3.4, the mean of the outputs of each of the main layers for one forward pass for the full precision network, the binary weight network, and the binary weight network with rescaling is shown. It is clearly visible, that the activations of the full precision network and the binary weight network with rescaling are similar while the activations of the standard binary weight network are much bigger.

	conv1	**conv2**	**affine**	**out**
full precision	35.93	17.25	19.03	15.67
binary weights	446.78	2.87e+5	7.66e+8	1.75e+9
binary weights (rescaled)	20.7	31.74	4.93	3.67

Table 3.4: Mean value of the activations after each of the main layers

3.3.2 Full Binary Network

In contrast to a binary weight network, a full binary network architecture requires more structural changes. First, the ReLU non-linearity has to be exchanged with a threshold type function. In case of the simulation using the $\{-1, 1\}$ representation, simply thresholding at zero with a sign function should be fine. For a future FPGA implementation with a $\{0, 1\}$ representation, a decision boundary greater zero has to be used instead. A suitable choice would be the half of the number of weights in the previous layer. Another possibility is making the threshold level a trainable parameter so that the network can find the best value during training.

The second change affects the order of the layers. It is common to chain convolutional (and affine) layers with an activation function together and refer to this combination as a single layer. This eases the overall architecture. Also, the sequence of activation function and pooling layer is arbitrary in full precision and binary weight networks for most activation functions as long as they are monotonically increasing. Thus, it makes no difference whether to apply a non-linearity or the max-pool function first. However, this is not true for the sign function. If one would execute the binarization before pooling, the pooling layer would only receive values of -1 or +1. This would require some kind of tie breaker like always taking the first activation or randomly sampling one activation. The result would be a big loss of information in both situations. To overcome this issue, pooling has to take place before applying the threshold-like non-linearity.

Multiplication			**XNOR**		
X_1	X_2	Y	X_1	X_2	Y
-1	-1	1	0	0	1
-1	1	-1	0	1	0
1	-1	-1	1	0	0
1	1	1	1	1	1

Table 3.5: Equality of the multiplication with values in $\{-1, 1\}$ and the XNOR operation with $\{0, 1\}$

Such full binary networks are not only extremely memory efficient but also very simple to compute. Depending on the datatype, binary weights can reduce the required memory by a factor of 64 (when using 64bit doubles for training the network). With a $\{0, 1\}$ representation, the multiplications and additions simplify to XNOR and bit-counting operations. These operations are highly resource-efficient and fast, especially on hardware platforms like FPGAs and ASICs. Besides modifying the decision boundary of the threshold function, the convolutions itself have to be adapted to compute the XNOR operation. In Table 3.5, the equality of the multiplication with a $\{-1, 1\}$ binary representation and the XNOR operation is shown. The remaining network structure can be left unchanged The remaining network structure can be left unchanged and the rescaling the activations like described in the previous section is not necessary because binarizing the activations is already similar to rescaling. Also, dropout is not applied here for two reasons. First, dropout reduces the effective size of the network and a binary network already has a smaller representational power compared to a full precision network. Second, binarizing the weights is also some kind of regularization making the regularization effect of dropout unnecessary.

4 Results

The following sections present the results obtained from different experiments regarding classification and detection accuracy. The experiments cover different sized networks as well as different implementations. This includes comparing the accuracy of the full precision network with an equal sized binary weight network and a full binary network. The last section contains a hardware simulation of a binary convolutional layer using high-level synthesis (HLS) to get an approximation of the size in hardware of the network, i.e., the required resources.

4.1 Classification Results

The first thing to test is how the different networks perform in the classification task. After testing different sized networks trained with the dataset explained in the previous chapter, a slightly modified dataset is used to train another network with the same architecture but a different output layer for binary classification, i.e., for deciding whether a tile contains a traffic sign or not without assigning a traffic sign class (class ID between 0 and 8). Finally, experiments performed on the binary-weight and full binary network show the impact of a limited representational power on the classification accuracy.

4.1.1 Ten-Class Classification

The network presented in Section 3.2.1 is only one example implementation out of a manifold of possibilities using the same architecture as all layers can be changed in width and dropout can be added at various locations. The following sections will present the results obtained with different implementations.

4.1.1.1 Network-Size Experiments

Various sized versions of the classification network are tested to find the best one for fitting the traffic sign dataset. Table 4.1 lists five networks named after the number of weights used, where the 50k network is the one developed in Section 3.2.1. Based on that, the 40k net has smaller and the 60k net wider convolutional layers. Last but not least, the 50k_v2 net features the small convolutional layers of the 40k version together with a fairly wide affine layer. With this setup, it is possible to draw conclusions about which layers impact the classification accuracy most.

The last row in Table 4.1 shows the best accuracy achieved on the validation set. For a meaningful comparison, all networks are trained with identical parameter settings and for an equal number of epochs without dropout. The reason for omitting dropout is that the effect depends on the size of the layers and changing layer sizes could manipulate the results. Additionally, the dropout strength is also a tunable hyperparameter. Testing different sized networks with diverse dropout settings would lead to a huge number of networks which is not manageable within this work. For a discussion on dropout, see Section 4.1.1.2. Comparing the results obtained with the 40k, the 50k and the 60k networks shows that the size of the conv1 and conv2 layers directly influences the accuracy positively. Looking at the 40k and the 50k_v2 network reveals that increasing the size of the affine layer also improves the accuracy of the network. However, the achieved gain is smaller compared to enlarging both convolutional layers. Opposing the 50k to the 50k_v2 network supports this argument as the 50k architecture reaches a higher accuracy while having the same number of weights. Finally, the 100k network performs about the same as the 60k architecture. The tiny difference can be neglected because training one of these networks again can lead to a higher difference than 0.01%. Overall, the 60k network is the best choice out of the architectures presented in Table 4.1. The accuracy is approximately 0.25 percentage points better compared to the 50k network, and with some additional training, it should be possible to break the 99% mark. On the other hand, the 10,000 extra weights can absorb possible penalties from moving to binary weights better.

	40k	**50k**	**50k_v2**	**60k**	**100k**
conv 1	5x5x24	5x5x32	5x5x24	5x5x40	5x5x32
conv 2	1x1x12 3x3x32 1x1x16 3x3x32	1x1x16 3x3x32 1x1x20 3x3x32	1x1x12 3x3x32 1x1x16 3x3x32	1x1x20 3x3x48 1x1x24 3x3x48	1x1x16 3x3x48 1x1x24 3x3x48
affine	1x1x16 3x3x64 1x1x32 3x3x64	1x1x24 3x3x64 1x1x32 3x3x64	1x1x16 3x3x64 1x1x40 3x3x80	1x1x24 3x3x64 1x1x32 3x3x64	1x1x32 3x3x96 1x1x48 3x3x96
out	1x1x10	1x1x10	1x1x10	1x1x10	1x1x10
val. accuracy (%)	96.79	98.73	97.72	98.97	98.98

Table 4.1: Different full-precision network configurations together with the best validation accuracy achieved

4.1.1.2 Dropout Experiments

Table 4.2 shows the effect of dropout on the network architecture described in Section 3.2.1 (50k network). Dropout is used with a probability p of keeping a feature map active in the second (conv2) and the third (affine) layer. For convenience, both 3x3 sublayers inside one layer share the same probability p and dropout is not applied on the 1x1 reducing layers. Because the first layer (conv1) is rather small, no dropout is applied there ($p = 1$). The results in Table 4.2 show, that dropout decreases the gap between the training accuracy and the validation accuracy as expected

but the accuracies itself get reduced as well. The network was trained for an equal number of epochs for all five experiments. In literature, it is often recommended to train networks with applied dropout for more epochs. A frequent suggestion is to increase the number of training epochs approximately by the amount of dropped neurons, e.g., to extend the training time by 30% for a dropout of ($p = 0.7$). However, stretching the training time was omitted in this work due to limited computational resources as this would eventually end up with doubling the total training time.

dropout settings of p			**val. acc. (%)**	**training acc. (%)**	**gap**	
conv1	conv2	affine			val	avg
1	1	1	99.48	98.52	0.95	1.35
1	1	0.85	99.24	98.46	0.78	1.3
1	1	0.7	99.01	98.29	0.72	1.22
1	0.85	0.85	99.08	98.32	0.76	1.19
1	0.85	0.7	98.68	98.03	0.65	1.27

Table 4.2: Effect of different dropout settings on the validation accuracy using the 50k network. In the last two columns, the gap between the best validation accuracy and the corresponding training accuracy and the average gap size are shown.

When looking at the summarized results in Table 4.2, the differences in the accuracies are rather small making a decision based on the facts only difficult. When omitting the last row, the gap between the best and the worst accuracy is only 0.23 percentage points. Based on the tiny accuracy difference between no dropout and dropout with ($p = 0.85$) in the affine layer together with the decreased gap size, the second option is chosen for the final architecture.

4.1.1.3 Detailed Results (60k Network)

Based on the previous experiments, the final classification network uses the 60k structure together with a dropout of $p = 0.85$ for the affine layer. Figure 4.1 shows the loss function, the relative size of the update steps, and the classification accuracy on the training and the validation set. The combination of these three plots helps to analyze the efficiency of the training progress. In the beginning, the update steps are quite large to speed up the convergence of the loss during the first training epochs. Then, the size of the update steps is greatly reduced to fine-tune the weights. The second plot also shows the effect of step size control. Except for the first few epochs, the step size is kept constant for some time and when the training process stops, the step size is reduced. The loss in Figure 4.1 contains a lot of noise which can be explained by using mini-batches of 100 training images for each forward pass. A larger batch size usually decreases this noise but also requires much more memory during training. Another advantage of the mini-batches is that the introduced noise helps training the network as this can be interpreted as some kind of regularization. Figure 4.1 also shows that the accuracy improves very fast in the beginning. After the first epoch, the validation accuracy is already above 90%, and after two epochs, a validation accuracy of slightly above 95% is reached. The gap between the training and the validation accuracy at the best validation accuracy is 0.69 percentage points with an average of 1.01. These values are lower than all numbers in Table 4.2 which is a good indicator that dropout works as expected. Finally, the best validation accuracy of this network is 98.97%.

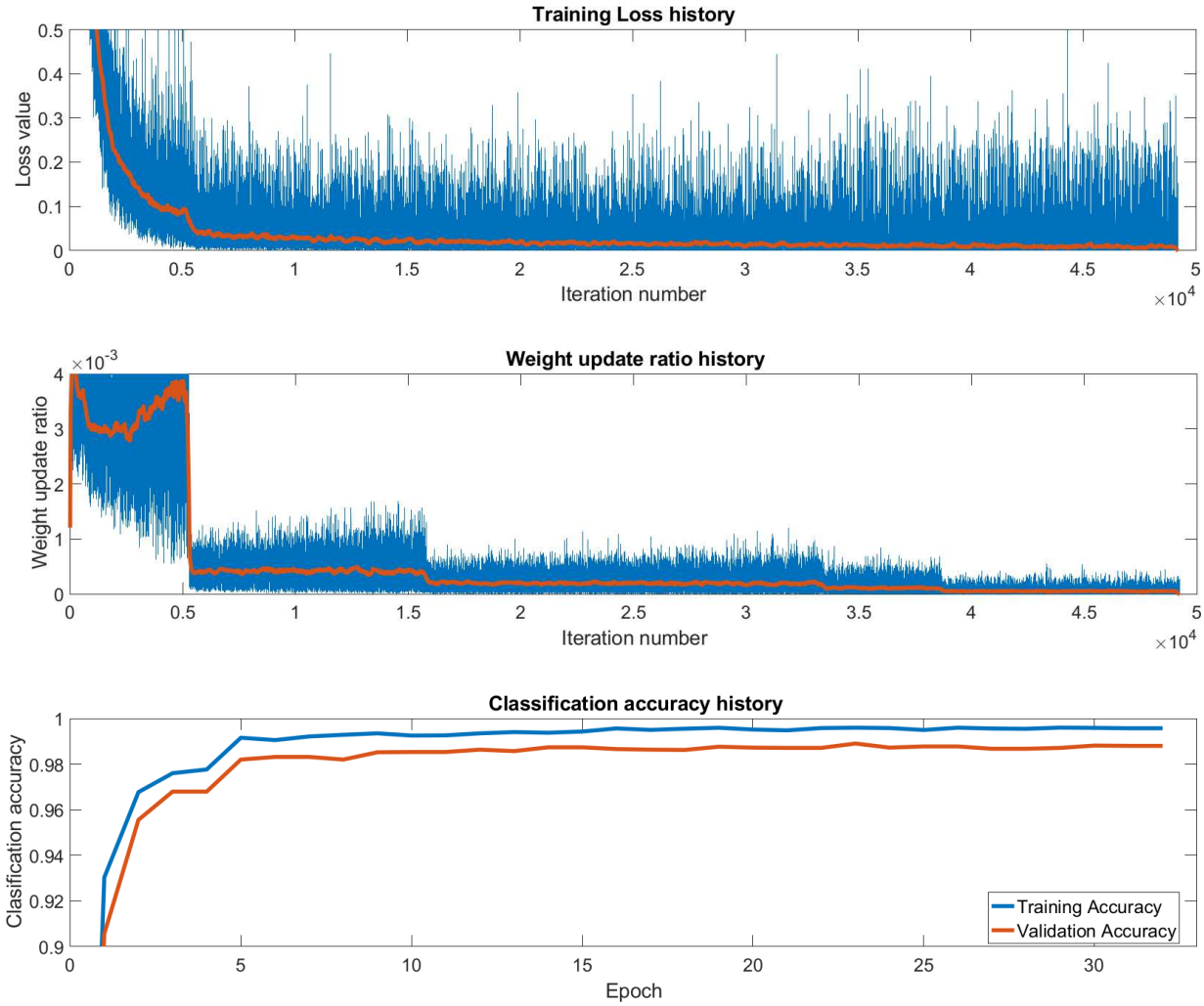
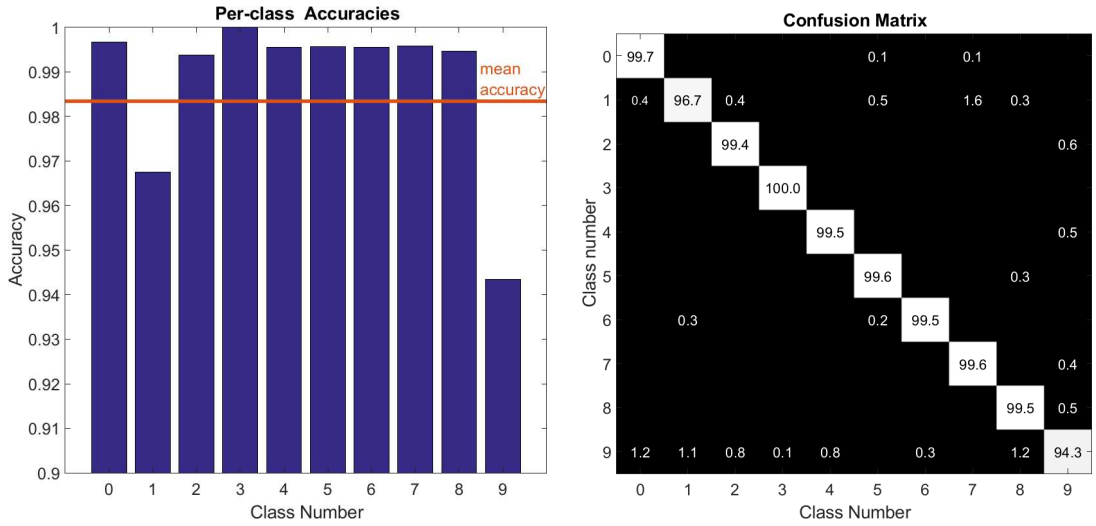


Figure 4.1: Loss, weight update ratio (relative step size), and classification accuracy recored when training the 60k network. The orange line in the first two plots is the sliding mean over 200 samples for an easier interpretation of the trend.

After finding the best network by cross-validating different architectures and dropout settings and selecting the best performing set of weights using the validation set, the final network is ready to be applied to the classification test set. On this data, the 60k network achieves an accuracy of 98.34%. For a more detailed result, Figure 4.2a shows the per-class accuracies together with the mean accuracy. Note that simply averaging those accuracies would lead to a different result because the test set does not contain the same number of images for each class. Two classes, namely class one (prohibitory signs) and class nine (background images) fall behind the other classes. In case of the background images, this drop can be explained by the complexity of this class. The included images show everything that is not a traffic sign, and thus, this class contains a huge variety of objects making a correct classification much more difficult. To analyze the reason for the comparable low accuracy of class one, another visualization giving a more in-depth view is necessary. Figure 4.2b shows the confusion matrix. In its diagonal, the confusion matrix contains the per-class accuracies from Figure 4.2a. The other columns show the percentages of wrong classifications and each row should sum up to 1. For convenience, values smaller than

0.1 are omitted in the plot, and all other values are rounded to a single decimal place. Thus, the condition of summing to 1 may not be fulfilled by all rows. The confusion matrix reveals that 1.6% of the prohibitory signs are wrongly classified as stop signs, and another 0.5% are classified as speed limits. Both cases can be explained by the design of the dataset. Especially the "No Entry" sign looks similar to a stop sign as both are mainly red with a white section in the center. Also, prohibitory signs and speed limits are very similar. Surprisingly, not a single stop sign is misclassified as any other sign. The reasons for this unexpected behavior are hard to identify, but it probably has something to do with how the network has trained. Figure 4.2b also shows how the wrongly classified background images are distributed over the other classes. Those misclassifications are strongly related to the amount of similar looking objects in the full-scale test images. Significantly improving the accuracy for the background class would require a bigger network to cover more fine-grained details.



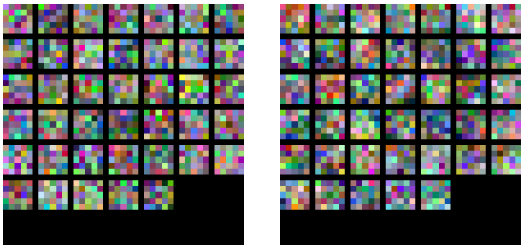
(a) (b)

Figure 4.2: Per-class accuracies together with the mean accuracy (orange line) ((a)) and confusion matrix ((b)) when applying the 60k network on the test set.

Another interesting thing is to look at the weights after training the network. The filters of the first layer are easy to display and to interpret. The weights in such a filter represent how much the filter, and therefore the entire network, likes a specific color or a transition between two colors. Thus, those filters react to the most basic shapes, colors and edges. When working with a dataset containing RGB images, the conv1 filters can be directly visualized as they are just multiple small color images weighting the input channels. Higher level filters with more than three input channels are much harder to visualize and interpret. For example, they could be plotted as grayscale images where each pixel of the image is the average of all input channels to this filter, though such a representation is not meaningful. Instead, there are different methods like presented in [ZF13] available, but visualizing neural networks is outside the scope of this work. Nevertheless, looking at the weight of the first layer can be still useful to analyze the training efficiency.

Figure 4.3a shows the randomly initialized filters of the conv1 layer and Figure 4.3b the trained filters after remapping them to the range 0 to 255. In a well-trained network, the filters should look clean with little to no noise. This is often used as an easy way of checking whether the network trains properly or not. Thus, the trained weights in Figure 4.3b indicate that there is

something wrong with the training process, but the network still performs well as shown above. Several circumstances can explain this behavior, and it is worth discussing them briefly.



(a) **(b)**

Figure 4.3: Randomly initialized weights ((a)) and weights after training the network ((b))

First of all, the gradient could vanquish during backpropagation leading to tiny update steps and eventually prevent learning. This possibility can be eliminated quickly because the filters before and after training look different and, more important, step size control would cause an extremely high learning rate to maintain the selected step size. This is not the case as proven in Table 4.3 where the last learning rates of all layers are listed.

conv1	**conv2**				**affine**				**out**
9.6e-6	2.16e-5	1.11e-5	2.85e-5	1.5e-5	2.84e-5	1.3e-5	2.25e-5	1.09e-5	0.0123

Table 4.3: Learning rates of all individual layers at the end of training.

A second possibility is that subtracting the mean during preprocessing causes the filters to be uninterpretable. Since the input images now contain also negative values, liking a specific color is expressed either by a positive valued weight with a positive input or a negative valued weight with a negative input. Training a network without preprocessed training images leads to similar looking filters allowing the conclusion that subtracting the mean is also not the reason for the random looking filters.

Another way to explain this behavior is that the following squeezing layer combines the conv1 filters with the weights of the squeezing layer to more reasonable looking filters. Combining the conv1 filters with the weights of the first reducing layer leads to the filters shown in Figure 4.4. Again, these filters look quite random and thus, this explanation is not valid as well.

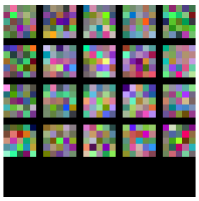


Figure 4.4: Filters after the first squeezing layer

The last possibility is that, due to the squeezing layers, the network is deeper than required for traffic sign classification. Thus, the filters in the conv1 layer could be responsible for some kind of pre-filtering before following layers compute actual, more interpretable features.

4.1.1.4 Binary Weight Network

Similar to the full precision network above, the binary weight network is based on the 60k network, but dropout is omitted to avoid further reducing the effective capacity of the network. The architectural changes are described in Section 3.3.1.

During training, the binary weight network reached a validation accuracy of 96.37% which is only 2.6 percentage points lower than the full precision network with 32bit weights. At test time, the 60k network with binary weights achieves a classification accuracy of 96.53% with the per-class accuracies in Figure 4.5a and the confusion matrix in Figure 4.5b. Note that the y-axis of Figure 4.5a starts at 0.85 compared to 0.9 in Figure 4.2b. The accuracies are lower but very similarly distributed like the ones of the full precision network with significant drops for the classes 1 and 9. Thus, the given discussion is also valid in this case.

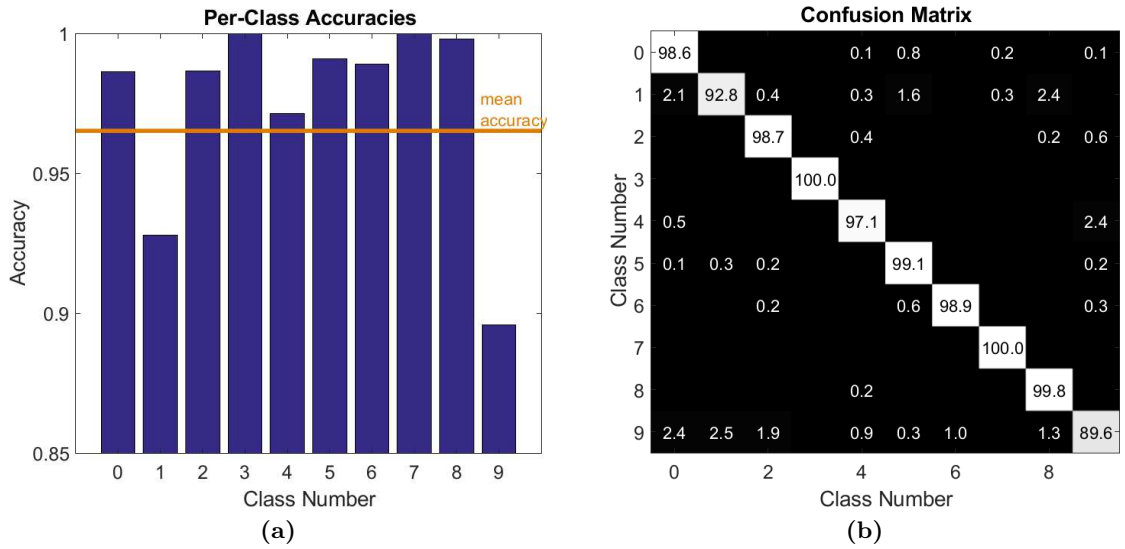


Figure 4.5: Per-class accuracies together with the mean accuracy (orange line) ((a)) and confusion matrix ((b)) when applying the binary weight version of the 60k network on the test set.

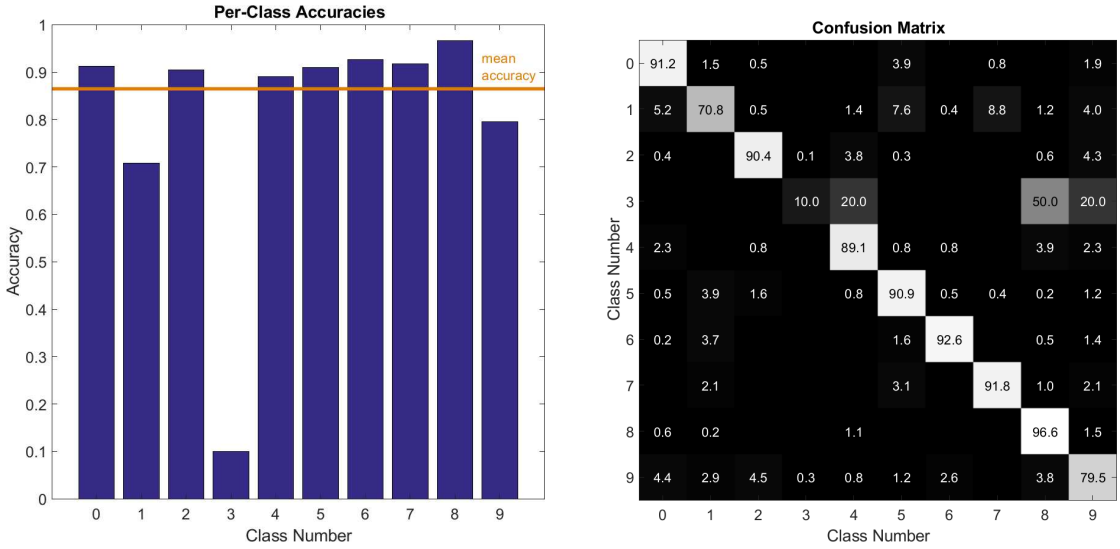
The results using the binary weight network suggest that a full-precision like accuracy can be accomplished with a slightly larger binary weight network. This should not be a problem regarding the required memory as there is plenty of room between the binary weight and the full-precision network.

4.1.1.5 Full Binary Network

Moving to a full binary network with binary weights and activations requires more structural changes as explained in Section 3.3.2. Again, no dropout is used to keep the effective capacity of the network as big as possible.

The full binary network reaches a validation accuracy of 87.63% which is about 9 percentage points lower than the binary weight network. This leads to the assumption that binary activations have a much higher impact on the performance of a network than binary weights. At test time, the accuracy is 86.5%. The corresponding per-class accuracies and the confusion matrix are shown in

Figure 4.6a and Figure 4.6b. Note that the accuracy scale of Figure 4.6a starts at zero this time. Apart from the reduced accuracy values, the per-class accuracies plot with accuracy drops for the classes 1and 9 looks very similar to the plots of the full-precision and the binary weight network. This suggests a proper training process. However, the accuracy for class 3 (informational signs) is very bad. One reason is that this class contains way less images compared to the other classes as shown in Figure 3.4. The test set, for example, includes only 10 images. Thus, misclassifications will not average out. The confusion matrix shows that only one out if the ten images is classified correctly, but five images are assigned to class 8 (Priority Road) for no apparent reason.



(a) (b)

Figure 4.6: Per-class accuracies together with the mean accuracy (orange line) ((a)) and confusion matrix ((b)) when applying the full-binary version of the 60k network on the test set.

The results are a clear indicator that the capacity of the network is too small to fit the traffic sign data with high accuracy. Thus, the number of weights has to be increased.

4.1.2 Two-Class Classification

The 60k network can also be used for binary classification by simply adapting the out layer to two output classes. This experiment uses the pretrained network from Section 4.1.1.3 for faster training with a re-initialized affine layer, i.e., the conv1 and conv2 layers remain unchanged and the affine and out layers are trained from scratch.

The network achieves an accuracy of 99.35% on the validation set with a gap between training and validation accuracy of 0.43 percentage points. On the test set, the mean accuracy is 98.86%. Figure 4.7a shows the per-class accuracies and Figure 4.7b the confusion matrix where class 0 represents traffic signs and class 1 background images. L. Like the ten-class classification, the two-class classification also shows a significant accuracy drop for the background image class. The confusion matrix also suggests the conclusion that in case of the two-class classification almost all traffic signs can be detected (only 0.1% false negatives). The high classification accuracy for class 0 can be explained by shared features across varicose traffic sign classes. For example, most prohibitory signs are very similar to speed limits. However, the number of false positives,

i.e., background images classified as traffic signs, is slightly larger than in the ten-class scenario. There is no obvious reason for this behavior, and thus, it is not possible to give a reasonable explanation.

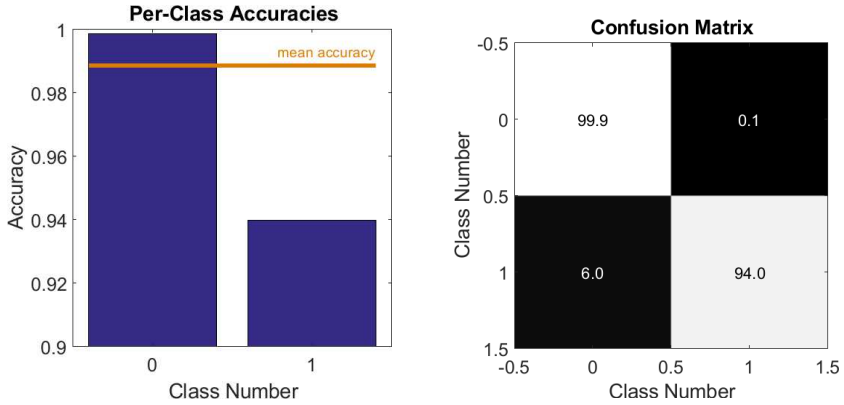


Figure 4.7: Per-class accuracies together with the mean accuracy (orange line) ((a)) and confusion matrix ((b)) when applying the two-class version of the 60k network on the modified two-class test set.

4.2 Detection Results

After evaluating the performance of the classification task, the 60k network is also analyzed regarding its detection accuracy. Therefore, the network is applied to the full-size images, and four downsampled versions taken from the GTSDB dataset. This dataset does not contain annotations for informational signs. Furthermore, the annotations consist of tight bounding boxes together with the corresponding sign ID. Thus, it is hard to compare the results with existing works using the as exactly locating the position of a traffic sign it is not the purpose of this work. The most common performance metric, the mean average precision (mAP) heavily penalties discrepancies from the ground truth. The results presented in this section are based on a simplified version of the GTSDB which only contains the annotated classes. A specific traffic sign is said to be detected when the ground truth labels match the outputs of the network independent of the position of the bounding box. A downside of this method is that false positives can manipulate the results and informational signs are completely skipped. As a consequence, the obtained results are only an estimation. To overcome those limitations, the GTSDB has to be reworked which will be done in future work (see Chapter 5).

Using the efficient sliding window technique requires a very high classification accuracy. As shown in Table 3.3, the network produces an array of 329x185 class labels when applying it to the cropped GTSDB dataset. This equals a total of 60,865 labels for the original sized images only where most of them contain background information. Thus, even an accuracy of 99% for the background class would lead to an average of 609 false positives. The best performing network, the 60k full-precision net, however, has an accuracy of 94.3% for this class resulting in an average of 3,469 false positives. For that reason, the detection performance is only evaluated for the full-precision 60k network and the two-class network.

In Figure 4.8a, one example image of the GTSDB is shown together with the detection results in Figure 4.8b. Because individual colors for all nine traffic sign classes can be hard to distinguish, all detected signs share one color for the bounding box. Note that each of the bounding boxes is simply drawn around the corresponding label and no post-processing is done at this stage. The predicted class is obtained by selecting the class with the highest score, which is very efficient but not ideal. As assumed, there are a lot of false positives, especially in the upper right part of Figure 4.8b. The left "Attention Sign" is not detected because it is slightly too large on this scale of the image. In practice, this problem is solved by applying the network on different scales of the input image, but for convenience, not every scale is shown here.

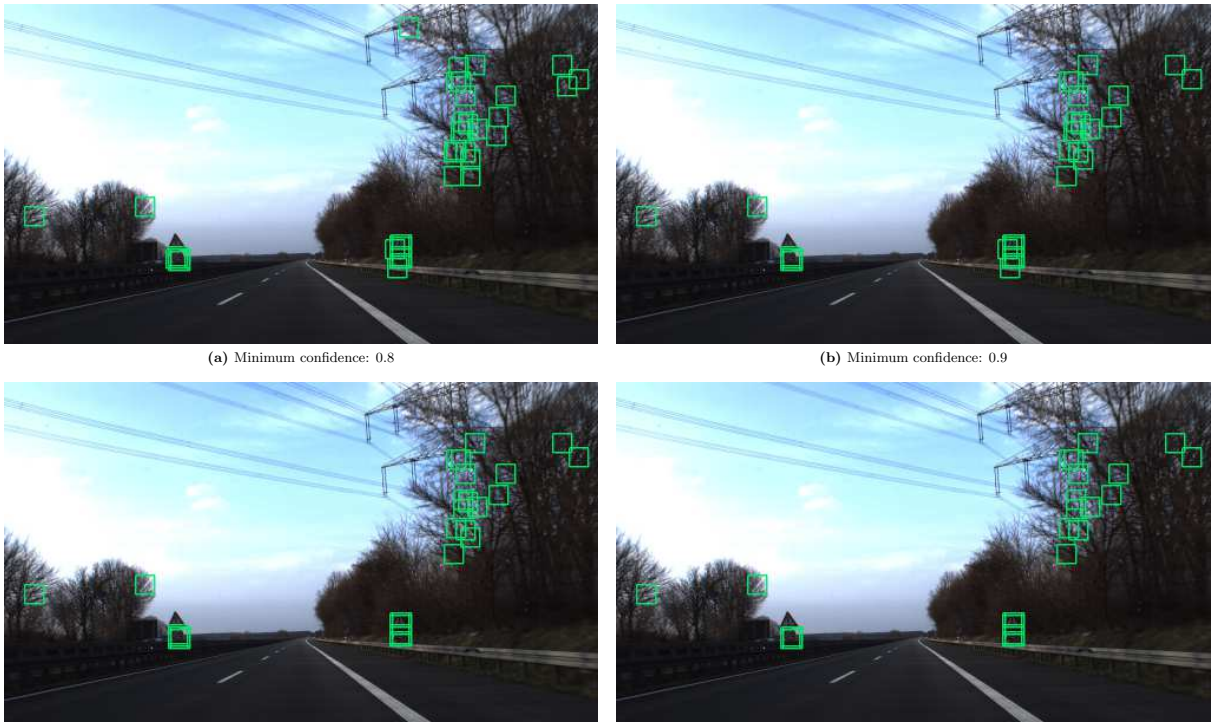
(a) (b)

Figure 4.8: Example image of the GTSDB dataset ((a)) and basic detection results ((b))

A more reasonable way of selecting the predicted class is using the *Softmax* function (Equation 4.1) and thresholding the computed confidence at a fixed level. That means that a detected traffic sign is only valid, i.e., a different label than ten is assigned to a tile, if the confidence is above a minimum value. Otherwise, the tile is set to contain background information. Figure 4.9a shows that when using a threshold of 0.8, the number of false positives is reduced compared to the trivial approach in Figure 4.8b. However, even with a threshold of 0.99 (Figure 4.9d), the improvement is moderate. A downside of the *Softmax* function is that it requires more complex computations (exponential function). The complexity is a problem especially for a future hardware implementation as it requires a significant amount of resources. The exponential functions also introduce another problem. The scores are typically in the range of 1e1 up to 1e3 with some even higher outliers. Exponentiating these values leads to extremely high numbers blowing up almost every modern embedded system. Although there are some normalization tricks to overcome this issue, the additional computations increase the footprint of an FPGA implementation even more. Thus, using the *Softmax* function is not feasible in this case.

$$f_j(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{with} \quad j = 1, ..., K \tag{4.1}$$

A more efficient solution is checking a 3-by-3 neighborhood around each position the score matrix using a sliding window. . Like in the first experiment, the predicted class labels are obtained by selecting the maximum score in the score matrix. The currently examined label is in the center of the neighborhood. Only if the entire window contains this label at least N times, i.e., the remaining eight locations contain the label $N - 1$ times, the detected traffic sign is marked as valid. Figure 4.10 shows the detection results for $N = 2$ and $N = 3$. A higher value of N is more aggressive, and even for $N = 3$, two of the three previously detected signs are not detected



(a) Minimum confidence: 0.8

(b) Minimum confidence: 0.9

(c) Minimum confidence: 0.95

(d) Minimum confidence: 0.99

Figure 4.9: Detection results using the Softmax function and thresholding the confidence at different levels

anymore. Thus, setting $N = 2$ seems a good solution. In both cases, however, one false positive remains. Overall, checking a 3-by-3 neighborhood gives more accurate results than the Softmax function while keeping the necessary computations at a minimum. A larger neighborhood, e.g., 5-by-5, is less meaningful because it does not add further constraints. In contrast, a bigger window increases the chance of finding an additional label making this method less efficient.

(a) Neighborhood checking with $N = 2$

(b) Neighborhood checking with $N = 3$

Figure 4.10: Detection results using 3-by-3 neighborhood-checking with $N = 2$ and $N = 3$

For a comparison, Figure 4.11 shows the results using the two-class classification network. As expected, the number of false positives in Figure 4.11a is higher than in the scenario above because the two-class classification network has a slightly lower accuracy for the background class. On the other hand, having only two classes allows a more aggressive neighborhood checking. Figure

4.11b shows the results with $N = 3$. In this case, all false positives are eliminated. The resulting tiles could now be fed into a second classification network to assign the correct class labels to each tile. These results suggest combining the ten-class classification network with the two-class network in a way similar to Faster R-CNN to reach even better accuracies.

(a) Basic results (taking the maximum score)

(b) Neighborhood checking with $N = 3$

Figure 4.11: Detection results using the 2-class classification network

Overall, the ten-class network was able to detect and classify 98.1% of the traffic signs and the two-class network reached an accuracy of 95.9%. Again, it is important to state that these values could be corrupted by false positives. Repeating the experiments above on 40 different images at various scales reveals that the accuracies have to be reduced by approximately 15% to 20% for the ten-class network and by about 5% for the two-class network. Thus, the final detection accuracy is around 80% for the ten-class network and 90.9% for the two-class network. While the ten-class network has a higher accuracy before applying this correction factor, it turned out that it is also more prone to false positives than the two-class network. This leads to a significantly worse estimated performance when taking the false positives into account. Although a full-precision network produces the results above, a binary weight network should allow similar detection rates when increasing the number of the weights used as its classification accuracy is not far behind the full-precision network. A similar conclusion can be drawn for the full binary network. However, the classification accuracy is a couple of percentage points worse compared to the binary weight network.Thus, the number of weights has to be significantly increased, and the threshold of the sign-activation function may have to be adapted too.

4.3 Hardware Simulation

In this last section of Chapter 4, the approximate size of the 60k network in an FPGA is determined using high-level syntheses (HLS) based simulations.

4.3.1 Simulation Setup

For simulation, the Zynq SoC (xc7z020clg484-1) is used, which is, for example, mounted on the widely used ZedBoard. The obtained results regarding the number of required resources are also valid for other Zynq SoCs and Xilinx FPGAs as long as they share the same technology. Table 4.4 lists the resources available on the mentioned Zynq SoC where BRAM_18K is a dedicated 18kbit

LUT	FF	DSP48E	BRAM_18K
53,200	106,400	220	280

Table 4.4: Available resources on the xc7z020clg484-1

memory block (block RAM), and DSP48E is a special slice for efficient digital signal processing applications. The other resources are flip-flops (FF) and look-up tables (LUT).

During simulation, a clock frequency of 100MHz is used. However, higher frequencies like are also possible to improve the speed of the design when necessary.

4.3.2 Simulation Results (Binary Weight Network)

Implementing the full-precision network on an FPGA is not a good idea because floating-point computations are very inefficient on hardware platforms and the required memory is significantly larger compared to binary weights. However, the binary weight network presented in Section 3.3.1 with the results in Section 4.1.1.4 has one problem. It requires rescaling the weights (division) which results in full-precision weights. For the following simulations, the architecture is modified to overcome this issue. First, the division is replaced by the much more efficient binary shifting operation. Second, the activations are limited to 10-bit signed integer variables. Although both modifications are not tested in the sections above, such a network is more convenient for an FPGA implementation, and the impact on the performance should be low. The results of the full binary network support this assumption because even binary activations do not cause the accuracy to drop below 87%. If necessary, the size of the network can be increased to compensate a possible negative effect.

The following simulations are based on a single, binary 3-by-3 filter applied to an input with one channel in a sliding window fashion. This can be implemented very efficiently on an FPGA in a streaming environment. A one-channel input could, for example, be a grayscale image. This simple filter is then extended to multiple input channels and multiple outputs representing the number of filters applied. To save resources, the weights are stored using the $\{0, 1\}$ binary representation although the network uses -1 and 1. Figure 4.12 shows a simplified version of the original code. The loops iterate over a square filter matrix W and a window A, each of size FILTER_SIZE$\times$FILTER_SIZE. The result of this convolution is summed up in the *temp* variable.

```
for i=0 to FILTER_SIZE
    for j=0 to FILTER_SIZE
        temp = temp + A[i][j] * W[i][j]
```

Figure 4.12: Original code using a $\{-1, 1\}$ binary representation and multiplications

The modified version of this code is shown in Figure 4.13. The "multiply and sum" section is replaced by an *if*-condition. Next to reducing the required memory for the weights, this modification also removes the multiplication leading to a more efficient code. Also, the ReLU functionality is directly included in the output of the convolutional layer. Because each convolutional layer (except the out layer) is followed by a ReLU layer, it is convenient to link both layers together.

Another benefit of this combination is that the output fits in an unsigned integer variable of 8bit instead of a 10bit signed integer.

```
for i=0 to FILTER_SIZE
    for j=0 to FILTER_SIZE
        if W[i][j] = 0 then
            temp = temp + A[i][j]
        else
            temp = temp - A[i][j]
```

Figure 4.13: Modified code for a $\{0, 1\}$ representation. As a consequence, the multiplication is replaced by a more simple *if*-condition.

With the modifications above, the binary weight network is simulated using Vivado HLS. Assuming a hardware-friendly C/C++ code, HLS allows a simple and fast hardware synthesis. In this context, hardware-friendly means, among other constraints, that the loops should not have variable boundaries and that an array should not be accessed more than two times in a single loop iteration (dual-port memory).

Because the design is pipelined and embedded in a streaming environment, it can process one input at each clock cycle. One input can contain multiple variables depending on the number of input channels. This would allow a frame rate of 325fps. However, synthesizing such a design shows, that a single 3-by-3 layer with 32 input channels and 64 filters (which matches the size of the last layer) would require more LUTs than available on the FPGA. The results together with the utilization in are given in Table 4.5. Of course, such an implementation is not feasible in practice, and a frame rate of 325fps is not necessary for traffic sign detection.

LUT	FF	DSP48E	BRAM_18K
74,911 (141%)	67,475 (63%)	0 (0%)	64 (23%)

Table 4.5: Resource usage for a full-parallel design of a 3x3 convolutional layer with 32 input channels and 64 filters

A nice property of HLS is that the performance and footprint of a synthesized design can be easily modified by changing some synthesis parameters (directives). In case of a convolutional layer, it is meaningful to compute each filter element at a time and share the resources between those computations. For a 3-by-3 layer, this causes a drop in the execution time by a factor of 9 leading to a frame rate of 36fps which is still an acceptable value. If necessary, the frame rate can be increased to 72fps when doubling the FPGA clock to 200MHz. Table 4.6 presents the simulation results of different layers of the 60k architecture. Because the design does not require DSP48E slices, these resources are not shown. Compared to the previous results, the necessary LUTs of the layer with 32 channels and 64 filters are reduced by a factor of 8.3 and the flip-flops by 8.5, but the required memory blocks remain unchanged. The reason for this is that buffering two lines of each input channel requires one memory unit per line and channel. Because buffering requires two accesses per memory block in a single clock cycle, these accesses are the limiting factor. Thus, the number of necessary BRAM_18K units is independent of the width and the height of the input feature map as long as one line fits in each memory block. The limit is a

width of 2048 since the BRAM_18K blocks are organized as 2048 slots with 9bit each and the inputs have a size of 8bit.

Channels	Filters	LUT	FF	BRAM_18K
20	48	4,418 (8%)	3,722 (3%)	40 (14%)
24	48	5,223 (10%)	4,558 (4%)	48 (17%)
24	64	6,790 (13%)	5,779 (5%)	48 (17%)
32	64	9,025 (17%)	7,942 (7%)	64 (23%)

Table 4.6: Simulation results for the 3x3 convolutional layers of the 60k architecture

To generalize the results above for an arbitrary 3-by-3 convolutional layer, it can be useful to find a function for the required FFs and LUTs depending on the number of input channels and filters. Therefore, additional simulations are done with the results presented in Table B.1 (appendix). These values are then fed into Matlab's surface fitting tool. Figure 4.14a shows the fitted surface for the LUT resources and Figure 4.14b the corresponding residuals. The residuals show that the maximum fitting error is about 50 which allows a good approximation for the larger layers of Table B.1.

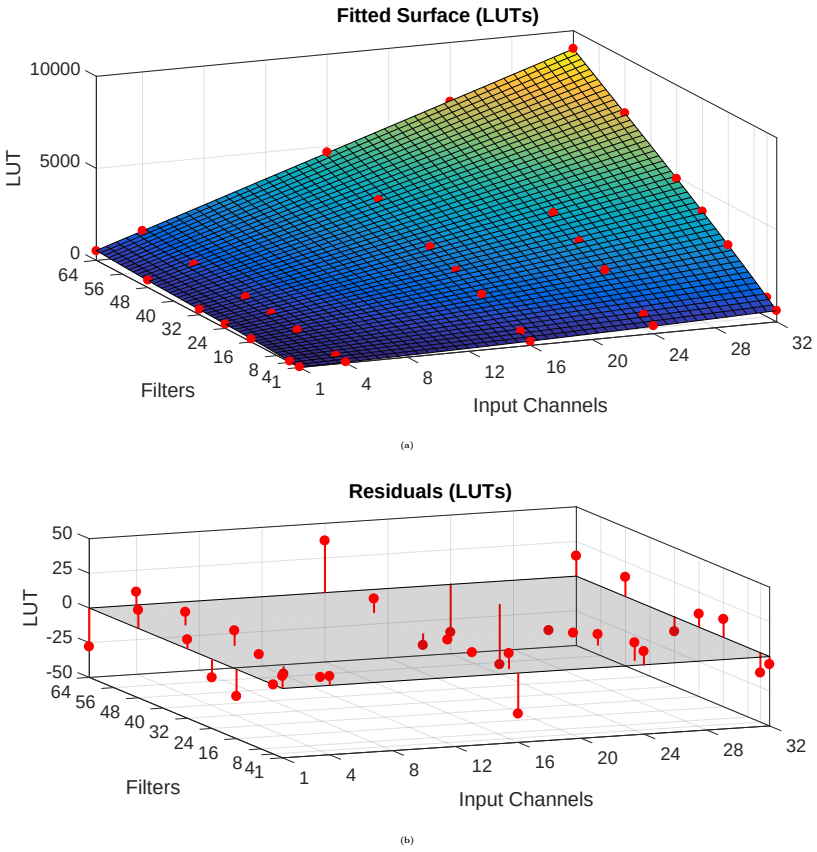


Figure 4.14: Surface fitted to the LUT simulation results ((a)) and corresponding residuals ((b))

Similarly, fitting a surface to the simulated FF resources leads to the surface in Figure 4.15a

together with the residuals in Figure 4.15b. The errors are slightly larger than in the previous case but with a maximum of about 75.

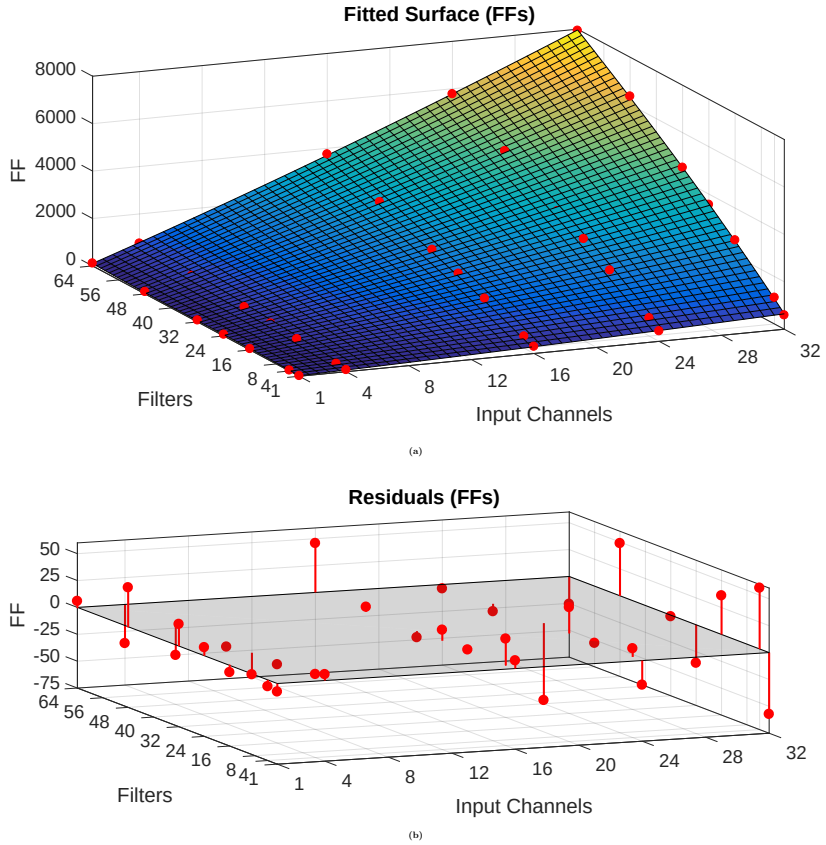


Figure 4.15: Surface fitted to the FF simulation results ((a)) and corresponding residuals ((b))

Both surfaces can be approximated by a polynomial function with an order of two in the first and an order of three in the second dimension. The function is given in Equation 4.2 where d is the number of input channels, and k is the number of filters applied. The values of the nine parameters p_{00} to p_{03} are given in Table 4.7 for calculating the number of LUTs as well for calculating the number of FFs. For the number of required BRAM units, no surface fitting method is necessary. Because there are two line buffers for each input channel, the number of needed memory blocks is simply twice the number of input channels.

$$p(d,k) = p_{00} + p_{10}d + p_{01}k + p_{11}dk + p_{20}d^2 + p_{02}k^2 + p_{21}d^2y + p_{12}dk^2 + p_{03}k^3 \tag{4.2}$$

Table 4.8 compares the synthesis results taken from Table 4.6 with the calculated layer sizes using Equation 4.2. The small differences indicate that Equation 4.2 is a good approximation. Note that the layer in the first row of this table is not used for surface fitting. Validating the results of surface fitting with other layer sizes shows, that the majority of errors is smaller than 1% with some outliers up to 2%. Overall, the polynomial function gives a decent approximation for the size of the hardware implementation without having to perform a time-consuming synthesis.

The 60k architecture also contains a couple of 1×1 squeezing layers and also the *out* layer has this structure. Therefor, it is necessary to simulate those layers too in order to get an idea of

Parameter	LUT	FF
p_{00}	-1.735	11.485
p_{10}	4.695	10.368
p_{01}	9.619	-0.1605
p_{20}	0.3045	0.1657
p_{11}	4.562	3.971
p_{02}	-0.2487	-0.1318
p_{21}	-0.004765	0.01062
p_{12}	-0.005713	-0.009464
p_{03}	0.002561	0.001588

Table 4.7: Parameters for calculating the number of LUTs / FFs of an arbitrary 3×3 convolutional layer using the polynomial function given in Equation 4.2

Channels	Filters	LUT			FF		
		synth	calc	diff	synth	calc	diff
20	48	4,418	4,410	8 (0.18%)	3,722	3,729	7 (0.19%)
24	48	5,223	5,266	43 (0.8%)	4,558	4,564	7 (0.15%)
24	64	6,790	6,824	34 (0.5%)	5,779	5,782	3 (0.052%)
32	64	9,025	9,010	15 (0.17%)	7,942	7,967	27 (0.34%)

Table 4.8: Comparison of the synthesis results and the calculated results using Equation 4.2 regarding the number of LUTs / FFs required for an arbitrary sized 3×3 convolutional layer

the final hardware footprint of the whole system. Table 4.9 contains the HLS synthesis results of the squeezing layers as well as the out layer. Similar to the 3-by-3 layers, the layer does not run at the maximum possible performance. It produces one output every eight clock cycles which greatly decreases the required resources. For the 60k architecture, the squeezing layers require an average of 1.34% of the available LUTs and an average of 0.3% of the available FFs. Thus, it should be enough to approximate the size of a squeezing layer with 1.5% of the look-up tables and 0.5% of the flip-flops instead of using a more complicated polynomial. Table 4.9 also reveals that the 1-by-1 layers do not require any BRAM units. The reason is that a 1×1 convolution does not require buffering complete lines as the mask has a size of one in each dimension. This is also the main reason for having a rather small footprint in hardware.

Channels	Filters	LUT	FF	BRAM_18K
40	20	504 (0.9%)	234 (0.2%)	0 (0%)
48	24	690 (1.3%)	324 (0.3%)	0 (0%)
64	32	1,207 (2.3%)	537 (0.5%)	0 (0%)
64	10	465 (0.9%)	226 (0.2%)	0 (0%)

Table 4.9: Simulation results for the 1x1 convolutional layers of the 60k architecture. Rows one to three are squeezing layers and row four is the *out* layer

For an approximation of the hardware size of the whole 60k architecture, two additional layers have to be analyzed. These are the 5-by-5 convolutional input layer and the 2-by-2 max-pooling layers. Synthesizing the convolutional layer results in an usage of 1,422 (3%) LUTs, 1,040 (1%) FFs, and 12 (4%) BRAM_18Ks under the same constraints as used for the 3×3 layers. The pooling layers are simple to analyze. The hardware required for finding the maximum out of four values should be minimal, but 2-by-2 pooling requires buffering one line of each input channel. Because splitting up the maximum operation, i.e., taking the maximum of two values followed by taking the maximum of the resulting value and two additional values (see Equation 4.3), does not change the result, half of the necessary BRAM units can be saved. In more detail, taking the maximum of the two values to be buffered reduces the required memory as well as the memory accesses by the factor of two since a new value only has to be stored every second clock cycle. As a result, each 2-by-2 max-pooling layer needs a number of BRAM units equal to the half of the input channels of this layer. That is 20 for the first and 24 for the second pooling layer.

$$\max(a_0, a_1, a_2, a_3) = \max(\max(a_0, a_1), \max(a_2, a_3)) = \max(\max(a_0, a_1), a_2, a_3) \tag{4.3}$$

Combining all simulation results allows giving an approximation for the whole 60k architecture. Summing up all resources from the different layers as described above leads to the hardware footprint given in Table 4.10. Based on previous experiences with HLS, the final hardware blocks (IP cores produced by the Vivado HLS tool) require about 5% fewer resources (LUTS and FFs) due to further optimizations. Table 4.10 indicates that the BRAM_18k blocks are the main limitation of the Zynq SoC mounted on the Zed Board. It is also important to note that the given approximation does not include resources required for controlling the individual blocks, controlling a suitable camera and filtering the detection results. However, there are plenty LUTs and FFs available to implement those functionalities, but it is important to pay attention to the BRAM blocks.

LUT	FF	DSP48E	BRAM_18K
30,434 (57%)	24,686 (23%)	0 (0%)	256 (91%)

Table 4.10: Estimation of the required resources for the 60k network on a ZedBoard

5 Conclusion and future work

5.1 Conclusion

The objective of this work was to develop a resource- and performance-efficient convolutional neural network for combined traffic sign detection and classification. The final architecture has only 60,000 weights without using computational expensive functions and reaches a classification accuracy of 98.34% at test time. This result is very close to other state-of-the-art implementations (see Section 2.4.1), but with only 3% to 8% of the weights. However, the accuracies are hard to compare as different works use different datasets with varying numbers of classes. Moving from full-precision to binary weights can drastically improve the resource efficiency of an FPGA implementation at the cost of only 1.81 percentage points of classification accuracy. Compared to that, a full-binary network of equal size reaches only a test-time accuracy of 86.5% which indicates that such configurations require significantly more weights or that a better training method is needed.

On the other hand, reliable traffic sign detection is a more demanding task. As a consequence, applying the network to a full-scale image in a sliding window fashion results in an accuracy of approximately 80%. Again, it is not easy to draw a comparison to existing solutions (Section 2.4.2) since this work does not focus on outputting precise bounding-boxes. Thus, it is not possible to use the mean Average Precision (mAP) as accuracy metric. The detection results also show that a classification based detection requires a very high classification accuracy because running the network on a 640×480 image produces 60,865 predictions and an overall accuracy of 99% still causes 608 misclassifications. To reduce the number of misclassifications, this work compares some possible post-processing methods of which neighborhood checking works best.

Finally, this work shows that it is possible to implement a high-accuracy neural network on a low-cost FPGA platform. Generalizing the synthesis outputs helps to get an approximation of the hardware size of arbitrary layers. The results also show that the required memory due to buffering is the major limitation while storing binary weights does not require block memory resources.

5.2 Future work

This work has shown that a small-scale convolutional neural network can achieve a state-of-the-art classification accuracy but lacks at detecting traffic signs when applied to an image using

the detection by classification approach. One alternative solution could be a modified Faster R-CNN approach based on such a binary weight network. Because it is not necessary to locate the traffic signs precisely in the context of autonomous driving, the regression head predicting four bounding box coordinates can be removed to save resources. Another thing to optimize is the region pooling together with the feature warping which takes place after the convolutional network. There are two possibilities to do this. The first and probably simpler one is finding a resource and performance efficient way of transforming and rescaling the features to equally sized feature maps so that they can easily be fed into the fully connected network. The second solution is modifying the region pooling algorithm in a way that it directly outputs suitable features.

Another thing to work on is improving the accuracy of full binary networks. Such networks allow a very efficient implementation on an FPGA platform, but they are still not mature enough to be widely used in practice. The main advantages of full binary networks are the convolution operation implemented using XNOR and bit-counting functions and the reduced memory requirements due to buffering. Because the activations are binary, it is not necessary to buffer lines of 8bit values. One of the biggest problems is training full binary networks. However, improving the training process, also for full precision networks, is a very active area of research with new papers appearing at regular intervals.

Last but not least, the dataset is another topic to be worked on. Heaving a properly designed training and validation set is crucial for reaching a high accuracy. One problem discovered in this work is that most datasets are designed for traffic sign detection with accurate bounding boxes. However, this is not required in this context as outlined in Section 1.2. Thus, it is necessary to rework the dataset to obtain more valuable results at test time.

A Dataset Sign IDs

Name	Sign ID	Class ID	Name	Sign ID	Class ID
Speed Limit (20)	0	5	Traffic Signals	26	0
Speed Limit (30)	1	5	Pedestrian Crossing	27	0
Speed Limit (50)	2	5	Children	28	0
Speed Limit (60)	3	5	Cyclist Crossing	29	0
Speed Limit (70)	4	5	Slippery Road	30	0
Speed Limit (80)	5	5	Animals (Deer)	31	0
End of Speed Limit (80)	6	4	End of all Restrictions	32	4
Speed Limit (100)	7	5	Turn Right	33	2
Speed Limit (120)	8	5	Turn Left	34	2
No Overtaking	9	1	Straight only	35	2
No Overtaking (Lorries)	10	1	Turn right or continue straight	36	2
Crossroad w. non-priority road	11	0	Turn left or continue straight	37	2
Priority Road	12	8	Follow lane on the right side	38	2
Give Way	13	6	Follow lane on the left side	39	2
Stop	14	7	Roundabout mandatory	40	2
Closed to all	15	1	End of overtaking Restrictions	41	4
No Lorries	16	1	End of overtaking Restr. (Lorries)	42	4
No Entry	17	1	Dangerous Curves (right first)	43	0
Other Dangers	18	0	Road narrows from both sides	44	0
Dangerous curve (left)	19	0	Road narrows from left	45	0
Dangerous curve (right)	20	0	No Left Turn	46	1
Dangerous curves (left first)	21	0	No Right Turn	47	1
Uneven Road	22	0	Pedestrian Crossing	48	3
Slippery Road	23	0	Cyclist Crossing	49	3
Road narrows from right	24	0	End of Priority Road	50	8
Road Works	25	0	Background	99	9

Table A.1: Traffic Sign IDs

B Complete Simulation Results

Channels	Filters	LUT	FF	BRAM18k	Channels	Filters	LUT	FF	BRAM18k
1	1	26	18	2	24	1	412	434	48
1	4	56	29	2	24	4	759	763	48
1	16	156	47	2	24	16	2,068	1,890	48
1	24	219	62	2	24	24	2,894	2,625	48
1	32	287	72	2	24	32	3,695	3,209	48
1	48	394	92	2	24	48	5,223	4,558	48
1	64	512	110	2	24	64	6,790	5,779	48
4	1	56	77	8	32	1	605	593	64
4	4	132	119	8	32	4	1,042	1,114	64
4	16	402	271	8	32	16	2,785	2,649	64
4	24	569	341	8	32	24	3,874	3,556	64
4	32	728	439	8	32	32	4,915	4,528	64
4	48	1,023	574	8	32	48	6,998	6,350	64
4	64	1,369	758	8	32	64	9,025	7,942	64
16	1	203	293	32					
16	4	483	504	32					
16	16	1,378	1,212	32					
16	24	1,947	1,675	32					
16	32	2,483	2,091	32					
16	48	3,568	2,920	32					
16	64	4676	3,752	32					

Table B.1: Complete overview of the simulation results for 3x3 convolutional layers used for surface fitting

Literature

[ACRB16] Andri, Renzo ; Cavigelli, Lukas ; Rossi, Davide ; Benini, Luca: YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights. In: *CoRR* abs/1606.05487 (2016)

[AKV+15] Angelova, Anelia ; Krizhevsky, Alex ; Vanhoucke, Vincent ; Ogale, Abhijit ; Ferguson, Dave: Real-Time Pedestrian Detection With Deep Network Cascades. In: *Proceedings of BMVC 2015*, 2015

[AMA13] AlAli, M. I. ; Mhaidat, K. M. ; Aljarrah, I. A.: Implementing image processing algorithms in FPGA hardware. In: *2013 IEEE AEECT*, 2013, S. 1–5

[CB16] Courbariaux, Matthieu ; Bengio, Yoshua: BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. In: *CoRR* abs/1602.02830 (2016)

[CCWY16] Changzhen, X. ; Cong, W. ; Weixin, M. ; Yanmei, S.: A traffic sign detection algorithm based on deep convolutional neural network. In: *2016 IEEE International Conference on Signal and Image Processing (ICSIP)*, 2016, S. 676–679

[CMMS12] Cireşan, Dan ; Meier, Ueli ; Masci, Jonathan ; Schmidhuber, Jürgen: Multi-Column Deep Neural Network for Traffic Sign Classification. 32 (2012), 02, S. 333–8

[CUH15] Clevert, Djork-Arné ; Unterthiner, Thomas ; Hochreiter, Sepp: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In: *CoRR* abs/1511.07289 (2015)

[CWHW10] Creusen, I. M. ; Wijnhoven, R. G. J. ; Herbschleb, E. ; With, P. H. N.: Color exploitation in hog-based traffic sign detection. In: *2010 IEEE International Conference on Image Processing*, 2010. – ISSN 1522–4880, S. 2669–2672

[EVGW+10] Everingham, M. ; Van-Gool, L. ; Williams, C. K. I. ; Winn, J. ; Zisserman, A.: The Pascal Visual Object Classes (VOC) Challenge. In: *International Journal of Computer Vision* 88 (2010), Juni, Nr. 2, S. 303–338

[FBCS12] Fowers, Jeremy ; Brown, Greg ; Cooke, Patrick ; Stitt, Greg: A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA : ACM, 2012 (FPGA '12). – ISBN 978–1–4503–1155–7, S. 47–56

[GB10] Glorot, Xavier ; Bengio, Yoshua: Understanding the difficulty of training deep feedforward neural networks. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010

[GDDM14] Girshick, Ross ; Donahue, Jeff ; Darrell, Trevor ; Malik, Jitendra: Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA : IEEE Computer Society, 2014 (CVPR '14). – ISBN 978–1–4799–5118–5, S. 580–587

[Gir15] Girshick, Ross: Fast R-CNN. In: *Proceedings of the IEEE International Conference on Computer Vision* 2015 Inter (2015), S. 1440–1448. – ISBN 9781467383912

[HBDS16] Hosang, Jan ; Benenson, Rodrigo ; Dollar, Piotr ; Schiele, Bernt: What Makes for Effective Detection Proposals? In: *IEEE Trans. Pattern Anal. Mach. Intell.* 38 (2016), April, Nr. 4, S. 814–830. – ISSN 0162–8828

[HSS+13] Houben, Sebastian ; Stallkamp, Johannes ; Salmen, Jan ; Schlipsing, Marc ; Igel, Christian: Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark. In: *International Joint Conference on Neural Networks*, 2013

[HW59] Hubel, D. H. ; Wiesel, T. N.: Receptive fields of single neurones in the cat's striate cortex. In: *The Journal of Physiology* 148 (1959), Nr. 3, S. 574–591. `http://dx.doi.org/10.1113/jphysiol.1959.sp006308`. – DOI 10.1113/jphysiol.1959.sp006308. – ISSN 1469–7793

[HZRS15a] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep Residual Learning for Image Recognition. In: *CoRR* abs/1512.03385 (2015)

[HZRS15b] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In: *CoRR* abs/1502.01852 (2015)

[IH00] Igel, Christian ; Hüsken, Michael: Improving the Rprop Learning Algorithm. In: *Proceedings of the Second International Symposium on Neural Computation, NC'2000*, 2000, S. 115–121

[IMA+16] Iandola, Forrest N. ; Moskewicz, Matthew W. ; Ashraf, Khalid ; Han, Song ; Dally, William J. ; Keutzer, Kurt: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. In: *CoRR* abs/1602.07360 (2016)

[IS15] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *CoRR* abs/1502.03167 (2015)

[JC17] Janocha, Katarzyna ; Czarnecki, Wojciech M.: On Loss Functions for Deep Neural Networks in Classification. In: *CoRR* abs/1702.05659 (2017)

[KB14] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *CoRR* abs/1412.6980 (2014)

[KDDD15] Krähenbühl, Philipp ; Doersch, Carl ; Donahue, Jeff ; Darrell, Trevor: Data-dependent Initializations of Convolutional Neural Networks. In: *CoRR* abs/1511.06856 (2015)

[KGEU08] Kus, M. C. ; Gokmen, M. ; Etaner-Uyar, S.: Traffic sign recognition using Scale Invariant Feature Transform and color classification. In: *2008 23rd International Symposium on Computer and Information Sciences*, 2008, S. 1–6

[KSH12] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*. USA : Curran Associates Inc., 2012 (NIPS'12), S. 1097–1105

[LBBH98] Lecun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE* 86 (1998), Nov, Nr. 11,

S. 2278–2324. http://dx.doi.org/10.1109/5.726791. – DOI 10.1109/5.726791. – ISSN 0018–9219

[LFJ+16] Li, Huimin ; Fan, Xitian ; Jiao, Li ; Cao, Wei ; Zhou, Xuegong ; Wang, Lingli: A high performance FPGA-based accelerator for large-scale convolutional neural networks. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, S. 1–9

[LMB+14] Lin, Tsung-Yi ; Maire, Michael ; Belongie, Serge J. ; Bourdev, Lubomir D. ; Girshick, Ross B. ; Hays, James ; Perona, Pietro ; Ramanan, Deva ; Dollár, Piotr ; Zitnick, C. L.: Microsoft COCO: Common Objects in Context. In: *CoRR* abs/1405.0312 (2014)

[LYH+13] Liang, M. ; Yuan, M. ; Hu, X. ; Li, J. ; Liu, H.: Traffic sign detection by ROI extraction and histogram features-based recognition. In: *IJCNN 2013*, 2013. – ISSN 2161–4393, S. 1–8

[QLY+16] Qian, R. ; Liu, Q. ; Yue, Y. ; Coenen, F. ; Zhang, B.: Road surface traffic sign detection with hybrid region proposal and fast R-CNN. In: *2016 12th ICNC-FSKD*, 2016, S. 555–559

[RDGF15] Redmon, Joseph ; Divvala, Santosh K. ; Girshick, Ross B. ; Farhadi, Ali: You Only Look Once: Unified, Real-Time Object Detection. In: *CoRR* abs/1506.02640 (2015)

[RDS+15] Russakovsky, Olga ; Deng, Jia ; Su, Hao ; Krause, Jonathan ; Satheesh, Sanjeev ; Ma, Sean ; Huang, Zhiheng ; Karpathy, Andrej ; Khosla, Aditya ; Bernstein, Michael ; Berg, Alexander C. ; Fei-Fei, Li: ImageNet Large Scale Visual Recognition Challenge. In: *International Journal of Computer Vision (IJCV)* 115 (2015), Nr. 3, S. 211–252

[RF16] Redmon, Joseph ; Farhadi, Ali: YOLO9000: Better, Faster, Stronger. In: *CoRR* abs/1612.08242 (2016)

[RHGS15] Ren, Shaoqing ; He, Kaiming ; Girshick, Ross B. ; Sun, Jian: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In: *CoRR* abs/1506.01497 (2015)

[RMNM16] Ribeiro, David ; Mateus, André ; Nascimento, Jacinto C. ; Miraldo, Pedro: A Real-Time Pedestrian Detector using Deep Learning for Human-Aware Navigation. In: *CoRR* abs/1607.04441 (2016)

[RORF16] Rastegari, Mohammad ; Ordonez, Vicente ; Redmon, Joseph ; Farhadi, Ali: XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In: *CoRR* abs/1603.05279 (2016)

[Rud16] Ruder, Sebastian: An overview of gradient descent optimization algorithms. In: *CoRR* abs/1609.04747 (2016)

[SEZ+13] Sermanet, Pierre ; Eigen, David ; Zhang, Xiang ; Mathieu, Michaël ; Fergus, Rob ; LeCun, Yann: OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. In: *CoRR* abs/1312.6229 (2013)

[SGH15] Schwiegelshohn, F. ; Gierke, L. ; Hübner, M.: FPGA based traffic sign detection for automotive camera systems. In: *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2015, S. 1–6

[SHK+14] Srivastava, Nitish ; Hinton, Geoffrey ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In: *Journal of Machine Learning Research* 15 (2014), S. 1929–1958

[SIV16] Szegedy, Christian ; Ioffe, Sergey ; Vanhoucke, Vincent: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In: *CoRR* abs/1602.07261 (2016)

[SL11] Sermanet, P. ; LeCun, Y.: Traffic sign recognition with multi-scale Convolutional Networks. In: *The 2011 International Joint Conference on Neural Networks*, 2011. – ISSN 2161–4393, S. 2809–2813

[SLJ+14] Szegedy, Christian ; Liu, Wei ; Jia, Yangqing ; Sermanet, Pierre ; Reed, Scott E. ; Anguelov, Dragomir ; Erhan, Dumitru ; Vanhoucke, Vincent ; Rabinovich, Andrew: Going Deeper with Convolutions. In: *CoRR* abs/1409.4842 (2014)

[SLYO17] Shi, W. ; Li, X. ; Yu, Z. ; Overett, G.: An FPGA-Based Hardware Accelerator for Traffic Sign Detection. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25 (2017), April, Nr. 4, S. 1362–1372. – ISSN 1063–8210

[SSSI12] Stallkamp, J. ; Schlipsing, M. ; Salmen, J. ; Igel, C.: Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. In: *Neural Networks* (2012), Nr. 0, S. –. – ISSN 0893–6080

[SZ14] Simonyan, Karen ; Zisserman, Andrew: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: *CoRR* abs/1409.1556 (2014)

[TGJ+14] Tompson, Jonathan ; Goroshin, Ross ; Jain, Arjun ; LeCun, Yann ; Bregler, Christoph: Efficient Object Localization Using Convolutional Networks. In: *CoRR* abs/1411.4280 (2014)

[TH12] Tieleman, T. ; Hinton, G.: *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude.* COURSERA: Neural Networks for Machine Learning, 2012

[VJ01] Viola, P. ; Jones, M.: Rapid object detection using a boosted cascade of simple features. In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001* Bd. 1, 2001. – ISSN 1063–6919, S. I–511–I–518 vol.1

[WLL+13] Wu, Y. ; Liu, Y. ; Li, J. ; Liu, H. ; Hu, X.: Traffic sign detection based on convolutional neural networks. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*, 2013. – ISSN 2161–4393, S. 1–7

[XHL16] Xu, Bing ; Huang, Ruitong ; Li, Mu: Revise Saturated Activation Functions. In: *CoRR* abs/1602.05980 (2016)

[XWCL15] Xu, Bing ; Wang, Naiyan ; Chen, Tianqi ; Li, Mu: Empirical Evaluation of Rectified Activations in Convolutional Network. In: *CoRR* abs/1505.00853 (2015)

[ZF13] Zeiler, Matthew D. ; Fergus, Rob: Visualizing and Understanding Convolutional Networks. In: *CoRR* abs/1311.2901 (2013)

[ZYZ+17] Zuo, Z. ; Yu, K. ; Zhou, Q. ; Wang, X. ; Li, T.: Traffic Signs Detection Based on Faster R-CNN. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, S. 286–288

[ZZZ+16] Zang, D. ; Zhang, J. ; Zhang, D. ; Bao, M. ; Cheng, J. ; Tang, K.: Traffic sign detection based on cascaded convolutional neural networks. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016, S. 201–206

Internet References

[1] *http://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.* Available at `http://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/` (9/2017).

[2] *ImageNet Large Scale Visual Recognition Challenge 2013 (ILSVRC2013).* Available at `http://www.image-net.org/challenges/LSVRC/2013/results.php` (9/2017).

[3] Berkeley Artificial Intelligence Research (BAIR). *Caffe - Deep learning framework (Sum-of-Squares / Euclidean Loss Layer).* Available at `http://caffe.berkeleyvision.org/tutorial/layers/euclideanloss.html` (9/2017).

[4] COCO Consortium. *COCO - Common Objects in Context dataset.* Available at `http://cocodataset.org/#home` (9/2017).

[5] Deshpande, Adit. *The 9 Deep Learning Papers You Need To Know About.* Available at `https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html` (8/2017).

[6] L. Fei-Fei, A. Karpathy, and J. Johnson. Lecture 5, Slide 100 — Track the ratio of weight updates / weight magnitudes, 2016.

[7] Kaggle Inc. *Kaggle - ImageNet Object Localization Challenge.* Available at `https://www.kaggle.com/c/imagenet-object-localization-challenge` (9/2017).

[8] Karpathy, Andrej. *Stanford.edu, cs231n - Convolutional Neural Networks.* Available at `http://cs231n.github.io/convolutionalnetworks/` (8/2017).

[9] Karpathy, Andrej. *Stanford.edu, cs231n - Linear Classification.* Available at `http://cs231n.github.io/linear-classify/` (8/2017).

[10] Karpathy, Andrej. *Stanford.edu, cs231n - Neural Networks 1.* Available at `http://cs231n.github.io/neural-networks-1/` (8/2017).

[11] Karpathy, Andrej. *Stanford.edu, cs231n - Neural Networks 2.* Available at `http://cs231n.github.io/neural-networks-2/` (8/2017).

[12] Karpathy, Andrej. *Stanford.edu, cs231n - Neural Networks 3.* Available at `http://cs231n.github.io/neural-networks-3/` (8/2017).

[13] Krizhevsky, Alex. *CIFAR-10 and CIFAR-100 dataset.* Available at `https://www.cs.toronto.edu/~kriz/cifar.html` (8/2017).

[14] Krizhevsky, Alex; Nair, Vinod; Hinton, Geoffrey. *80 Million Tiny Images - Visual Dictionary.* Available at `http://groups.csail.mit.edu/vision/TinyImages/` (8/2017).

[15] Li, Fei-Fei; Karpathy, Andrej; Johnson, Justin. *Stanford cs231n - Lecture 8: Spatial Localization and Detection (2016).* Available at `http://cs231n.stanford.edu/slides/2016/winter1516_lecture8.pdf` (7/2017).

[16] Mordor Intelligence. *Advanced Driver Assistance Systems (ADAS) Mar-*

ket. Available at `https://www.mordorintelligence.com/industry-reports/advanced-driver-assistance-systems-market` (12/2017).

[17] PASCAL2 Network of Excellence on Pattern Analysis, Statistical Modelling and Computational Learning. *The PASCAL Visual Object Classes Homepage.* Available at `http://host.robots.ox.ac.uk/pascal/VOC/` (9/2017).

[18] Presentation by Xilinx. *Embedded Vision Alliance - "Caffe to Zynq: State-of-the-Art Machine Learning Inference Performance in Less Than 5 Watts,".* Available at `https://www.embedded-vision.com/platinum-members/xilinx/embedded-vision-training/videos/pages/may-2017-embedded-vision-summit-kathail` (7/2017).

[19] Princeton University. *Princeton University "About WordNet".* Available at `http://wordnet.princeton.edu/` (9/2017).

[20] Ruhr-Universität Bochum, Institut für Neuroinformatik. *German Traffic Sign Benchmark.* Available at `http://benchmark.ini.rub.de/dev/index.php?section=home&subsection=news` (5/2017).

[21] Stanford Vision Lab, Stanford University, Princeton University. *ImageNet.* Available at `http://www.image-net.org/index` (8/2017).

[22] Statista - The Statistics Portal. *Estimated size of the global Level 1 ADAS market between 2016 and 2025 (in million U.S. dollars).* Available at `https://www.statista.com/statistics/789583/estimated-global-adas-market-growth/` (12/2017).

[23] Timofte, Radu. *BelgiumTS Dataset.* Available at `http://btsd.ethz.ch/shareddata/` (5/2017).

[24] Wikimedia Foundation, Inc. *Boosting (machine learning).* Available at `https://en.wikipedia.org/wiki/Boosting_(machine_learning)` (11/2017).

[25] Xilinx Inc. *Xilinx Virtex-7 FPGA VC709 Connectivity Kit.* Available at `https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html` (9/2017).

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 15.Februar 2018 _____

Martin Lechner BSc