

# CHALMERS



## Convolutional Networks for Traffic Sign Classification

Master's thesis in Communication Engineering

FEDERICO ZANETTI

Department of Signals and Systems  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Göteborg, Sweden 2016  
Master's Thesis EX096/2016



MASTER'S THESIS EX096/2016

**Convolutional Networks  
for Traffic Sign Classification**

FEDERICO ZANETTI



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Signals and Systems  
Signal Processing  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Göteborg, Sweden 2016

Concolutional Networks Traffic Sign Classification  
FEDERICO ZANETTI

© FEDERICO ZANETTI, 2016.

Supervisor: Irene Yu-Hua Gu, Signals and Systems  
Examiner: Irene Yu-Hua Gu, Signals and Systems

Master's Thesis EX096/2016  
Department of Signals and systems  
Signal Processing  
Chalmers University of Technology  
SE-412 96 Göteborg  
Telephone +46 31 772 1000

Typeset in L<sup>A</sup>T<sub>E</sub>X  
Gothenburg, Sweden 2016

# **Abstract**

Traffic sign classification is an important task in autonomous driving and assistant driving systems. In this thesis we do automatic learning of features and classification on traffic signs from images. First, we study several publicly available libraries for deep learning. Several CNN architectures are then tested under different parameter settings and scenarios, such as network depth, filter size, dropout rate and preprocessing by using original images and segmented images. The German Traffic Sign Recognition Benchmark was used to train in a supervised way the CNN model. Preprocessing and segmentation are tested to make the training more robust and the network able to generate more independent features. The results obtained are good for all study cases and all 43 traffic sign classes. We reached test accuracies above 98% that are comparable to state of the art performances.

Keywords: Convolutional Neural Networks, Traffic Sign Classification, Supervised Learning, German Traffic Sign Recognition Benchmark



## Acknowledgements

I would like to express my sincere gratitude to my supervisor and examiner Prof. Irene Yu-Hua Gu for constantly guiding and motivating me in this Master Thesis Project with great dedication. My sincere thanks also goes to my Professor Nicola Conci for opening the opportunity to develop this Thesis in a partner University. This thesis was carried out through Erasmus financial support that allowed me as an undergraduate student from the University of Trento to study in the host University, Chalmers University of Technology. I want to thank the people from the Signals and Systems department for creating a stimulating working environment and the people behind the Erasmus program that patiently guided me through all the steps to make this experience possible. I want to thank Cristina Rigato, PhD student at the Biomedical signals and systems research group for welcoming me to Chalmers and helping me from the beginning to the end of this thesis. My sincere gratitude to Doug Boston, a great man who has taught me a lot and has inspired me in life. He has contributed a lot to my education and has made me a better student and a better person. Many thanks to my friends who never fail to cheer me up and are always there for me even from a great distance. I did miss them during this time and I will be very happy to see them in person again. I am very grateful to my family for their endless support in the good and in the bad times. I would not have been able to make it this far without you and I am very happy because you gave me this huge opportunity. I promise I will always remember it and I will be there for you in the future.

Federico Zanetti, Göteborg, October 2016



# Contents

<b>List of Figures</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.0.1 Outline . . . . .	2
<b>2 Background theories and methods</b>	<b>3</b>
2.1 From Neural Networks to Deep Learning . . . . .	3
2.1.1 Advantages of deep learning . . . . .	4
2.2 Deep learning tools and libraries . . . . .	5
2.2.1 Theano and Theano wrappers . . . . .	5
2.2.2 TensorFlow, Caffe, Torch . . . . .	6
2.3 Deep learning models . . . . .	7
2.3.1 Convolutional Neural Network (CNN) . . . . .	8
2.3.2 Residual Neural Network (ResNet) . . . . .	9
2.3.3 Recurrent Neural Network (RNN) . . . . .	10
2.3.4 Long Short-Term Memory (LSTM) . . . . .	11
2.4 Training a neural network . . . . .	12
2.4.1 Backpropagation . . . . .	13
2.4.2 Initialization of the weights and Glorot Initialization . . . . .	15
2.4.3 Sigmoid and rectified linear activation function . . . . .	15
2.4.4 Cross-entropy cost function . . . . .	16
2.4.5 Update criterions . . . . .	17
2.5 Training a Convolutional Network . . . . .	19
2.5.1 Backpropagation algorithm for CNNs . . . . .	19
2.5.2 Using dropout . . . . .	21
2.5.3 Online and batch training . . . . .	21
2.5.4 Using normalization . . . . .	22
2.6 Growcut segmentation preprocessing . . . . .	22
<b>3 Work performed in this thesis</b>	<b>24</b>
3.0.1 Hyperparameters . . . . .	24
3.0.2 Training the model . . . . .	27
3.1 Case studies . . . . .	27
3.1.1 Case study 1 (CS1): raw images . . . . .	27
3.1.2 Case study 2 (CS2): clipped images . . . . .	28
3.1.3 Case study 3 (CS3): segmentation . . . . .	28

<b>4 Results</b>	<b>30</b>
4.1 Dataset description . . . . .	30
4.2 Experimental setup . . . . .	32
4.3 Experimental results . . . . .	34
<b>5 Conclusions</b>	<b>47</b>
5.1 Observations on the model and training . . . . .	47
5.1.1 The model . . . . .	47
5.1.2 The training. . . . .	48
5.2 Discussion . . . . .	48
<b>Bibliography</b>	<b>51</b>

# List of Figures

1.1	Samples from GTSRB . . . . .	2
2.1	Theano and its wrappers . . . . .	6
2.2	Overview on major libraries for Deep Learning . . . . .	7
2.3	Zero padding example . . . . .	9
2.4	Building block for residual learning . . . . .	9
2.5	Basic RNN structure . . . . .	11
2.6	RNN structure . . . . .	12
2.7	Neural connections . . . . .	13
2.8	Activation functions . . . . .	16
2.9	Momentum update . . . . .	18
2.10	Convolutional layers . . . . .	20
2.11	Dropout . . . . .	21
2.12	Batch training . . . . .	22
3.1	CNN architecture . . . . .	26
3.2	Block diagram for CS1. . . . .	28
3.3	Block diagram for CS2. . . . .	28
3.4	Block diagram for CS3. . . . .	28
3.5	Segmented samples . . . . .	29
4.1	All 43 benchmark classes in GTSRB. . . . .	30
4.2	Relative class frequencies . . . . .	31
4.3	Histogram of dimensions . . . . .	31
4.4	All 43 benchmark classes in GTSRB. . . . .	33
4.5	Test accuracy of CS1 . . . . .	34
4.6	The classification errors in CS1. . . . .	34
4.7	Confusion matrix CS1 (1) . . . . .	36
4.8	Confusion matrix CS1 (2) . . . . .	37
4.9	Test accuracy of CS2 . . . . .	38
4.10	The classification errors in CS2. . . . .	38
4.11	Confusion matrix CS2 (1) . . . . .	40
4.12	Confusion matrix CS2 (2) . . . . .	41
4.13	Test accuracy of CS3 . . . . .	42
4.14	The classification errors in CS3. . . . .	42
4.15	Confusion matrix CS3 (1) . . . . .	44
4.16	Confusion matrix CS3 (2) . . . . .	45

4.17 Example of data volume . . . . .	46
5.1 Accuracies by class groups . . . . .	49

# 1

## Introduction

Traffic signs have been designed to be easily readable for humans, who perform very well at this task. For computer systems, however, classifying traffic signs still seems a challenging task [37]. Checking the road ahead, what is behind and the traffic, all while trying to maintain the speed may sometimes become quite difficult. Car firms are constantly looking to introduce new technologies to make this task easier and the development of Traffic Sign Recognition (TSR) systems can be considered a challenging real-world problem of high industrial relevance nowadays. Its purpose is to support the driver so that the chances of not noticing a change in speed limit or the warning of a potential hazard ahead shall be vastly reduced. TSR is also one of the foremost important integral parts of autonomous vehicles and Advanced Driver Assistance Systems (ADAS) [2], [35]. Earlier work on TSR employed Bayesian classifiers [42] or boosting [43]. Other works focus on ensemble classifiers, such as one-to-all strategy with support vector machines (SVMs) as base classifiers [38], [39], [40] and random forests with K-d trees as weak classifiers [41]. However, due to their underlying mechanism of binary classification, these methods have to face the unbalance between the number of positive and negative training samples. As a result, these methods are likely to achieve a local optimum or an over-fitting solution. In recent work, convolutional neural networks (CNNs) [33], [?], have been used to automatically learn feature representations of traffic signs. These DNN algorithms combine feature extraction and classification into a unified neural network. CNN methods are frequently considered using hand-crafted features such as a circle detector [42], a Haar wavelet [43], and a histogram of oriented gradient (HOG) [44] or scale-invariant feature transform (SIFT) in [43], [3]. However, designing such features requires a good deal of time and it is hard to know what feature is robust to a specific task. Traffic signs may be divided into different categories according to function, and in each category they may be further divided into subclasses with similar generic shape and appearance but different details. This suggests traffic-sign recognition should be carried out as a two-phase task: detection followed by classification. The detection step uses shared information to suggest bounding boxes that may contain traffic-signs in a specific category, while the classification step uses differences to determine which specific kind of sign is present. Since the launch of the German traffic-sign detection and classification benchmark data [36], [37], various research groups have made progress in both the detection (GTSDB) [37] task and classification (GTSRB) [36] task. Current methods achieve near perfect results for both tasks, with 100% recall and precision for detection and 99.65% [46] precision for classification. Jin *et al.* [46] in particular train an ensemble of CNNs and do classification by averaging the decisions of the ensemble. Recently Haloi

[47] has reached 99.81% accuracy by using a spatial transformer network capable of generating automatic transformation of the input images.

In this work the objective is threefold:

1. to gain knowledge and understanding deep learning, in particular, convolutional Neural Networks (CNNs);
2. use deep learning to automatically classifying traffic signs. In particular, in this work we use the standard German Traffic Sign Benchmark Dataset (GTSRB) for training and testing.
3. compare classification results obtained with CNNs on raw GTSRB images and those obtained with CNNs on enhanced images.

### 1.0.1 Outline

This Thesis report is organized as follows: Section 2 describes the theoretical background on deep learning architectures and their training and the tools that allow for a fast and optimized implementation of neural networks. Section 3 describes the implementation of the neural network architecture and training and the different settings on which the GTSRB training was carried out. Section 4 shows the experimental results and their evaluation. At last future work is discussed.



**Figure 1.1:** Samples from the German Traffic Sign Recognition Benchmark. Image from [44].

# 2

## Background theories and methods

This section reviews some existing theories and methods related to this thesis work. First we provide a short background on the evolution of deep learning until now. Second, the main libraries for deep learning and the advantages each one provides are investigated. After that, we describe the most relevant architectures for deep learning and their suitability for Traffic Sign Classification. The following section is about the general backpropagation and the last one talks about the training in convolutional networks.

### 2.1 From Neural Networks to Deep Learning

The core of an AI program for feature extraction and classification is the mapping of a set of virtual neurons and the assignment of numerical values or “weights” to the connections between them. Neural networks employ an iterative learning process in which data vectors are presented to the network one at a time, and the weights associated to the input values are adjusted each time. During the learning phase, the network learns by adjusting the weights in order to be able to predict the correct class label of input samples. The most popular neural network (NN) training algorithm is the back-propagation algorithm that was proposed in the 1980’s, and it was in the 80s that the concept of Deep Neural Networks was born as well. The first true, practical application of backpropagation came about through the work of LeCun in 1989 at Bell Labs. He used convolutional networks in combination with backpropagation to classify handwritten digits (MNIST). Despite the expectations on NNs, funding for research was still scarce. Another important advance that was made in this time is the long short-term memory (LSTM) for Recurrent Neural Networks (RNN) by Hochreiter and Schmidhuber in 1997, but these advances went mostly unnoticed until later as they were overshadowed by the Support Vector Machine (SVM) developed by Cortes and Vapnik in 1995.

A big shift occurred as computers grew faster and later thanks to the introduction of graphics processing units (GPUs). Neural networks can be slow when compared to SVMs, but they reach much better results with the same amount of data. As the speed of GPUs increased, it was possible to train deep networks such as convolutional networks without the help of pretraining as demonstrated by Ciresan and colleagues in 2011 and 2012 in works such as [33]. They managed to win character recognition, traffic sign, and medical imaging competitions with their convolutional network architecture. Krizhevsky, Sutskever, and Hinton used a similar architectures in 2012 (like AlexNet) that also features rectified linear activation functions

and dropout for regularization. They received outstanding results in the ILSVRC-2012 ImageNet competition, which marked the abandonment of feature engineering and the adoption of feature learning in the form of deep learning. Google, Facebook, and Microsoft noticed this trend and made major acquisitions of deep learning startups and research teams between 2012 and 2014. In June 2015 Google demonstrated one of the largest neural networks yet, with more than a billion connections. A team led by Stanford computer science professor Andrew Ng and Jeff Dean showed to the system images from 10 million randomly selected YouTube videos. The simulated neurons in the software model were fixed on images such as of human faces, dogs, cats, yellow flowers and other objects. Thanks to the power of deep learning, the system identified these discrete objects even though no humans had ever defined or labeled them. Currently, research in deep learning is still rapidly growing.

### 2.1.1 Advantages of deep learning

The human brain does not interpret an image by pixel but it decomposes a problem into sub-problems through multiple levels of interpretations. As shown in [1], the human brains processes visual signals through a structure of multiple layers, well represented by Neural Networks. One of the promises of deep learning is replacing handcrafted features with unsupervised or semi-supervised feature learning and hierarchical feature extraction. Research in this area attempts to make better representations and create models to learn these representations from large-scale data. One of the most striking facts about neural networks is that they can compute any function at all. No matter what the function, it is guaranteed there is a neural network so that for every possible input  $x$ , the value  $f(x)$  or some close approximation is output from the network. Theoretical results (works such as [34]) show that an architecture with an insufficient depth can require many more computational elements and the growth in the number of these is exponential with respect to input size. This also causes a slower learning. Architectures with multiple levels facilitate the sharing and re-use of components. As pointed out in [4], the meaning of filters in deep learning roots in the principle of shared weights and biases. It means that all the neurons in the first hidden layer detect different features at different locations in the input image. For this reason, sometimes the map from the input layer to the hidden layer is called a feature map. The weights defining the feature map are called *shared weights* and the bias *shared bias*. The shared weights and bias are often said to define a kernel or filter on a *convolutional layer*. A big advantage of sharing weights and biases is that it greatly reduces the number of parameters in the network. While on one side kernels allow to make use of the convolution operation, on the other networks take advantage of the *pooling layers*. A pooling layer is a layer that takes each feature map output from a convolutional layer and prepares a condensed feature map, which contributes to the successful hierarchical representation of a CNN.

## 2.2 Deep learning tools and libraries

There is a number of open-source tools on the internet that supports deep learning. In the next section we discuss the most promising of them in terms of reliability, performances and usability in order to select one of them for the final implementation for this project.

### 2.2.1 Theano and Theano wrappers

Theano and its high level wrappers form a sort of family of libraries that share similar characteristics. Since Theano is not actually a machine learning library but rather a low level optimization library for computational graphs, there are many other libraries that envelop it.

**Theano:** Theano is a Python library that has been developed at the University of Montreal since 2008. It is an optimized tensor manipulation library that is thought to serve as “backend engine” of high-level Theano wrappers. Theano is centered around the idea of computational graphs: it knows how to take a computational graph structure and turn it into very efficient code that uses SciPy-NumPy, native libraries like BLAS and native C++ code to run as fast as possible on CPUs or GPUs. Theano offers speed and stability optimisations since it internally reorganizes and optimises the computations. One of its perks is automatic differentiation: it only takes to implement the forward (prediction) part of the model, and Theano will automatically figure out how to calculate the gradients at various points, allowing users to perform gradient descent for model training. Another key aspect of Theano is the use of GPU for computations. The use of GPU in Theano is actually transparent, in the sense that users can write the same code and run it either on CPU or GPU. More specifically, Theano figures out which parts of the computation should be moved to the GPU. Theano is not actually a machine learning library, as it does not provide the user with pre-built models that may be trained on the dataset. Instead, it is a mathematical library that provides tools to build custom machine learning models. Using Theano makes it easy to implement backpropagation for convolutional networks and Recurrent Networks in general, as it is thought to model the NN as a computational graph. For this reason it provides the “ifelse” or “switch” functions to allow for conditional control flow in the graph. Loops are made easy with a “scan” function (which favors RNN implementation).

**Lasagne:** Lasagne is a Python lightweight library built on top of Theano, basically an high level wrapper for Theano. While another Theano wrapper such as Keras can be used without ever "knowing" about Theano sitting underneath it, this is not true for Lasagne, which is designed to be used side by side with Theano and to interface with it. Lasagne has been implemented with a special focus on transparency: it does not hide Theano behind abstractions because it directly processes and returns Theano expressions or Python numpy data types. Lasagne allows architectures of multiple inputs and multiple outputs and provides optimization methods including stochastic gradient descent, Nesterov momentum, Adagrad, Adadelta and Adam.

The cost function is easy to define and there is no need to derive gradients due to Theano’s symbolic differentiation. Another useful characteristic of Lasagne is its modularity: it allows all parts (layers, weights, filters...) to be used independently of Lasagne, being saved and easily exported for re-use.

**Other Theano wrappers:** Other Theano wrappers include Pylearn2, Bloaks and Keras. Pylearn2 algorithms can be written using Theano expressions and Theano optimizes and stabilizes the expressions. It includes all the things needed for multilayer perceptrons, RBMs and CNNs. Blocks is a framework that introduces the concept of “bricks” to build models. Bricks are parametrized Theano operations. Blocks includes interesting features such as the monitoring and analysis of values during training progress and automatic saving and resuming of the training. Keras is a highly modular neural networks library, written in Python and capable of running on top of either TensorFlow or Theano (supports the Theano backend and the TensorFlow backend as well). It was developed with the focus on enabling fast experimentation, since being able to move easily from idea to result is key to doing efficient research.

Lasagne	Keras	Blocks	Pylearn2
Theano 0.8.2			
Ubuntu 16.04.1 LTS			

**Figure 2.1:** Overview on Theano and its wrappers. The Linux installation is the preferable choice.

### 2.2.2 TensorFlow, Caffe, Torch

The number of available libraries for Deep Learning is constantly growing. TensorFlow, Caffe and Torch are arguably the other three major libraries when it comes to interface, optimization, training speed, scalability and readability of the code.

**TensorFlow:** TensorFlow is an open source software library for numerical computation developed by the Google Brain Team within Google’s Machine Intelligence research organization. In a way it is very similar to Theano, for its central idea of data flow graphs. Nodes in the graph represent mathematical operations and the graph edges represent the multidimensional data arrays (tensors) propagated between them. TensorFlow can be used from C, C++ and Python programs. TensorFlow also allows a graphical visualization of the graph (“TensorBoard” tool) and to scope on sub-levels of computations for debugging purposes.

**Caffe:** Caffe is a deep learning framework that was developed by the Berkeley Vision and Learning Center (BVLC) mostly for vision tasks. Caffe takes its first steps from the AlexNet implementation and it is implemented mostly in C++; it was originally intended for CNNs and provides easy implementation of feedforward

NN. It is notoriously good for finetuning existing models, although it is not as well as good for RNNs. It requires a number of software packages such as SciPy-NumPy, BLAS, Intel MKL, OpenBLAS. Caffe has some level of abstraction, which is higher than Cuda-Convnet and lower than others such as Pylearn and Torch. Caffe includes the so called “Model Zoo” providing lots of pre-trained models.

**Torch:** Torch is heavily used by Facebook AI Research Labs and Google’s Deep-Mind Lab. Instead of following the Python trend, Torch is C and Lua-based. Lua is a high level scripting language intended for embedded devices. Torch allows to build arbitrary graphs of neural networks, and parallelize them over CPUs and GPUs in an efficient manner. Torch has a large ecosystem of community-driven packages in machine learning, computer vision, signal processing, parallel processing, image and video. Torch gravitates around the “Tensor” class with great analogies to the Numpy arrays with the same role as tensors in Theano. It allows to define a deep network in a sequential way as a stack of layers, similarly to Theano and Caffe, and to cast the types and computation on a GPU. In the end Torch has less Plug and Play functionalities than Caffe, as it results in more self coding work which on the other side also allows for more flexibility. It has a lot of modular pieces that can be combined and also offers lots of pre-trained models.

	Caffe	Torch	Theano	TensorFlow
<b>Language</b>	C++, Python	Lua	Python	Python
<b>Pretrained</b>	Yes ++	Yes ++	Yes (Lasagne)	Inception
<b>Multi-GPU: Data parallel</b>	Yes	Yes <small>cunn. DataParallelTable</small>	Yes <small>platoon</small>	Yes
<b>Multi-GPU: Model parallel</b>	No	Yes <small>fbcunn.ModelParallel</small>	Experimental	Yes (best)
<b>Readable source code</b>	Yes (C++)	Yes (Lua)	No	No
<b>Good at RNN</b>	No	Mediocre	Yes	Yes (best)

**Figure 2.2:** Overview on major libraries for Deep Learning. Image taken from the course CS231: Convolutional Neural Networks for Visual Recognition from Stanford University.

## 2.3 Deep learning models

Once the choice on the software is made, it is necessary to figure out which architecture is the best for Traffic Sign Classification. We consider the following options.

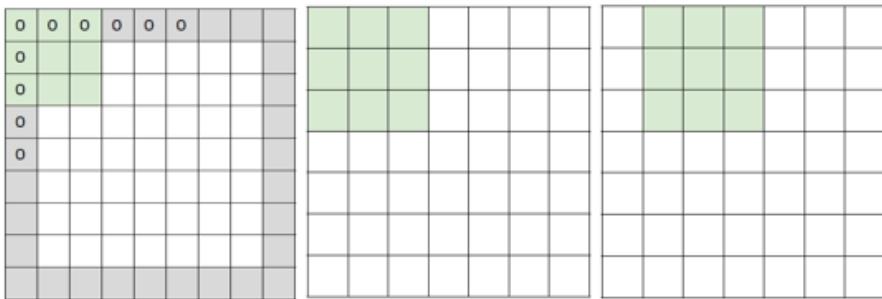
### 2.3.1 Convolutional Neural Network (CNN)

CNN is one of the neural network models for deep learning, which can be described by three specific characteristics, namely locally connected neurons, shared weights and spatial or temporal sub-sampling. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. A benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. To provide an overview on the architecture characteristics, CNNs are composed of convolutional and subsampling layers optionally followed by fully connected layers. The input to a convolutional layer is an  $m \times m \times r$  image where  $m$  is the height and width of the image and  $r$  is the number of channels, e.g. an RGB image has  $r = 3$ . The convolutional layer will have  $k$  filters of size  $n \times n \times q$  where  $n$  is smaller than the dimension of the image and  $q$  can either be the same as the number of channels  $r$  or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure where each element is convolved with the image to produce  $k$  feature maps of size  $mn + 1$ . In convolutional layer, each neuron is connected locally to its inputs of the previous layer, which functions like a 2D convolution with a certain filter, then its activation could be computed as the result of a nonlinear transformation:

$$\alpha_{i,j} = \rho(f * x) = \rho\left(\sum_{i'=1}^n \sum_{j'=1}^n f_{i',j'} x_{i+i',j+j'} + b\right) \quad (2.1)$$

where  $f$  is an  $n \times n$  weight matrix of the convolutional filter,  $x$  refers to the activations of the input neurons connected to the neurons  $(i,j)$  in the following convolutional layer.  $\rho()$  is a nonlinear activation function (usually sigmoid or hyperbolic tangent),  $b$  is the bias,  $*$  is the convolution operator. After the convolution each map may be subsampled typically with mean or max pooling over  $p \times p$  contiguous regions where  $p$  ranges between 2 for small images (e.g. MNIST) and is usually not more than 5 for larger inputs. As pointed out in [10] CNNs have won several competitions in:

1. handwriting recognition (MNIST, Arabic HWX - ISDIA);
  2. OCR in the wild (2011), Streetview house numbers(NYU);
  3. Traffic sign recognition (2011), GTSRB competition (IDSIA, NYU);
  4. Pedestrian detection (2013), INRIA datasets and others (NYU);
  5. Object recognition (2012), ImageNet competition;
  6. Human Action Recognition (2011), Hollywood II dataset(Stanford);
- and many more.



**Figure 2.3:** On the left: an example of zero-padding with 1-pixel border. Middle and right: areas superimposed on subsequent multiplications during stride-1 convolution. Images from [24].

One aspect of CNNs to pay attention to is to preserve the size of the data throughout the network which would otherwise decrease without zero padding. Without zero-padding the size of the output  $N_{out}$  is:

$$N_{out} = \frac{N_{in} - F}{Stride} + 1 \quad (2.2)$$

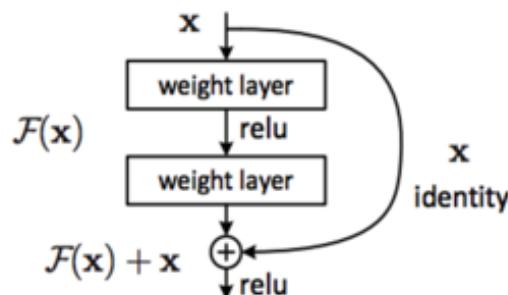
For this reason, having stride 1, to preserve the size of the input data past the convolutional layer, zero-pad  $P$  is set according to:

$$P = \frac{F - 1}{2} \quad (2.3)$$

For the maximum pooling we choose 2x2 kernels at stride 2 in order to avoid overlapping.

### 2.3.2 Residual Neural Network (ResNet)

A variation of standard CNNs are Deep Residual Neural Networks. A description and a model for ResNets can be found in [17]. ResNets take a standard feed-forward ConvNet and add skip connections that bypass (or shortcut) a few convolution layers at a time. Each bypass gives rise to a residual block in which the convolution layers predict a residual that is added to the following block's input. [17] proves that these networks can gain accuracy from considerably increased depth. A residual network 8 times deeper than VGG net [22] but still having lower complexity compared to it has won the first place in ILSVRC 2015.



**Figure 2.4:** A building block for residual learning. Image from [17].

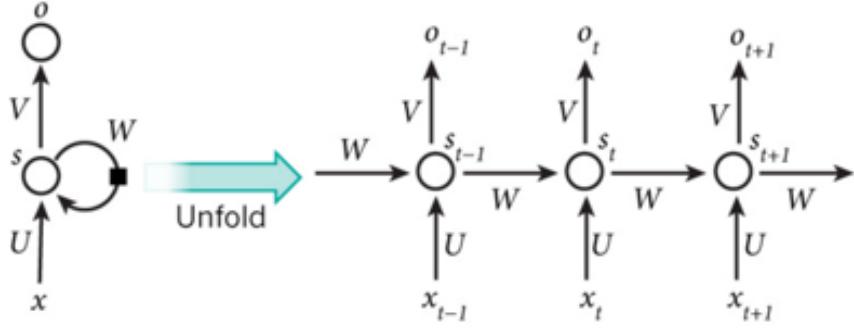
Residual layers address a problem of degradation that occurs as the depth increases and this problem is not due to overfitting [17]. The accuracy saturates first and then degrades rapidly. When extra-layers turn out to be unnecessary or even cause of degradation, an identity mapping between layers is the most effective solution. A residual connection allows to push the residual to zero and propagate along the stack an identity mapping, thus “hiding” the presence of extra (unnecessary) layers. To the extreme, if an identity mapping were optimal to a specific training propagation, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers. Along with the depth degradation, the internal covariate shift issue has been considered with ResNets. A thorough explanation is given in [27]. The internal covariate shift is caused by the fact that the distribution of each layer’s inputs changes during training along with the change in the network parameter’s values. As a consequence the training is a slower because it forces lower learning rates and makes it hard to train networks with saturating nonlinearities [27]. Batch normalization fixes the distribution of the layer inputs as the training progresses. [27] explains that the whitening of the input (linearly transformed to have zero mean and unit variance) makes the network training converge faster. Since mini-batches are used in stochastic gradient training, each mini batch produces estimates of the mean and variance of each activation. The statistics used in normalization then can be thought as fully participating to the gradient backpropagation. [17] uses in fact, the batch normalization from [27]. On top of that, a scale and shift transformation of type:

$$y_{k+1} = y_k * \text{norm}(x) + \beta_k \quad (2.4)$$

is being added to every residual learning block. The reason is that since full whitening of each input layer is costly, the simplification of normalizing each feature independently is made. This will though alter what a layer represents. Scaling and shifting trasformations allow the network to be able to represent the identity transformation.

### 2.3.3 Recurrent Neural Network (RNN)

As explained in [11], the idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that may be a bad idea. To predict the next word in a sentence it is better to know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output made dependent on the previous computations. Another way to think about RNNs is that they have a “memory” which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps. In Figure 2.5 can be seen the basic structure of a RNN:



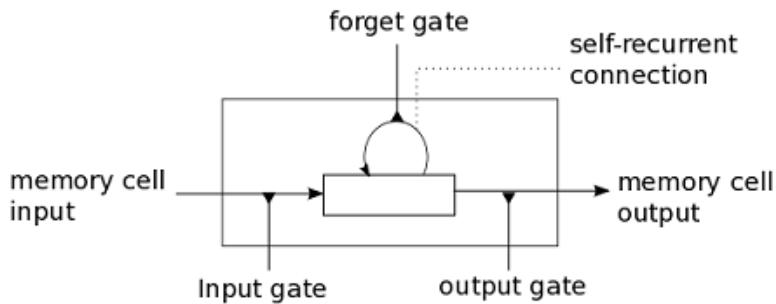
**Figure 2.5:** Basic RNN structure. Image from [11].

$x_t$  is the input at time step  $t$ . For example,  $x_1$  could be a vector corresponding to the second frame of a video.  $s_t$  is the hidden state at time step  $t$ . It is the “memory” of the network.  $s_t$  is calculated based on the previous hidden state and the input at the current step:  $s_t = f(U_{xt} + W_{st-1})$ . The function  $f$  usually is a nonlinearity such as tanh or a rectifier.  $s_{t-1}, \dots, s_{t-n}$ , which are required to calculate the first hidden state, are typically initialized to all zeroes.  $o_t$  is the output at step  $t$ . For example, to make prediction on the next frame in a video it would be a vector of probabilities across all possible pixels (the descriptors of the frame) in use in the given system. The literature around RNN is very wide. In Bi-directional RNN [16], training is accomplished by forwarding the data simultaneously in positive and negative time direction. Multi-dimensional RNN, introduced in [17], extend RNNs one-dimensional input data to multi-dimensional data, thereby extending the potential applicability of RNNs to vision, video processing, medical imaging and many other areas. Gated Feedback Recurrent Neural Networks (GFRNN) [18] extends the existing approach of stacking multiple recurrent layers by allowing and controlling signals flowing from upper recurrent layers to lower layers.

### 2.3.4 Long Short-Term Memory (LSTM)

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. RNNs can basically learn to use the past information. But there are also cases where more context is needed. Considering to try to predict the last word in the text “I grew up in Sweden... I speak fluent ...”. Recent information suggests that an upcoming word is probably the name of a language, but to narrow down which language, we need the context of Sweden, from further back. It is entirely possible for the gap between the relevant information and the point where it is needed to become very large. LSTMs are explicitly designed to avoid the long-term dependency problem. As explained in [20], in a traditional recurrent neural network, during gradient back-propagation step, the gradient values can end up being multiplied a large number of times by the weight matrix associated with the connections between the neurons of the recurrent hidden layer. This means that the magnitude of weights in the transition matrix can have a strong impact on the learning process. If the weights in this matrix are small (or, more formally,

if the leading eigenvalue of the weight matrix is smaller than 1.0), it can lead to a situation called vanishing gradients where the gradient signal becomes so small that learning either becomes very slow or stops working altogether. It can also make more difficult the task of learning long-term dependencies in the data. Conversely, if the weights in this matrix are large (or the leading eigenvalue of the weight matrix is larger than 1.0), this can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This event is often referred to as exploding gradients.



**Figure 2.6:** Basic RNN structure. Image from [wildml.com](http://wildml.com).

These issues are the main motivation behind the LSTM model, introduced in [2] in 1997. This model makes use of a new structure called a memory cell. A memory cell is composed of four main elements: an input gate, an output gate, a neuron with a connection to itself and a forget gate. The self-recurrent connection has a weight of 1.0 and ensures that, barring any outside interference, the state of a memory cell can remain constant from one time step to another. The gates serve to modulate the interactions between the memory cell itself and the environment. The input gate can allow incoming signal to alter the state of the memory cell or block it. On the other hand, the output gate can allow the state of the memory cell to have an effect on other neurons or prevent it. And last, the forget gate can modulate the memory cell's self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.

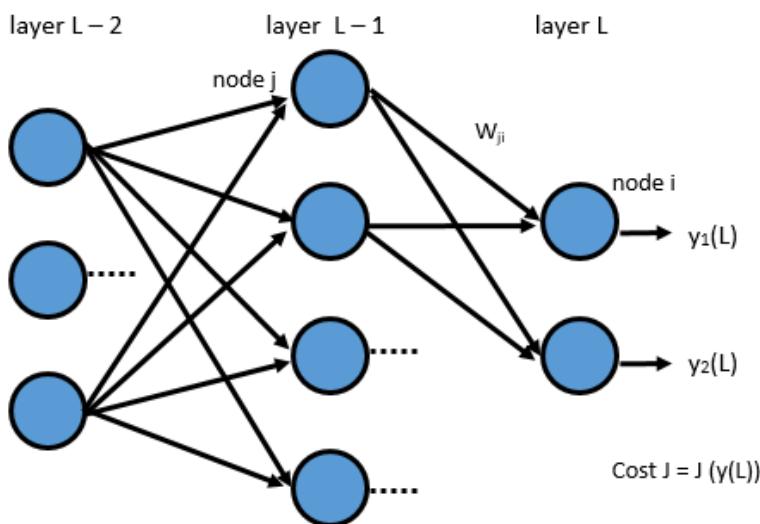
## 2.4 Training a neural network

In the following section the backpropagation algorithm and other training and network parameters are discussed. We also focus mostly on specific parameters that are favored in this thesis. For example, we speak about weights initialization and in particular the Glorot initialization. We discuss the use of sigmoid activation function in order to compute the errors at the output of the network. An expression that returns a measure of the error that is commonly used is the categorical crossentropy function. Backpropagation explains how from the errors the updates are obtained. Specifically, Nesterov momentum is eventually applied to the updates to obtain the new values for the weights.

### 2.4.1 Backpropagation

Backpropagation is a common method for training a neural network. Its goal is to compute the partial derivatives  $\partial w$  and  $\partial C/\partial b$  of a cost function  $C$  with respect to any weight  $w$  or bias  $b$  in the network. For backpropagation to work two main assumptions about the form of the cost function must be made:

1. the cost function can be written as an average  $J = \frac{1}{n} \sum_x J_x$  over cost functions  $J_x$  for individual training examples,  $x$ ;
2. the second assumption is that the cost shall be written as a function of the outputs from the neural network.



**Figure 2.7:** An example model of neural connections.

The reason for the first assumption is that what backpropagation actually allows to do is compute the partial derivatives  $\partial J_x/\partial w$  and  $\partial J_x/\partial b$  for a single training example. Afterwards it is possible to recover  $\partial J_x/\partial w$  and  $\partial J_x/\partial b$  of an entire epoch by averaging over training examples (batch training, Section 2.5.3). By the second assumption, the cost shall be formulated as a function of the network output  $y(L)$ ,  $L$  being the (last) output layer. A typical function that is adopted for this role is the quadratic cost function:

$$J = \frac{1}{2} \sum_i (l_i - y(L)_i)^2 \quad (2.5)$$

In general, the online algorithm is the following:

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. This means computing the partial derivatives  $\partial J_x/\partial w(l)_{ji}$  and  $\partial J_x/\partial b(l)_j$ . But to compute those, an intermediate quantity  $\delta(l)_j$  is introduced. This corresponds to the “error” in the  $i$ -th neuron in the  $L$ -th layer. Backpropagation gives the procedure to compute the error  $\delta_j^{err}$  by means of the chain rule:

$$\frac{\partial J(w_{ji})}{\partial w_{ji}} = \frac{\partial J(w_{ji})}{\partial y(l)_{ji}} \frac{\partial y(l)_{ji}}{\partial x(l)_{ji}} \frac{\partial x(l)_{ji}}{\partial w_{ji}} \quad (2.6)$$

**Algorithm 1** Backpropagation.

```

function Initialize network weights( $W_L, b_L$ )
//typically small random values
for each training example  $x$  do
    //forward pass
    prediction = function Neural net output(network,  $x$ )
    for each output neuron  $t$  do
        //compute  $\delta(l)_L$  on output neurons
        function Compute error(prediction $_t$ )
    end
    for for index  $m$  from 1 to  $L$  do
        for each hidden neuron in layer  $L - m$  do
            //backward pass to compute  $\delta(l)^{err}$ 
            function Compute error(prediction $_t, W_{L-m,L}, b_{L-m,L}$ )
        end
    end
end
function Update network weights( $W_L, b_L, \delta(l)_L$ )

```

---

where:

$$\frac{\partial J(w_{ji})}{\partial y(l)_{ji}} \frac{\partial y(l)_{ji}}{\partial x(l)_{ji}} = \delta(l)_i = \frac{\partial J(w_{ji})}{\partial y(l)_{ji}} f'(x(l)_j) \quad (2.7)$$

and:

$$\frac{\partial x(l)_{ji}}{\partial w_{ji}} = y(l)_{ji} \quad (2.8)$$

which can be easily computed at the output layer. For the inner layers the deltas are propagated “backwards”:

$$\delta_j = \sum_i w_{ji} \delta(l)_i f'(x(l)_j) \quad (2.9)$$

And for the bias:

$$\frac{\partial J(b_i)}{\partial b_i} = \delta(l)_i \quad (2.10)$$

Without losing in generality, the previous values can be computed for any generic layer  $l$  during backpropagation and for this reason it is now referred to any generic weight for any layer  $l$ . The new values for  $w(l)_{ji}$  and the bias (after the backpropagation for sample  $x$ ) can be computed using the Stochastic Gradient Descent method (SGD) to yield:

$$w(l)_{ji}^x = w(l)_{ji}^{x-1} - \alpha \frac{\partial J(w(l)_{ji}^{x-1})}{\partial w(l)_{ji}^{x-1}} = w(l)_{ji}^{x-1} - \alpha \delta(l)_j y(l)_{ji} \quad (2.11)$$

$$b(l)_j^x = b(l)_j^{x-1} - \alpha \delta(l)_j \quad (2.12)$$

### 2.4.2 Initialization of the weights and Glorot Initialization

The most common ways to initialize the weights is by sampling randomly the values from a uniform or a normal distribution. This serves for the purpose of symmetry breaking. For the bias, it can be simply initialized to zero. Another popular initialization is the Glorot initialization, also known as Xavier initialization [30]. It is intended to help the signals reach deep into the network. In fact, if the weights in a network start too small, then the signal shrinks as it passes through each layer until it is too tiny to be useful. If the weights in a network are too large when training begins, then the signal grows as it passes through each layer until it is too massive to be useful. Xavier initialization keeps the signal in a reasonable range of values through many layers. In [30] the intuition is that preserving the variance of the backpropagated gradients throughout the network prevents the gradients from fading. Assuming to have an input  $X$  with  $n$  components and a linear neuron with random weights  $W$  that spits out a number  $Y$ ,  $Y$  is:

$$Y = W_1X_1 + W_2X_2 + \dots + W_nX_n \quad (2.13)$$

Assuming  $W_i$  and  $X_i$  to be independent and identically distributed, the variance of  $Y$  is:

$$\text{Var}(Y) = \text{Var}(W_1X_1 + W_2X_2 + \dots + W_nX_n) = n\text{Var}(W_i)\text{Var}(X_i) \quad (2.14)$$

the variance of the output is the variance of the input, but scaled by  $n\text{Var}(W_i)$ . So to have the variance of the input and output to be the same, that means  $n\text{Var}(W_i)$  should be 1. This means the variance of the weights should be

$$\text{Var}(W_i) = 1/n = 1/n_{in} \quad (2.15)$$

By going through the same steps for the backpropagated signal, from [30] it is possible to elaborate that to keep the variance of the input gradient and the output gradient the same:

$$\text{Var}(W_i) = 1/n = 1/n_{out} \quad (2.16)$$

The two constraints can only be satisfied simultaneously if  $n_{in} = n_{out}$  but given that this is most likely not always the case, it is recommended to use an average of the two as a compromise [30]:

$$\text{Var}(W_i) = 2/(n_{in} + n_{out}) \quad (2.17)$$

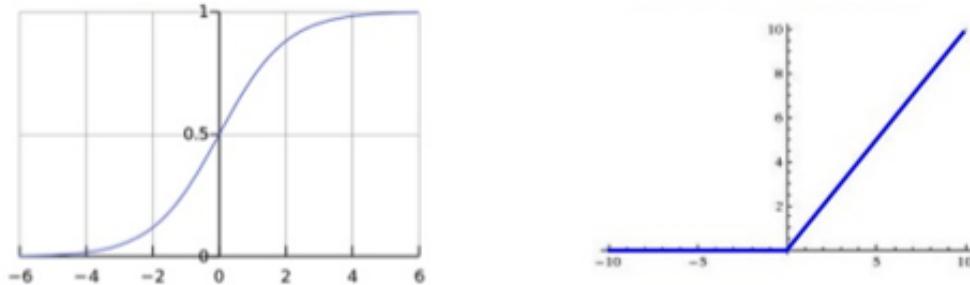
For this reason Xavier initialization draws values from a distribution with zero mean and a specific variance.

### 2.4.3 Sigmoid and rectified linear activation function

Non-linear activation functions introduce non-linearity in the network. Without a non-linear activation function, the network can only learn functions which are linear combinations of its inputs. Using the sigmoid function in the output layer as activation function is very common practice. Its expression is:

$$f(x) = \frac{1}{(1 + e^{-x})} \quad (2.18)$$

The reason why it is a good practice to use a sigmoid as opposed to something else is that it is continuous and differentiable and its derivative is very fast to compute (as opposed to the derivative of tanh, which has similar properties). It also maintains a limited range (from 0 to 1, exclusive).



**Figure 2.8:** On the left: sigmoid activation function. On the right: rectified linear function.

The rectified linear function is frequently used for hidden layers [18]. It is a simple non-linearity: It evaluates to 0 for negative inputs, and positive values remain untouched ( $f(x) = \max(0, x)$ ). The gradient of the rectified linear function is 1 for all positive values and 0 for negative values. This means that during backpropagation, negative gradients will not be used to update the weights of the outgoing rectified linear unit. However, because the network has a gradient of 1 for any positive value it has much better training speed when compared to other non-linear functions due to the good gradient flow. According to [18], deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units. For example, the logistic sigmoid function has very tiny gradients for large positive and negative values so that learning nearly stops in these regions (this behavior is similar to a saddle point). The ReLUs also have the desirable property that they do not require input normalization to prevent them from saturating [18]. In fact, despite the fact that negative gradients do not propagate with rectified linear functions (the gradient is zero here), large gradients for positive values are very powerful and ensure fast training regardless of the size of the gradient.

#### 2.4.4 Cross-entropy cost function

Considering that a neuron learns by changing the weight and bias at a rate determined by the partial derivatives of the cost function, the expressions  $\partial C/\partial w$  and  $\partial C/\partial b$  suggest that a "slow learning" is quite the same as saying that those partial derivatives are small. From the graph of image 2.1 it is possible to see that when the neuron's output is close to 1, the curve gets very flat, and so ( $z$ ) gets very small. It also implies that  $\partial C/\partial w$  and  $\partial C/\partial b$  get very small. This is cause to a learning slowdown. It turns out that the issue can be solved by replacing the quadratic cost with a different cost function, known as the cross-entropy, whose expression is:

$$C = -\frac{1}{n} \sum_x l_j^x \ln(y_j^x) + (1 - l_j^x) \ln(1 - y_j^x) \quad (2.19)$$

where  $n$  is the total number of items of training data, the sum is over all training inputs,  $x$ , and  $l$  is the corresponding desired output label. Two properties in particular make it reasonable to interpret the cross-entropy as a cost function:

1. first, it is non-negative. That is:  $J > 0$  because all the individual terms in the sum are negative and logarithms are of numbers in the range 0 to 1;
2. second, if the neuron's actual output is close to the desired output for all training inputs  $x$ , then the cross-entropy will be close to zero.

To see this, suppose for example that  $l = 0$  and  $y \approx 0$  for some input  $x$ . This is a case when the neuron is doing a good job on that input. It is easy to see that the first term in the expression for the cost vanishes, since  $l = 0$ , while the second term is also approximately 0. A similar analysis holds when  $l = 1$  and  $y \approx 1$ . Hence the contribution to the cost will be low provided the actual output is close to the desired output. On top of that, the cross-entropy cost function has the benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down. It provides that the larger the error, the faster the neuron will learn. To show that, it just takes to substitute  $(z)$  to the  $y$  in the cross-entropy expression and compute its partial derivative:

$$\frac{\partial J(w)}{\partial w(l)_j} = -\frac{1}{n} \sum_x \left( \frac{l}{\sigma(z)} - \frac{1-l}{1-\sigma(z)} \right) \frac{\partial \sigma(z)}{\partial w(l)_j} = -\frac{1}{n} \sum_x \left( \frac{l}{\sigma(z)} - \frac{1-l}{1-\sigma(z)} \right) \sigma'(z) \quad (2.20)$$

$$\frac{\partial J(w)}{\partial w(l)_j} = \frac{1}{n} \sum_x \left( \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma'(z))} \right) (\sigma(z) - l) \quad (2.21)$$

Using the definition of the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.22)$$

it is possible to show that:

$$\sigma'(z) = \sigma(z)(1 - \sigma'(z)) \quad (2.23)$$

It is now easy to see that the  $\sigma'(z)$  and  $\sigma(z)(1 - \sigma'(z))$  terms cancel out in the equation 2.16, and it simplifies to become:

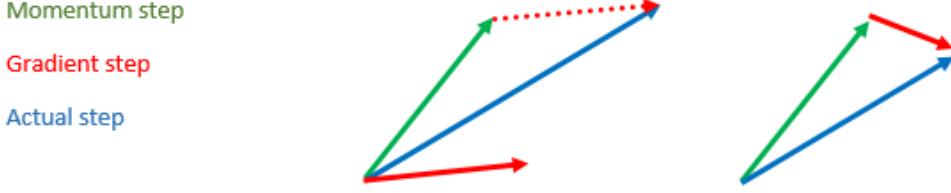
$$\frac{\partial J(w)}{\partial w(l)_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - l) \quad (2.24)$$

This expression tells that the rate at which the weight learns is controlled by  $\sigma(z)l$ , i.e., by the error in the output. Hence, as stated before, the larger the error, the faster the neuron will learn. When the cross-entropy is used, the  $\omega'(z)$  term gets canceled out, and it is no longer needed to worry about it being small.

#### 2.4.5 Update criterions

**Nesterov Momentum:** Nesterov momentum is a slightly different version of the momentum update (used in Nesterov accelerated gradient - NAG) that grants stronger theoretical converge for convex functions and in practice works slightly better than standard momentum. The Nesterov momentum can be used in combination

with the backpropagation algorithm to provide the new values to the weights of the network.



**Figure 2.9:** On the left: classical momentum update. On the right: Nesterov momentum update.

The difference between classical momentum and the Nesterov version is that while in the first one the velocity is updated first and afterwards a big step according to that velocity is made, in Nesterov momentum first a step into the velocity direction is made, followed by a correction to a velocity vector based on new location. The utility of Nesterov momentum is discussed in [31]. While NAG is not typically thought of as a type of momentum, it indeed turns out to be closely related to classical momentum and differs only in the update of the velocity vector  $v$ . As a comparison, Nesterov momentum update is compared to the equations for classical momentum that follow:

$$V(t+1) = m * V(t) - \alpha * \text{grad}(W(t)) \quad (2.25)$$

$$W(t+1) = W(t) + V(t+1) \quad (2.26)$$

Being  $\alpha$  the learning rate. Nesterov momentum update equations:

$$V(t+1) = m * V(t) - \alpha * \text{grad}(W(t) + m * V(t)) \quad (2.27)$$

$$W(t+1) = W(t) + V(t+1) \quad (2.28)$$

Where  $m$  is the momentum. An alternative implementation that is quite similar to the original is presented. It is in the same form as regular momentum in the sense that both velocity and parameter updates depend only on the gradient at the current value of the parameters. The equations for the alternate formulation for Nesterov momentum are:

$$V(t) = \mu * V(t) - \alpha * \text{grad}(W(t)) \quad (2.29)$$

$$W(t+1) = W(t) + m^2 * V(t) - \alpha(1 + m * V(t)) * \text{grad}(W(t)) \quad (2.30)$$

In this case  $W(t)$  represents the new value of the weight from 2.5. Using the notations from Section 2.4:

$$v(l)_{ij}^{x+1} = \mu * v(l)_{ij}^x - \alpha * \delta(l)_j y(l)_{ji} \quad (2.31)$$

$$w(l)_{ij}^{x+1} = w(l)_{ij}^x + m^2 * v(l)_{ij}^{x+1} - \alpha(1 + m * v(l)_{ij}^x) * \delta(l)_j y(l)_{ji} \quad (2.32)$$

**Adaptive gradient (AdaGrad):** AdaGrad is an optimization method that allows different step sizes for different features. It increases the influence of rare but informative features. This method modifies the general learning rate at each iteration for every parameter  $w(l)_{ij}^x$  based on the past gradients that have been computed for  $w(l)_{ij}^x$ :

$$v(l)_{ji}^{x+1} = v(l)_{ji}^x + (\delta(l)_j y(l)_{ji})^2 \quad (2.33)$$

$$w(l)_{ij}^{x+1} = w(l)_{ij}^x - \frac{\eta}{\sqrt{v(l)_{ij}^{x+1} + e}} * \delta(l)_j y(l)_{ji} \quad (2.34)$$

Where  $e$  is a smoothing term that avoids division by zero,  $\eta$  is the learning rate. One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

**Adaptive delta (AdaDelta):** Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size  $d$ . Instead of inefficiently storing  $d$  previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. Now the running average  $v(l)_{ij}^{x+1}$  corresponding to the iteration of sample  $x+1$  depends only on the previous average and the current gradient.

$$v(l)_{ij}^{x+1} = \rho * v(l)_{ij}^x + (1 - \rho) * (\delta(l)_j y(l)_{ji})^2 \quad (2.35)$$

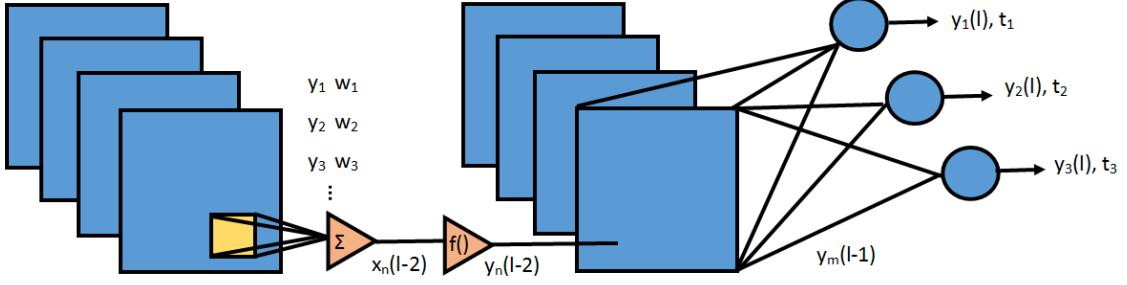
## 2.5 Training a Convolutional Network

The backpropagation algorithm formulation can be adapted to convolutional and pooling layers in CNNs. In this case the different sets of connections must be taken into account. For last, the concepts of dropout and batch training as opposed to online training are considered.

### 2.5.1 Backpropagation algorithm for CNNs

The backpropagation in convolutional neural networks is similar to the one used in fully connected neural networks. The only difference is that the partial derivative of the loss with respect to a weight element is the sum of chain-rule expressions for all the neurons that are affected by that weight. This is because of the weight sharing. The error ( $C_{em}$ ) will be used to compute for the previous layers the partial derivative of the cost function with respect to each neuron output by applying the chain rule:

$$\frac{\partial J(w)}{\partial w(l)_{ab}} = \sum_{i=0}^{p-q} \sum_{j=0}^{p-q} \frac{\partial J(w)}{\partial x(l)_{ji}} \frac{\partial x(l)_{ji}}{\partial w(l)_{ab}} = \sum_{i=0}^{p-q} \sum_{j=0}^{p-q} \frac{\partial J(w)}{\partial x(l)_{ji}} y(l)_{(i+a)(i+b)} \quad (2.36)$$



**Figure 2.10:** Two simplified convolutional layers followed by a fully connected layer. In yellow: the kernel made up by a set of weights  $(w_1, \dots, w_p)$ .  $f()$  is the activation function (not shown for output  $y_n(l-1)$ ). The outputs  $y_n(l-2)$  are the features for the corresponding layer  $l-2$ .

$P$  is the input data size (typically a  $P \times P$  square image) and  $q$  the filter size;  $a$  and  $b$  are the coordinates of a specific coefficient in the filter. The sum is computed over all  $x(l)_{ji}$  expressions from which  $w_{ab}$  is originated (this is the weight-sharing in the neural network!).  $y(l-1)_{(i+a),(j+b)}$  is already known from the forward propagation equations.

$$\frac{\partial J(w)}{\partial x(l)_{ji}} = \frac{\partial J(w)}{\partial y(l)_{ji}} \frac{\partial y(l)_{ji}}{\partial x(l)_{ji}} = \frac{\partial J(w)}{\partial y(l)_{ji}} \frac{\partial}{\partial x(l)_{ji}}(f(x(l)_{ji})) \quad (2.37)$$

Being  $f$  the activation function. The activation function in this case corresponds to the non-linearity ReLu that is applied to each convolutional layer. Its derivative is the unit constant and can be skipped in this case. Since the errors at the current layer are known, everything that is needed to compute the gradient with respect to the weights used by the convolutional layers is provided. In addition to compute the weights for this convolutional layer, it is necessary to propagate the errors back to the previous layer. Again the chain rule can be used for this purpose:

$$\frac{\partial J(w)}{\partial y(l)_{ji}} = \sum_{i=0}^{p-q} \sum_{j=0}^{p-q} \frac{\partial J(w)}{\partial x(l)_{ji}} \frac{\partial x(l)_{(i-a)(j-b)}}{\partial y(l-1)_{ji}} = \sum_{i=0}^{p-q} \sum_{j=0}^{p-q} \frac{\partial J(w)}{\partial x(l)_{ji}} w_{ab} \quad (2.38)$$

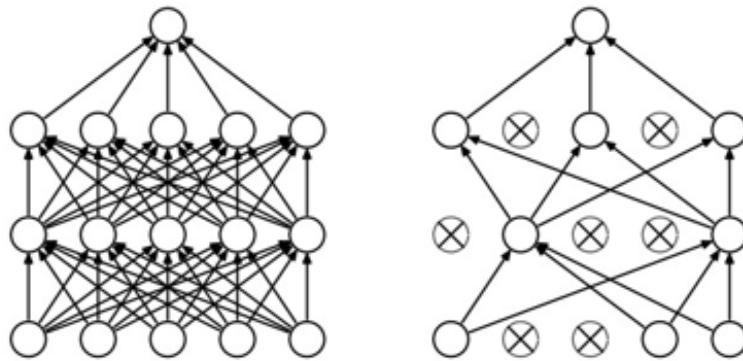
The new value for each weight is given by the same equation as :

$$w(l)_{kj} = w(l)_{kj} - \alpha \frac{\partial J(w_{ji})}{\partial w(l)_{ji}} = w(l)_{kj} - \alpha \delta(l)_k^{err} = w(l)_{kj} - \alpha \delta(l) y(l)_{kj} \quad (2.39)$$

On top of that the new values for the weights  $w(l)_{kj}$  may be optionally fed to the equations to compute the Nesterov momentum (2.29, 2.30), AdaGrad (2.33, 2.34) or Adadelta (2.35): Regarding the max-pooling layers, these do not actually do any learning themselves. Instead, then reduce the size of the problem by introducing sparseness. In forward propagation,  $k \times k$  blocks are reduced to a single value. Then, this single value acquires an error computed by backwards propagation from the previous layer. This error is then just forwarded to the place (the specific coefficient) where it came from. Since it only came from one place in the  $k \times k$  block, the back-propagated errors from max-pooling layers are rather sparse. Once all minibatches have been exhausted on the last iteration, the training process is completed.

### 2.5.2 Using dropout

Since the fully connected layers include most of the parameters of the network, it is prone to overfitting. The dropout method is introduced to prevent overfitting. At each training stage, individual nodes are either "dropped out" of the net with a given probability "p" or kept with probability  $1-p$ .



**Figure 2.11:** On the left: standard neural network; on the right same neural network after applying dropout. Image from [26].

Only the "reduced" network is trained on the data in that specific stage. The removed nodes are then reinserted into the network with their original weights on the following stage. References on the improvements of generalization brought by dropout can be found in [25] and [26]. As explained in [25], dropout prevents co-adaptation of feature detectors for which a feature is only helpful in the presence of other specific features. The training should instead learn to provide the correct features in any combinatorial set of detected features. In [25] 50% dropout was applied to all hidden layers and 20% dropout to input and output layers in different architectures for MNIST classification to cut the number of errors in the test dataset by about 30 (originally 160 samples were misclassified). [25] shows improved performances on a variety of datasets: for the MNIST dataset the error rate is reduced by about 0.25% (from an original 1.60% error rate).

### 2.5.3 Online and batch training

Neural networks are often trained using algorithms that approximate gradient descent. Gradient descent learning (also called "steepest descent") can be done using either a batch method or an on-line method. In batch training, weight changes are accumulated over an entire presentation of the training data (an epoch) before being averaged and applied, while on-line training updates weights after the presentation of each training element (instance). Another alternative is sometimes called mini-batch, in which weight changes are accumulated over some number  $u$  of instances before actually updating the weights. Using an update frequency (or batch size) of " $u$ " equals to 1 results in on-line training, while an  $u$  equals to  $N$  results in batch training, where  $N$  is the number of instances in the training set.

Random sequence, length = 10									
Batch 1					Batch 2			Leftovers	
IM_2, cl. 2	IM_6, cl. 25	IM_10, cl. 4	IM_60, cl. 30	IM_43, cl. 32	IM_70, cl. 5	IM_234, cl. 6	IM_678, cl. 9	IM_349, cl. 2	IM_56, cl. 18
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

**Figure 2.12:** Batch training in a simplified scenario. When the number of processed training items stored in an “Accumulation” reaches the batch size, all weights and biases are updated using the accumulated gradients, and the Accumulation is reset to 0.

### 2.5.4 Using normalization

The addition of normalization to CNNs is beneficial to training convergence. Normalization layers perform pixelwise normalization as follows:

$$x_j = \frac{x_i}{k + (\alpha \sum_j x_j^2)^\beta} \quad (2.40)$$

In other words, given the feature “k”, the normalization for k is computed including the values from the n neighbours (from  $k-n/2$  to  $k+n/2$ ). The ordering of the kernel maps (the channels) is arbitrary and determined before training begins. The reason to use NORM layers is due to the concept of “lateral inhibition”. This concept refers to the capacity of an excited neuron to subdue its neighbors. The desire is to have a significant peak in order to obtain a form of local maxima. Cross-channel normalization serves this purpose as it tends to amplify a prominent feature while dampening the surrounding ones. Response normalization reduces error rates up to 1.4% on [18] on the CIFAR-10 dataset.

## 2.6 Growcut segmentation preprocessing

To improve the traffic sign classification accuracy this simple growcut segmentation is tested. The purpose is to crop the pixels belonging to the actual sign out of the background pixels (sky, vegetation or other misleading details). The algorithm is very simple and can be thought of as using image pixels as “cells” of a certain type. These cells can be foreground or background. As the algorithm proceeds, these cells compete to dominate over the image domain. The ability of the cells to spread is related to the image pixel’s attack strength. The code that was used is an implementation that follows [28] and summarized in algorithm 2.

The attack force function that was used is:

$$g(x) = 1 - \frac{x}{\max\|C_q - C_p\|^2} \quad (2.41)$$

being C the feature vector of the pixel. Index p lists all pixels in the image on the for loops and q indexes all pixels belonging to a pixel p’s neighborhood. Neighborhood of choice is the Moore neighborhood, consisting of pixels such that:

$$N(p) = q \in Z^n : \|p - q\|_\infty := \max\|p_i - q_i\|, i \in [1, n] \quad (2.42)$$

---

**Algorithm 2** Growcut segmentation.

---

```
Ii=functionResizeto[64, 64](Ii)
//labels(32,32) = 1;
//labels(1,1), (1,64), (64,64), (64,1) ,(1,32),(64,32),(32,64), (32,1) = -1;
while below n. iterations do
    for each label in image Ii do
        //copy previous state
        labels-new = labels;
        strength-new = strength;
        // all neighbors q of p attack
        for all neighbors q do
            if attack-force*strength(q)>strength-new(p) then
                labels-new(p) = labels(q)
                strength(p) = strength-new(q)
            end
        end
    end
    binary-maski = labels
    //binary mask from region grown from labels(32,32);
    function Median filter(binary-maski)
    function Dilate(binary-maski)
    Si = binary-maski * Ii
    return segmented image Si
end
```

---

# 3

## Work performed in this thesis

In this thesis we use deep learning for Traffic Sign Classification from images. The standard German Traffic Sign Benchmark Dataset (GTSRB) is used for training and testing. Given the objective of this thesis, we focus entirely on the Classification of the GTSRB dataset for the sake of obtaining high accuracy and comparing different preprocessing techniques, network architectures and parameters. We then compare the performance by applying CNNs on raw GTSRB images and on enhanced images. For the first activity the tasks that were carried out in this project are the following:

1. Implementation of a basic CNN architecture.
2. Training CNN classifier.
3. Tests on the separate dataset using trained CNNs.
4. Preprocessing the training and testing data.
5. Tests on different settings and their impact on the performance.
6. Collecting the results and drawing conclusions.

**Deep learning using CNNs with Theano and Lasagne** The first important decisions that were made are the basic type of architecture to implement and the software library to be used for the implementation.

The models that were used are CNNs.

In this thesis the number of layers is limited due to hardware resources and relatively simple problems. We chose to use the Linux installation of the Theano and Lasagne libraries.

### 3.0.1 Hyperparameters

Training a CNN takes several hours or even days depending on the complexity of the network and the data. Designing an architecture to provide good accuracy while keeping the training time bounded requires to tune the multiple parameters involved one by one while constantly checking training and testing accuracy. The set of relevant hyperparameters that are initially considered and progressively tuned-up during this project are the following.

Related to the training images:

- image size,
- preprocessing parameters,
- size of the dataset.

Related to the training:

- number of epochs,
- learning rate,

- weight update criterion,
- training loss expression,
- weight initialization criterion,
- batch size.

Related to the architecture:

- number of layers,
- type of layers,
- number of filters,
- non-linearities,
- strides,
- dropout,
- zero padding.

**Initial setup and tests:** to begin with, we do not do any particular preprocessing on the training data except for a simple rescaling (on rectangular images) and resizing (bilinear interpolation method) to 32x32 pixels. Although it is possible to train with data of different sizes, this makes it more complicated to control the features size throughout the network and make the training convergent. The new model can be seen in Figure 3.1. We provide the following considerations about the layers we used:

- The input in this network is defined a priori. This means that this model cannot input images of shapes that are different from the 32x32 rgb setting. The batch size though is flexible so that the last batch can have less elements if needed.
- Convolutional 2D layers (CONV), are the layers that perform the convolution with a given set of filters. We set them in order to preserve the input feature size. Each one of the CONV layers we use is followed by a ReLu nonlinearity.
- Maximum pooling 2D layers (MAXPOOL), that allow to subsample the input data. These are actually the only layers that we use to reduce the input feature size. Halving the data at layer 4 and 7 is essential for convergence and to contain the training time.
- Normalization layers (NORM). The normalization we use here follows Section 2.5.4. where the summation is computed over this position up to 5 neighboring channels.
- Dropout layers (DOL), that iteratively inhibit random weights. Dropout is only used during training (not testing) and when it is used we rescale the input as:

$$x_{out,l} = \frac{x_{in,l}}{1-p} \quad (3.1)$$

where p is the probability of dropout and it is equal to 0.5.

- Fully connected layers (FCC) with all-to-all type of connections. These layers are placed at the end of the network. The second FCC closes the network and contains a number of neurons equal to the number of classes in the training data. In this case they are 43.

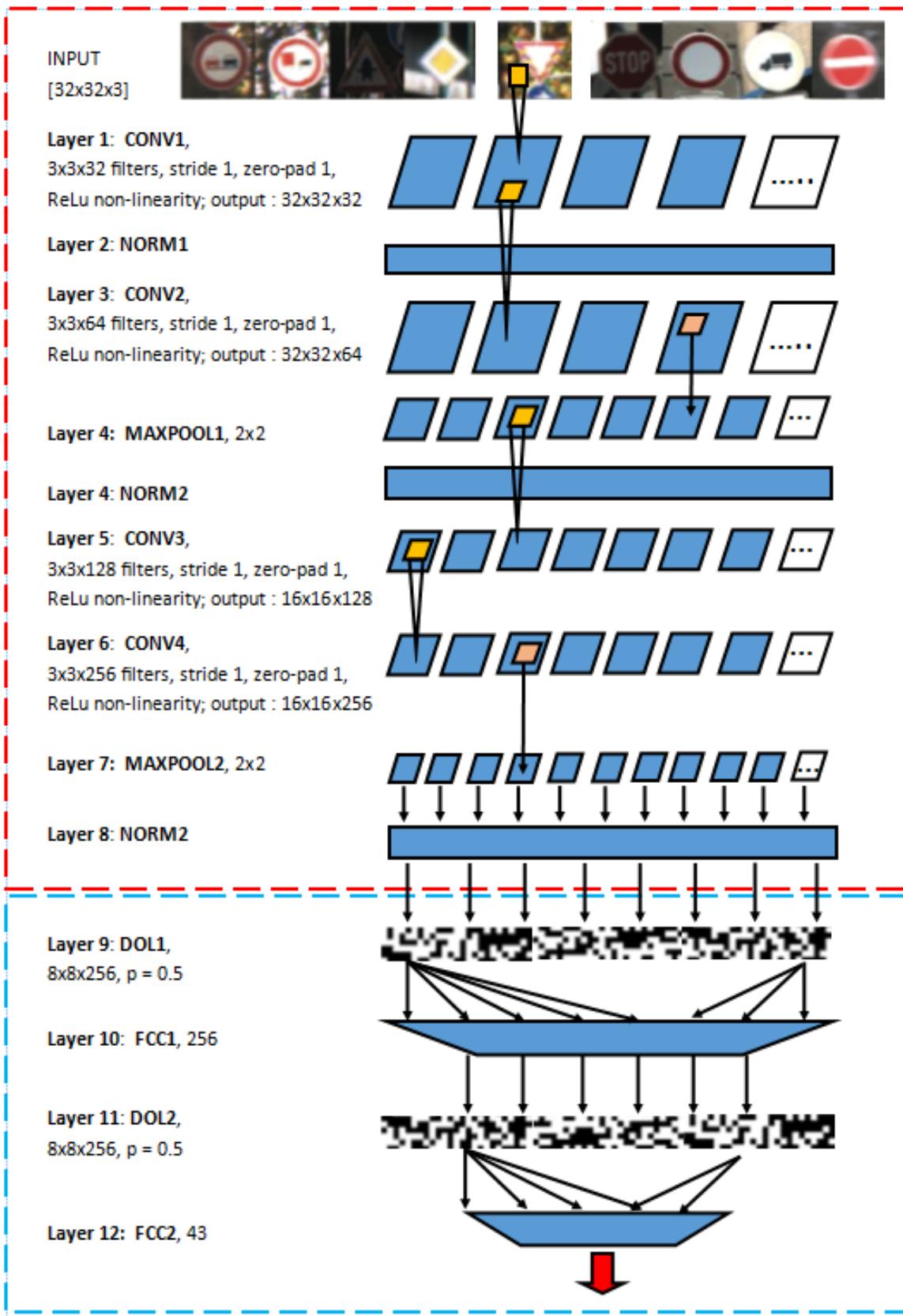


Figure 3.1: Convolutional architecture for GTSRB (CNN for GTSRB).

### 3.0.2 Training the model

To train the model we use the Nesterov momentum (paragraph 2.4.5 set to 0.9) for the updates, categorical crossentropy (section 2.4.4) for the training loss and a batch training by setting the batch size to 128 (section 2.5.3). The weights are initialized using Glorot initialization method (section 2.4.2). In this work we also do not use any pre-training. Every time the network is trained all weights are reinitialized. To summarize:

Parameter type	
Weight init.:	Glorot initialization method
Training loss:	Categorical crossentropy
Updates:	SGD and Nesterov momentum
Momentum	0.9
Batch size:	128
Dropout:	50%

Given this relatively small number of images in GTSRB, 214 iterations with batch size 128 are needed to cover one full epoch on the entire training dataset. We train the network in each study case until convergence and stop when overfitting occurs.

**Learning rate** The most significant parameter at this point is the learning rate. We notice that if  $\alpha$  is just “too high” at values close to 0.02 the result is bad. In fact above a certain value of  $\alpha$  the network does not converge at all and the training loss explodes. This is a hint to try a lower learning rate. When the learning rate  $\alpha$  is around 0.01, we typically observe constant oscillation in the error rate but this is a good starting value to begin with. The larger a learning rate, the easier is to miss the minimum in the error by "jumping" too far but we tackle this issue by adjusting the learning rate after a few iterations.

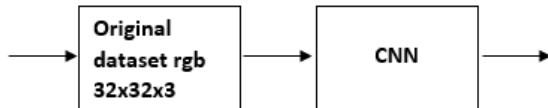
## 3.1 Case studies

We take into consideration three specific study cases to train the proposed network and classify the test data. Different preprocessing is applied to each one of the cases while using the same network across the different cases. The training parameters are those described in Section 3.0.2 and are shared between the cases. All the listed preprocessing normalizations need to be applied on both the training and testing data. On the fifth case study we change instead the number and the size of the filters as we are interested in studying the effectiveness of our supervised training on shaping the filters to extract the best features for TSR. The dataset is divided between 2/3 of the data for training and 1/3 for testing. We use the same subsets across the study cases.

### 3.1.1 Case study 1 (CS1): raw images

The standard normalizations are the resizing with bilinear interpolation and the image adjustment (rescale\_intensity, exposure module, scikit-image) to increase image contrast by mapping pixel intensities to new values such the minimum and maximum

intensities of the input image are stretched to the limits. This allows the darkest and brightest pixels of the image to get saturated at low and high intensities.



**Figure 3.2:** Block diagram for CS1.

### 3.1.2 Case study 2 (CS2): clipped images

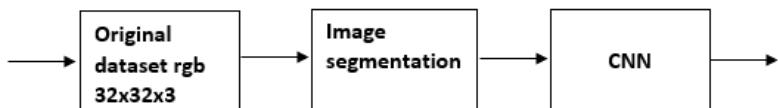
Following raw image training we try to increase the performances with cropped data. The data is first resized to 64x64. Central cropping is then used to clip the 32x32 image patch centered on the original image's center. Even though a few images may not be perfectly centered on the traffic sign (offsets of no more than a few pixels [36]), the approximation still allows to cut most of the background while preserving the digits or the stylized drawing on the sign. The information of the actual shape of the board on which the sign is printed is generally lost.



**Figure 3.3:** Block diagram for CS2.

### 3.1.3 Case study 3 (CS3): segmentation

In the segmented GTSRB dataset the actual symbol on the sign is very visible in most cases while the shape of the sign is frequently lost in the segmentation. Hence the tradeoff is removing the background while losing the shape of the sign. The segmentation we use is explained in Section 2.6.



**Figure 3.4:** Block diagram for CS3.



**Figure 3.5:** Examples of segmented images from GTSRB. From left to right and top to bottom: samples from classes “0”, “3”, “16”; “35”, “42”, “17”; “19”, “15”, “22”.

# 4

## Results

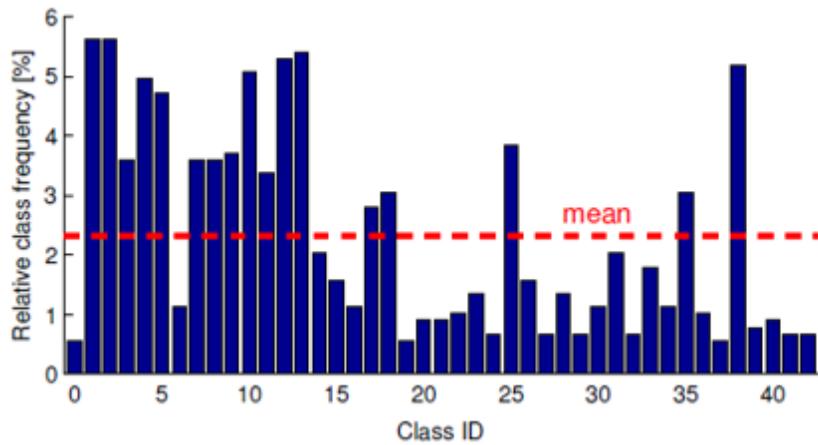
### 4.1 Dataset description



**Figure 4.1:** All 43 benchmark classes in GTSRB.

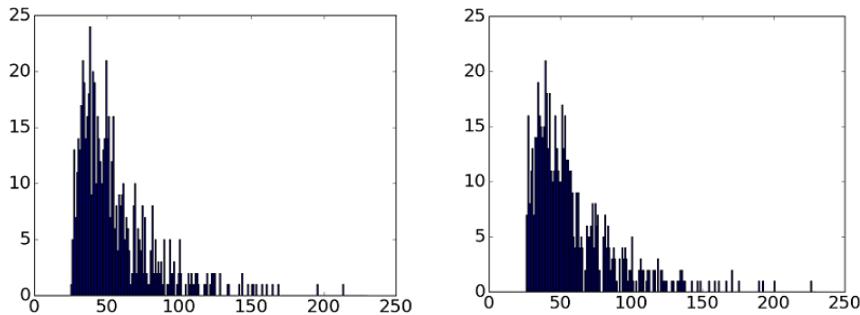
The German Traffic Sign Recognition Benchmark (GTSRB) [23], [36] was the object of analysis at the classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011. This dataset satisfies two important requirements.

1. it is sufficiently big for training a CNN,
2. it is a good representation of its own classes.



**Figure 4.2:** Relative class frequencies in the dataset. The class ID results from enumeration of the classes in Fig. 3 from top-left to bottom-right. Image from [36].

In fact, this dataset includes a total of 43 classes (Figure ??) and more than 50000 images total. Traffic sign recognition (TSR) is typically a multi-class classification challenge which requires to cope with unbalanced class frequencies. For instance, it is easy to realize that a 50 km/h speed limit is more frequent than a 120 km/h one. Also, a TSR dataset should account for situations of different illumination changes, partial occlusions, rotations, weather conditions etc. It consists of highly uneven classes in terms of number of samples for each class. The images reflect the strong variations in visual appearance of signs due to distance, illumination, weather, partial occlusions, and rotations [36]. Images are stored in PPM format (Portable Pixmap, P6) in separate locations for each class. On Figure 4.3 it is possible to see the histograms of the heights and widths of all GTSRB images.



**Figure 4.3:** On the left: histogram of heights of GTSRB. On the right: histogram of widths of GTSRB.

Their sizes vary between 15x15 to 250x250 pixels (not necessarily squared). The actual traffic sign is also not necessarily centered within the image. Averaging the size of the dataset (computed over shapes of the images) forces the resizing method to make use of different scaling factors. Given that training and testing images are 32x32 their size is roughly halved on average, although some of the smallest are almost unchanged. Most images contain a border of about 10% around the actual traffic sign (5 pixels or more on average) that allow for edge-based segmentation

approaches. The samples of each class have been randomly split into training and testing data, to mantain a proportion of roughly 70% of images as training data and 30% as testing data. The exact number of samples in each category is reported in table 4.1.

## 4.2 Experimental setup

For the tests we use a CPU Intel Core2 Duo T6500 at 2.10GHz with 2GB DDR3 RAM (used for 16x16 data) and a CPU Intel Core(TM)2 Duo E8400 at 3.0GHz with 4GB DDR3 RAM (32x32 data). It is no secret that this type of hardware does not guarantee high processing speed. We must stress the fact that the purpose of this thesis is in fact not to reach top classification performances and neither to run the training for a very high number of epochs. We focus on comparing performances of different study cases with the same network model and training parameters. We are satisfied with the results once the training is converged. Then we compare the study cases and draw conclusions.

**Dataset organization.** As mentioned in Section 4.1 the GTSRB is composed of highly unbalanced data. Some of the classes only have 200 images, which is quite a limitation for deep learning. We use the same subset of images for validation and testing (there is no update of hyperparameters during the training to justify validation-only data). We still keep track of both training and validation accuracy to avoid overfitting. The data is sorted out randomly as shown in Figure 4.4: roughly one third of images are used for training and two thirds for testing.

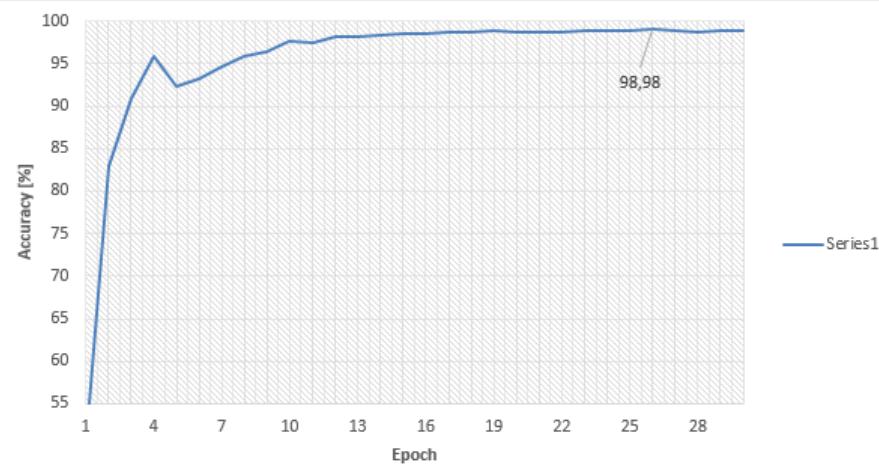
Class num.	Class name	Traffic sign	IJCNN 2011 class group	Total available samples
00	20 Speed limit		red-round, speed	210
01	30 Speed limit		red-round, speed	2220
02	50 Speed limit		red-round, speed	2250
03	60 Speed limit		red-round, speed	1410
04	70 Speed limit		red-round, speed	1980
05	80 Speed limit		red-round, speed	1860
06	End of 80 speed limit		end-of	420
07	100 Speed limit		red-round, speed	1440
08	120 Speed limit		red-round, speed	1410
09	No overtaking		red-round, red-other	1470
10	No overtaking by heavy vehicles		red-round, red-other	2010
11	Crossroads		danger	1320
12	Priority road		spezial	2500
13	Give way		spezial	2510
14	Stop and give way		spezial	780
15	Restricted vehicular access		red-round, red-other	630
16	No large heavy vehicles		red-round, red-other	420
17	No entry for vehicular traffic		spezial	1110
18	Other danger		danger	1200
19	Bend, to left		danger	210
20	Bend, to right		danger	360
21	Double bend, first to left		danger	330
22	Uneven road		danger	390
23	Slippery road		danger	510
24	Road narrows on right		danger	270
25	Road works		danger	1500
26	Traffic lights		danger	600
27	Pedestrians		danger	240
28	Children crossing		danger	540
29	Bicycles crossing		danger	270
30	Snow		danger	450
31	Animal crossing		danger	780
32	End of all speed limits		end-of	240
33	Turn right ahead		blue	659
34	Turn left ahead		blue	423
35	Ahead only		blue	1200
36	Go straight or right		blue	390
37	Go straight or left		blue	210
38	Keep right		blue	2070
39	Keep left		blue	300
40	Roundabout mandatory		blue	360
41	End of no overtaking		end-of	240
42	End of no overtaking by trucks		end-of	240

Figure 4.4: All 43 benchmark classes in GTSRB.

## 4.3 Experimental results

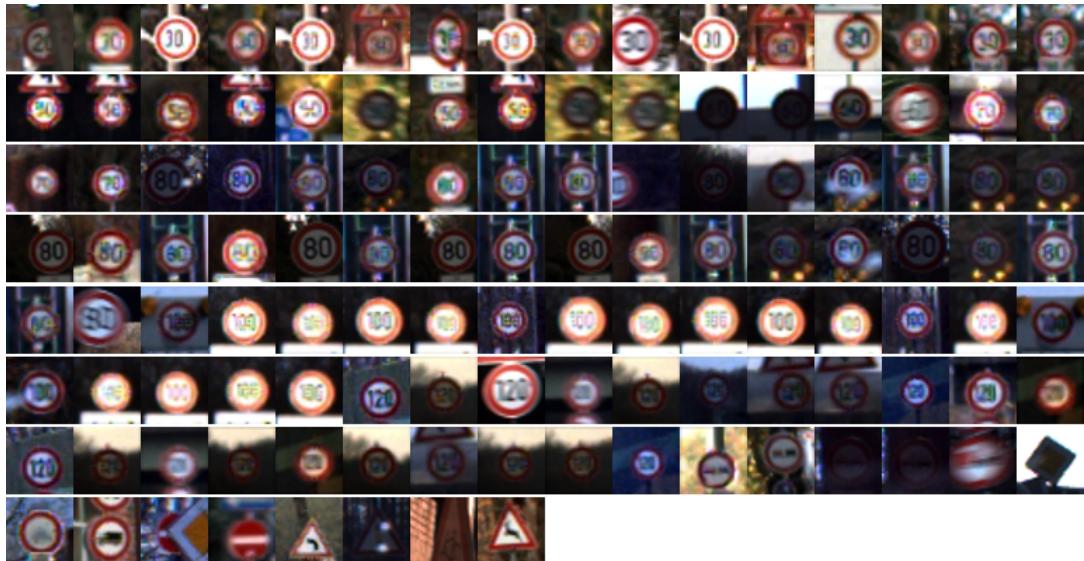
### Results, Case Study 1

Case Study 1 using the raw 32x32 rescaled images reaches 98.98% accuracy on the test data in about 30 iterations. Below in the table the classification rate and the false alarm rate obtained by averaging class rates. The confusion matrix is shown as well. The empty cells in the matrix are zeros.



**Figure 4.5:** Test accuracy of CS1 (raw images).

N. of signs	Avg. classification rate [%]	Avg. false alarm rate [%]
11763	99.40	0.63

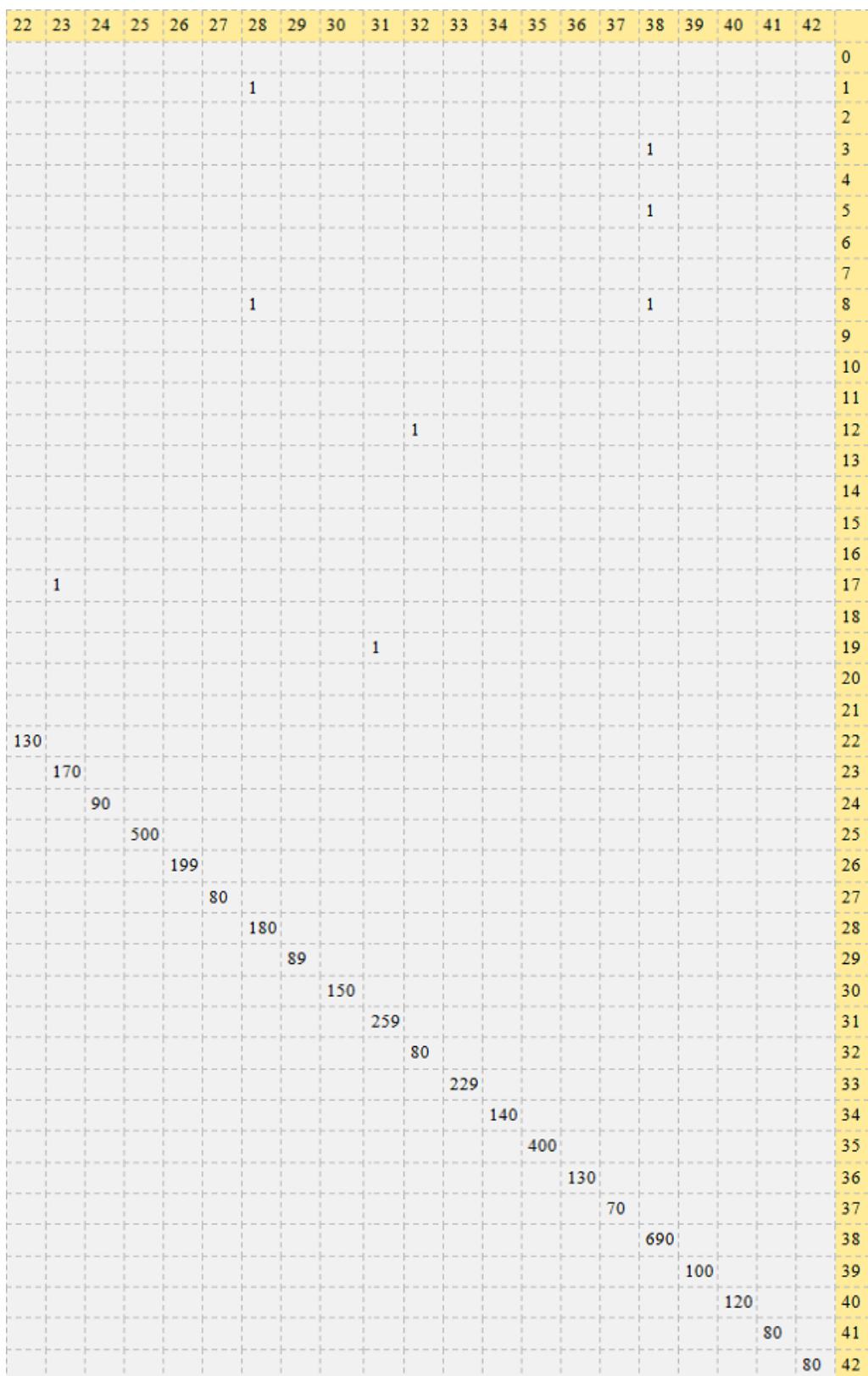


**Figure 4.6:** The classification errors in CS1.

Class					
Number	Test samples	False neg.	Classif. rate [%]	False pos.	F. a. rate [%]
0	70	1	98.57	0	0.00
1	740	15	97.97	19	2.53
2	750	10	98.66	20	1.33
3	470	4	99.14	6	0.85
4	660	4	99.39	5	0.61
5	620	32	94.83	21	5.16
6	140	0	100.00	0	0.00
7	480	15	96.87	16	3.12
8	470	21	95.53	8	4.47
9	490	4	99.18	6	0.82
10	670	1	99.85	0	0.15
11	440	0	100.00	1	0.00
12	700	1	99.86	1	0.14
13	720	0	100.00	2	1.39
14	260	1	99.61	0	0.00
15	210	1	99.52	1	0.48
16	140	0	100.00	1	0.71
17	370	2	99.46	1	0.27
18	400	0	100.0	0	0.00
19	70	1	98.57	1	1.43
20	120	0	100.00	0	0.0
21	110	0	100.00	0	0.00
22	130	0	100.00	0	0.00
23	170	0	100.00	1	0.59
24	90	0	100.0	0	0.00
25	500	0	100.00	0	0.00
26	200	1	99.50	0	0.00
27	80	0	100.00	0	0.00
28	180	0	100.00	2	1.11
29	90	1	98.89	0	0.00
30	150	0	100.00	0	0.00
31	260	1	99.61	1	0.38
32	80	0	100.00	1	1.25
33	229	0	100.00	0	0.00
34	140	0	100.00	0	0.00
35	400	0	100.00	0	0.00
36	130	0	100.00	0	0.00
37	70	0	100.00	0	0.00
38	690	0	100.00	3	0.43
39	100	0	100.00	0	0.00
40	120	0	100.00	0	0.00
41	80	0	100.00	0	0.00
42	80	0	100.00	0	0.00

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	69	1																				
1		725	8		3				1			1							1			
2		5	740	1	1	1			1	1												
3			466		3																	
4		1	1		656	1											1					
5		6	9	5	1	588		6	2	1								1				
6						140																
7				1		14		461	4													
8		5	1			2		9	449	2												
9									486	3						1						
10									1	669												
11										440												
12											699											
13											720											
14									1			259										
15									1			209										
16												140										
17											1		368									
18												400										
19												69										
20												120										
21													110									
22																						
23																						
24																						
25																						
26										1												
27																						
28																						
29		1																				
30																						
31																1						
32																						
33																						
34																						
35																						
36																						
37																						
38																						
39																						
40																						
41																						
42																						

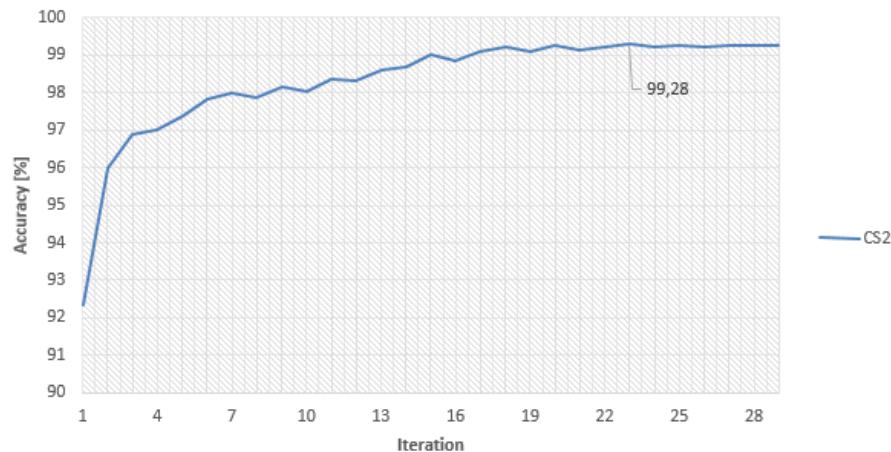
Figure 4.7: Confusion matrix(1) obtained from CNN in CS1.



**Figure 4.8:** Confusion matrix(2) obtained from CNN in CS1.

## Results, Case Study 2

Case Study 2 uses the 32x32 central patches and reaches 99.28% accuracy. Below in the table the classification rate and the false alarm rate obtained by averaging class rates.



**Figure 4.9:** Test accuracy of CS2 (central patches).

N. of signs	Avg. classification rate [%]	Avg. false alarm rate [%]
11763	99.17	0.58



**Figure 4.10:** The classification errors in CS2.

Class					
Number	Test samples	False neg.	Classif. rate [%]	False pos.	F. a. rate [%]
0	70	2	97.14	0	0.00
1	740	8	98.91	4	0.54
2	750	4	99.47	10	1.33
3	470	4	99.46	2	0.43
4	660	0	100.00	3	0.46
5	620	16	97.42	3	0.49
6	140	0	100.00	0	0.00
7	480	8	98.33	9	1.88
8	470	6	98.72	15	3.19
9	490	2	99.59	17	3.47
10	670	11	98.36	2	0.29
11	440	4	99.09	1	0.23
12	700	1	99.86	0	0.00
13	720	0	100.00	1	0.14
14	260	0	100.00	1	0.38
15	210	1	99.52	1	0.48
16	140	0	100.00	0	0.00
17	370	0	100.00	0	0.00
18	400	0	100.00	1	0.25
19	70	4	94.29	0	0.00
20	120	2	98.33	1	0.83
21	110	4	96.36	0	0.00
22	130	0	100.00	1	0.77
23	170	3	98.24	1	0.59
24	90	0	100.00	0	0.00
25	500	1	99.80	1	0.20
26	200	1	99.50	2	1.00
27	80	1	98.75	0	0.00
28	180	1	99.44	0	0.00
29	90	2	97.78	1	1.11
30	150	1	99.33	1	0.67
31	260	0	100.00	4	1.54
32	80	0	100.00	0	0.00
33	229	0	100.00	0	0.00
34	140	1	99.29	1	0.71
35	400	0	100.00	0	0.00
36	130	0	100.00	0	0.00
37	70	1	98.57	0	0.00
38	690	0	100.0	0	0.00
39	100	1	99.00	0	0.00
40	120	0	100.00	2	1.67
41	80	0	100.00	1	1.25
42	80	0	100.00	1	1.25

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	68				1				1													
1		732	2			1		1	2				1						1			
2		3	746						1													
3			2	466	2																	
4					660																	
5			4	1		604	5	4	1													
6							140															
7			2					472	6													
8				1	2			2	464	1												
9		1								488	1											
10										11	659											
11										3	437											
12												699		1								
13													720									
14														260								
15							1								209							
16																140						
17																	370					
18																		400				
19									1										66			
20																				118		
21																					106	
22																						
23									1											1		
24																						
25																						
26																						
27										1												
28																						
29															1							
30																						
31																						
32																						
33																						
34									1													
35																						
36																						
37																						
38																						
39																						
40																						
41																						
42																						

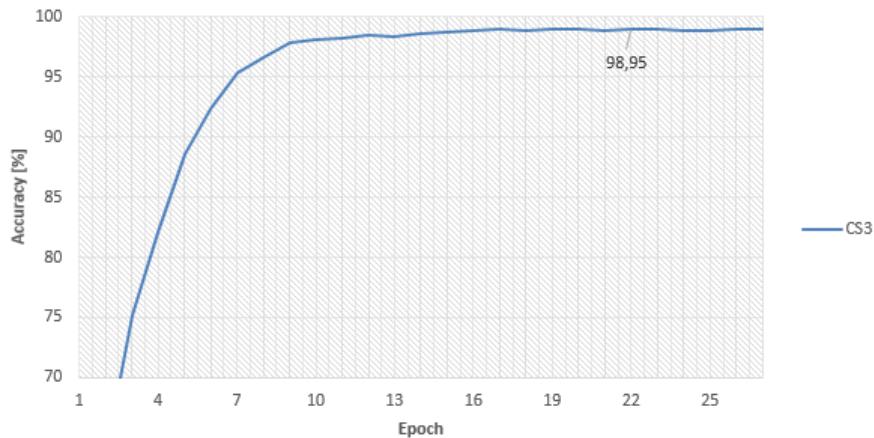
Figure 4.11: Confusion matrix(1) obtained from CNN in CS2.

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	
																					0
																					1
																					2
																					3
																					4
																			1	5	
																			6		
																			7		
																			8		
																			9		
																			10		
																		1	11		
																		12			
																		13			
																		14			
																		15			
																		16			
																		17			
																		18			
	2																	19			
1																		20			
	1																	21			
130																		22			
167																	1	23			
90																		24			
499	1																	25			
1		199																26			
	79																	27			
	1		179															28			
		88	1															29			
		1	149															30			
			260															31			
				80														32			
					229													33			
						139												34			
							400											35			
								130										36			
									69							1		37			
										690								38			
											1							39			
											99							40			
												120						41			
													80					42			

**Figure 4.12:** Confusion matrix(2) obtained from CNN in CS2.

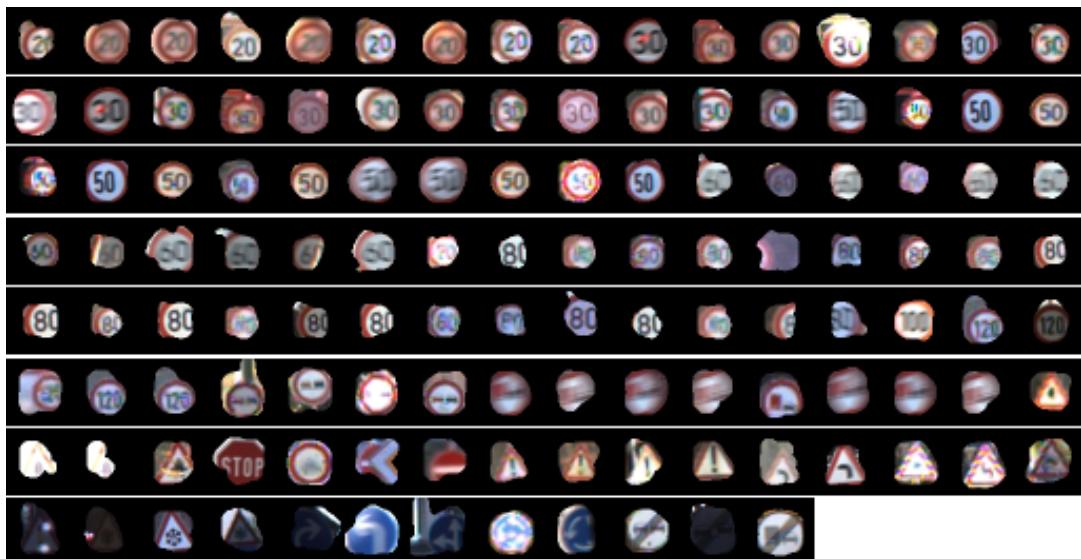
### Results, Case Study 3

Case Study 2 uses the segmented images and reaches 98.95% accuracy. Below in the table the classification rate and the false alarm rate obtained by averaging class rates.



**Figure 4.13:** Test accuracy of CS3 (central patches).

N. of signs	Avg. classification rate [%]	Avg. false alarm rate [%]
11763	98.93	0.77

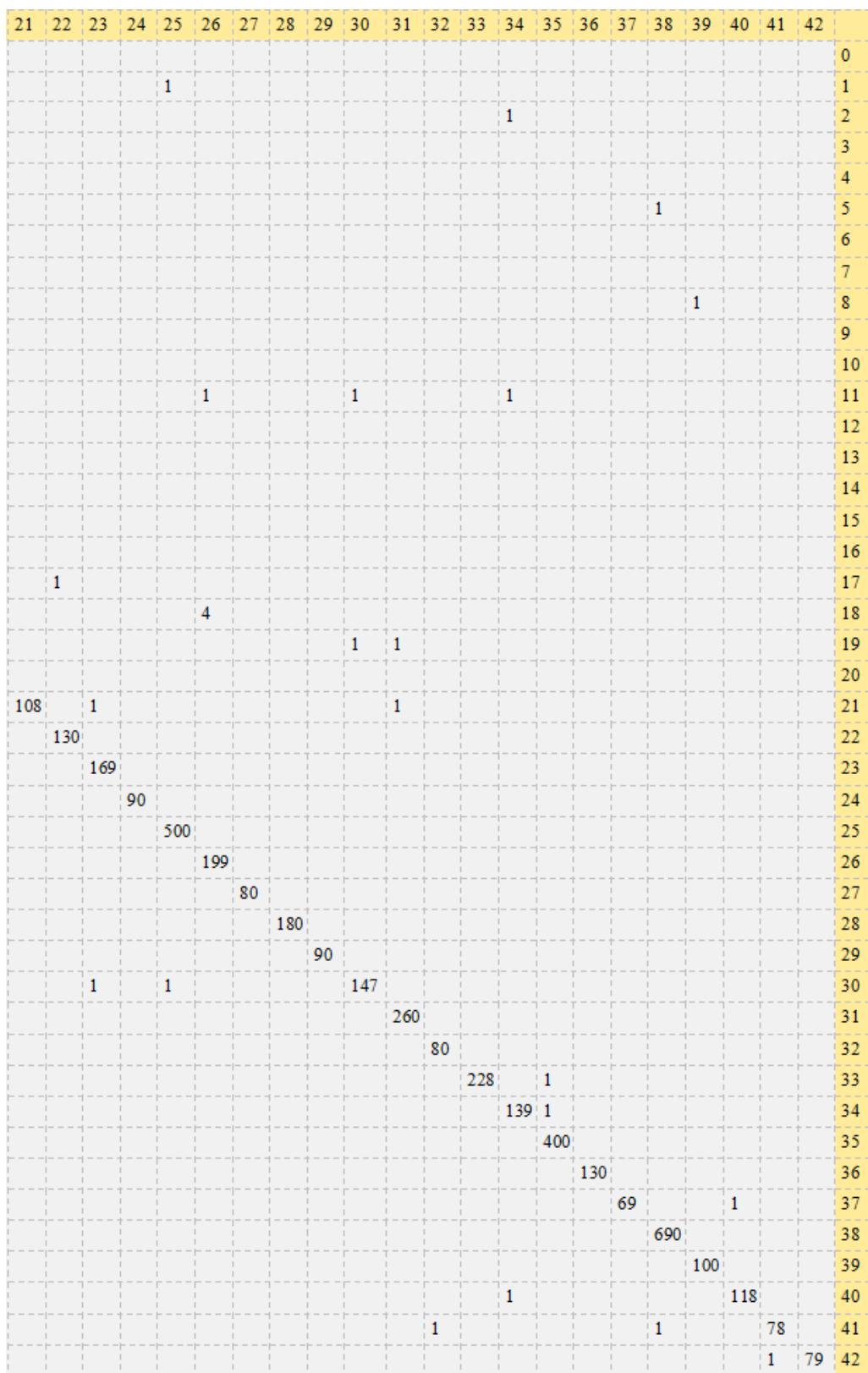


**Figure 4.14:** The classification errors in CS3.

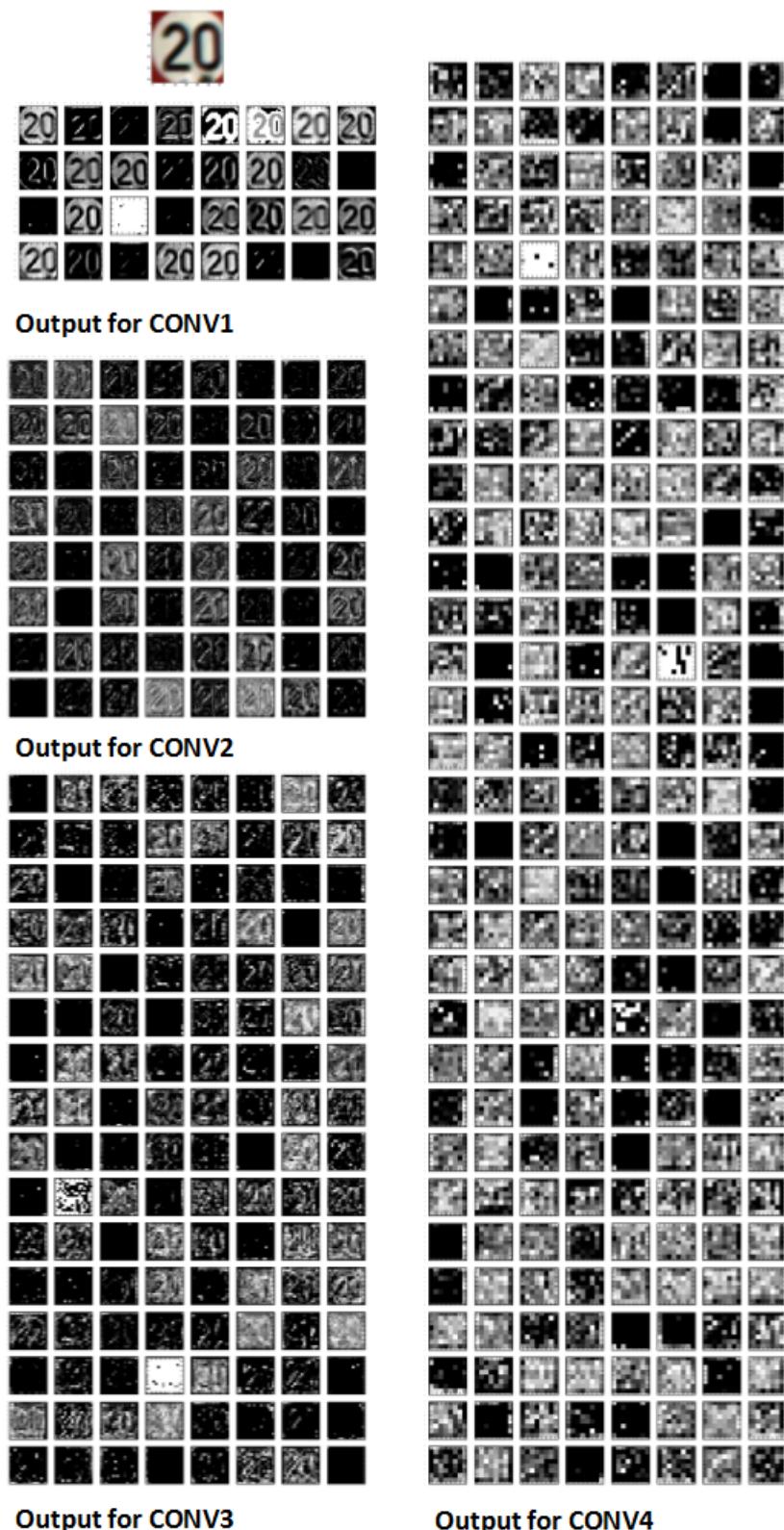
Class					
Number	Test samples	False neg.	Classif. rate [%]	False pos.	F. a. rate [%]
0	70	9	87.14	0	0.00
1	740	18	97.57	12	1.62
2	750	15	98.00	18	2.40
3	470	12	97.45	5	1.06
4	660	1	99.85	6	0.90
5	620	22	96.45	15	2.42
6	140	0	100.00	1	0.71
7	480	1	99.79	12	2.50
8	470	5	98.94	7	1.49
9	490	4	99.18	10	2.04
10	670	8	98.80	0	0.00
11	440	4	99.09	3	0.68
12	700	0	100.00	1	0.14
13	720	0	100.00	3	0.42
14	260	1	99.61	0	0.00
15	210	1	99.52	3	1.43
16	140	0	100.00	0	0.00
17	370	2	99.46	1	0.27
18	400	4	99.00	0	0.00
19	70	2	97.14	1	0.14
20	120	0	100.00	1	0.83
21	110	2	98.18	0	0.00
22	130	0	100.00	1	0.77
23	170	1	99.41	2	1.18
24	90	0	100.00	0	0.00
25	500	0	100.00	2	0.40
26	200	1	99.50	5	2.50
27	80	0	100.00	0	0.00
28	180	0	100.00	0	0.00
29	90	0	100.00	0	0.00
30	150	3	98.00	2	1.33
31	260	0	100.00	2	0.77
32	80	0	100.00	1	1.25
33	229	1	99.56	0	0.00
34	140	1	99.29	3	2.14
35	400	0	100.00	2	0.50
36	130	0	100.00	0	0.00
37	70	1	98.57	0	0.00
38	690	0	100.00	2	0.29
39	100	0	100.00	1	1.00
40	120	2	98.33	1	0.83
41	80	2	97.50	1	1.25
42	80	1	98.75	0	0.00

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	61	7	1		1																	
1		722	14						1										2			
2		5	735			4				4			1									
3			1	458		10	1															
4				659	1																	
5			5	5	598			11														
6					140																	
7						479	1															
8							1	465	3													
9								1	484					1		2						
10									7	662						1						
11										436									1			
12											700											
13												720										
14													259		1							
15			1											209								
16															140							
17											1					368						
18																396						
19																68						
20																	120					
21																		108				
22																			1			
23																						
24																						
25																						
26											1											
27																						
28																						
29																						
30												1										
31																						
32																						
33																						
34																						
35																						
36																						
37																						
38																						
39																						
40			1																			
41																						
42																						

Figure 4.15: Confusion matrix(1) obtained from CNN in CS3.



**Figure 4.16:** Confusion matrix(2) obtained from CNN in CS3.



**Figure 4.17:** An example of the data volume in intermediate levels of the CNN. In the image the output of CONV 1, CONV2, CONV3 and CONV4, case study 2, test sample from “20 Speed Limit”.

# 5

## Conclusions

This thesis proposes a deep convolutional network with a fewer number of parameters and memory requirements in comparisons to existing models.

### 5.1 Observations on the model and training

In this project several convolutional networks with different depth and type of layers were trained and compared. One model was chosen and used to explore in more details the impact of different preprocessing normalizations.

#### 5.1.1 The model

**Feature layers.** The part of the network highlighted in red in Figure 3.1 generates features. In its CONV layers we make use of 3x3 size filters because they are “large enough” to capture the variations on images initially resized to 32x32. Larger filters (11x11) were popular in 2012 with AlexNet (input data 227x227) and in [18] on the ImageNet dataset (input images 224x224). In this case the images are smaller and we consider a valid choice in the filter size to be between 3x3 or 5x5. This model that has been put into test is inspired by the VGGNet described in [22], where in fact they choose 3x3 filters and a deeper network in place of a shorter one with larger filters. Obviously in this thesis we cannot afford to use as many layers and filters as they do, but we follow the basic guidelines. To simplify the network and reduce the computational burden the number of filters has been reduced and especially on the fully connected layers we limit the number of weights. Each weight in a FCC in fact requires a full set of connections to the upper and lower layer. On the number of filters, we chose an increasing number of filters as a power of 2 (common setting, as explained in [24]) as moving down the layers. We make use of the ReLu nonlinearity past the first FCC and softmax nonlinearity on the output FCC. Krizhevsky *et al.* [18] shows that a rectified linear neuron gives no worse performance than sigmoid or tanh but makes training about 6 times faster. Networks such as VGGNet [29], [39], AlexNet (2012) and ZFNet (2013) use at least three MAXPOOL2 or even MAXPOOL3 layers to subsample the features and reduce them to 6x6 or 7x7. Having already small images we preferred not to subsample the data any further than the current setting.

**Classification layers.** The stack of the last four layers (DOL1 - FC1 - DOL2 - FC2), highlighted in blue in Figure 3.1 is a common setting, also seen for instance

in AlexNet, ZFNet, GoogLeNet and many more. With this part of the network we do not produce features any more and use it entirely for classification. The number of connections in these layers was chosen to be lower than VGGNet but still higher than the number of output classes (as in VGG). In this relatively small network the number of connections just on the first fully connected layer is 4194304. As a comparison, the deepest architecture in [29] has more than 130 million weights in their first FCC. The second FCC though only carries 11008 connections. In this context dropout work as regularizers that randomly set input values to zero and not only helps with generalization but also saves a lot of time in the training.

### 5.1.2 The training.

We find an optimal learning rate to be the largest learning rate that does not cause divergence of the training criterion. This learning rate speeds up the learning almost always guaranteeing a validation accuracy above 85% on the first two or three epochs (214 iterations with batch size 128). We notice though that the boundary between values for the learning rate that provide fast convergence and training loss explosion is quite narrow. An initial value set to 0.01 seems to be optimal.

Regarding the update method, the reason we favor Nesterov momentum over Adagrad update is quite arbitrary, since the slightly better performances on CNN2 tests do not fully guarantee the superiority of the first over the second. AdaDelta may yield better local minima in some cases but is much slower and we find its convergence speed to be much dependent on the initial learning rate. The reason is probably that a bad choice in the learning rate takes too much time to stabilize.

We use a softmax nonlinearity in the output layer, and cross entropy works generally very well with it because the derivative of the logarithmic term provides mathematical simplifications that speed up the learning. Cross entropy cost is appropriate to this classification problem where the goal is to minimize the number of misclassified training samples. With softmax we basically impose an exponentially increasing error the closer an output comes to being “1” when it should be “0”, and vice versa.

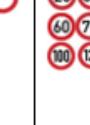
## 5.2 Discussion

**CS1.** The classification accuracy obtained, although not as high as [48], [33] and [46] is still above human performance 98.84% ([36]). The class average accuracy is 99.40% thanks to many classes reaching 100% accuracy. The average false alarm rate by class is 0.63%. [33] and [46] use ensemble of classifiers trained on slightly different versions of the dataset thanks to random distortions including translation, scaling and rotation. This is frequently called data augmentation and allows to increase the size of the training dataset. We think that the network used in this thesis could potentially reach higher accuracies with an ensemble method. By observing in detail the distribution of the errors we notice them to be distributed mostly within the speed limit group. Categories such as 30, 50, 70 and 80 speed limit are among the largest categories and it is to be expected an higher probability of having outlayer samples that are particularly difficult to classify. These categories not only have lower accuracies but also higher false alarm rates. We found the blue signs to be

very easy to classify. The winners of IJCNN 2011 [33] have also obtained a blue classes accuracy (99.89%) higher than the average on the blue signs.

**CS2.** The number of errors from CS2 is lower than CS1 and its highest test accuracy reaches 99.28%. The average classification rate by class is 99.17% and the average false alarm rate by class is 0.58%. This could be assumed as a "lucky" case as the vast majority of the images are already well centered on the sign, and the central patch excludes the background, which can be regarded as misleading information. Central patches have been used also in [18] on the ImageNet dataset. In this Case Study the cropping is even more impactful as it cuts off an even larger percentage of the original image.

**CS3.** CS3 performances are comparable to those of CS1. Training on the segmented images still allows a stable convergence up to just 0.03% classification accuracy from the raw images. CS3 fails mostly on speed signs which are also among the most frequent signs in the dataset and in real environments as well. It seems worth to consider the possibility of an improvement in accuracy with segmented datasets in future works.

	Blue	Danger	End-of	Red-round	Red-other	Speed	Spezial
							
CS1	100.00	99.75	100.00	98.29	99.64	97.62	99.73
CS2	99.61	98.70	100.00	98.91	99.37	98.68	99.96
CS3	99.46	99.37	99.06	97.72	99.37	96.90	99.77
Committee of CNNs (Ciresan et al.) [33]	99.89	99.07	99.72	99.74	99.93	99.47	99.22
Human perf. [23]	99.72	98.67	98.89	98.00	99.93	97.63	100.00
Multi-scale CNNs (Sermanet et al.) [48]	97.18	98.03	94.44	99.21	99.87	98.61	98.63

**Figure 5.1:** Comparison of classification accuracies by class groups used in IJCNN 2011 [23].

**CS1 vs CS2 vs CS3.** The accuracy obtained from CS2 suggests that TSR can in fact gain benefit from central patches and it is a hint to further explore other forms of preprocessing. This is though a particular fortunate case, since most traffic signs are centered in the image but it shows nonetheless that the network learns better from data that carries more information, such as the central patch. Despite CS1 having lower classification accuracy in some classes, we notice the average classification accuracy by class being higher than CS2, as in CS1 the majority of errors come from few classes. By looking at the errors we notice on CS1 many of the wrongly

classified samples being characterized by either very high or low illumination. Errors from CS2 show more strong distortions that may be reasonably the cause of the misclassification. From Figure 5.1 we notice CS1 and CS2 performing better in different class groups and may benefit from an ensemble (similar to the one proposed in [33]) of the two of them. CS3 accuracies by groups instead are similar to CS1 but slightly lower. Overall the results are nonetheless quite competitive in all three case studies and on average comparable to human performances.

# Bibliography

- [1] Li Deng, "Three Classes of Deep Learning Architectures and Their Applications: A Tutorial Survey", 2012.
- [2] Braunagel, C., Kasneci, E., Stolzmann, W., Rosenstiel, W. "Driver-activity recognition in the context of conditionally autonomous driving." In 2015 IEEE 18th International Conference on Intelligent Transportation Systems (pp. 1652-1657) IEEE, 2015.
- [3] Smolensky Paul, 1986. "Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory".
- [4] Michael Nielsen, Online book "Neural Networks and Deep Learning".
- [5] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio, NIPS 2012 deep learning workshop. (BibTeX) "Theano: new features and speed improvements".
- [6] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio. June 30 - July 3, Austin, TX (BibTeX). "Theano: A CPU and GPU Math Expression Compiler".
- [7] [www.journal.frontiersin.org/article/10.3389/frobt.2015.00028/fullh4](http://www.journal.frontiersin.org/article/10.3389/frobt.2015.00028/fullh4).
- [8] Y. LeCun and M.A. Ranzato, ICML 2013 tutorial.
- [9] [www.cs.utexas.edu](http://www.cs.utexas.edu)
- [10] G.E. Hinton and R.R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks", Science, 28 July 2006, Vol. 313. no. 5786, pp. 504 - 507.
- [11] [www.wildml.com](http://www.wildml.com)
- [12] <http://deeplearning4j.org/restrictedboltzmannmachine.html>
- [13] Kyung Hee University, Yong-In, Korea, 4 th World Conference on Applied Sciences, Engineering Technology 24-26 October 2015, Kumamoto University, Japan , "A Single Depth Sensor Based Human Activity Recognition via Deep Belief Network".
- [14] Mike Schuster and Kuldip K. Paliwal, Bidirectional Recurrent Neural Networks, Trans. on Signal Processing 1997.
- [15] Alex Graves, Santiago Fernandez, and Jurgen Schmidhuber, "Multi-Dimensional Recurrent Neural Networks", ICANN 2007.
- [16] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, Yoshua Bengio, "Gated Feedback Recurrent Neural Networks", arXiv:1502.02367, 2015.
- [17] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, 2015, "Deep Residual Learning for Image Recognition".
- [18] A. Krizhevsky, I. Sutskever, and G. Hinton. "Imagenet classification with deep convolutional neural networks". NIPS, 2012.

- [19] Ballas et al., “Delving Deeper into Convolutional Networks for Learning Video Representations”, 2016.
- [20] <http://deeplearning.net/tutorial/lstm.html>.
- [21] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, Trevor Darrell, 2015, “Long-term Recurrent Convolutional Networks for Visual Recognition and Description”.
- [22] K.Symonian, A. Zisserman, “Very deep convolutional networks for large-scale image recognition”, 2015.
- [23] <http://benchmark.ini.rub.de/?section=homesubsection=news>
- [24] CS231, Stanford University Winter 2016: <http://cs231n.stanford.edu/syllabus.html>
- [25] Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. R. (2012): “Improving neural networks by preventing co-adaptation of feature detectors”. arXiv preprint arXiv:1207.0580.
- [26] Srivastava Nitish, Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. R. (2014): “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. Journal of Machine Learning Research, 5(Jun)(2), 1929-1958.
- [27] S. Ioffe, C. Szegedy: “Batch normalization: accelerating deep network training by reducing internal covariate shift”, 2015.
- [28] Vezhnevets, Konouchine , “GrowCut” - Interactive Multi-Label N-D Image Segmentation By Cellular Automata, 2005.
- [29] IOFFE, Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, 2015
- [30] Xavier Glorot, Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks".
- [31] Sutskever, Martens et al. , “On the importance of initialization and momentum in deep learning”, 2013.
- [32] <https://searchcode.com/codesearch/view/15851013/>
- [33] Dan Ciresan, Ueli Meier, Jonathan Masci and Jürgen Schmidhuber, “Multi-Column Deep Neural Network for Traffic Sign Classification”, January 2012.
- [34] Delalleau, Bengio, “Shallow vs. Deep Sum-Product Network”, 2011.
- [35] Le, T. T., Tran, S. T., Mita, S., Nguyen, T. D. “Real time traffic sign detection using color and shape-based features.” In Intelligent Information and Database Systems (pp. 268-278). Springer Berlin Heidelberg, 2010.
- [36] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German Traffic Sign Recognition Benchmark: a Multi-Class Classification Competition. In Neural Networks (IJCNN), The 2011 International Joint Conference on, pages 1453–1460. IEEE, 2011.
- [37] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. “Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition”, 2012.
- [38] J. Greenhalgh and M. Mirmehdi, “Real-time detection and recognition of road traffic signs,” IEEE Trans. Intell. Transp. Syst., vol. 13, no. 4, pp. 1498–1506, Dec. 2012.
- [39] S. Maldonado-Bascon, S. Lafuente-Arroyo, P. Gil-Jimenez, H. Gomez-Moreno, and F. Lopez-Ferreras, “Road-sign detection and recognition based on support vector machines”, IEEE Trans. Intell. Transp. Syst., vol. 8, no. 2, pp. 264–278, Jun. 2007.

- [40] S. Maldonado-Bascón, J. Acevedo-Rodríguez, S. Lafuente-Arroyo, A. Fernández-Caballero, and F. López-Ferreras, "An optimization on pictogram identification for the road-sign recognition task using SVMs," *Comput. Vis. Image Understand.*, vol. 114, no. 3, pp. 373–383, 2010.
- [41] F. Zaslavskiy, B. Stanciulescu, and O. Hamdoun, "Traffic sign classification using K-d trees and random forests," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, San Diego, CA, USA, 2011, pp. 2151–2155.
- [42] M. Meuter, C. Nunny, S. M. Gormer, S. Muller-Schneiders, and A. Kummert, "A decision fusion and reasoning module for a traffic sign recognition system," *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 4, pp. 1126–1134, Dec. 2011.
- [43] A. Ruta, Y. Li, and X. Liu, "Robust class similarity measure for traffic sign recognition," *IEEE Trans. Intell. Transp. Syst.*, vol. 11, no. 4, pp. 846–855, Dec. 2010.
- [44] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE Comput. Soc. Conf. CVPR*, 2005, vol. 1, pp. 886–893.
- [45] J. Greenhalgh and M. Mirmehdi, "Real-time detection and recognition of road traffic signs," *IEEE Trans. Intell. Transp. Syst.*, vol. 13, no. 4, pp. 1498–1506, Dec. 2012.
- [46] J. Jin, K. Fu, and C. Zhang, "Traffic Sign Recognition With Hinge Loss Trained Convolutional Neural Networks", *IEEE Transactions on Intelligent Transportation Systems*, VOL. 15, NO. 5, October 2014.
- [47] Mrinal Haloi, "Traffic Sign Classification Using Deep Inception Based Convolutional Networks", arXiv.org > cs > arXiv:1503.06643, 2015.
- [48] Pierre Sermanet, Yann LeCun, "Traffic Sign Recognition with Multi-Scale Convolutional Networks", *Neural Networks (IJCNN)* conference, 2011.