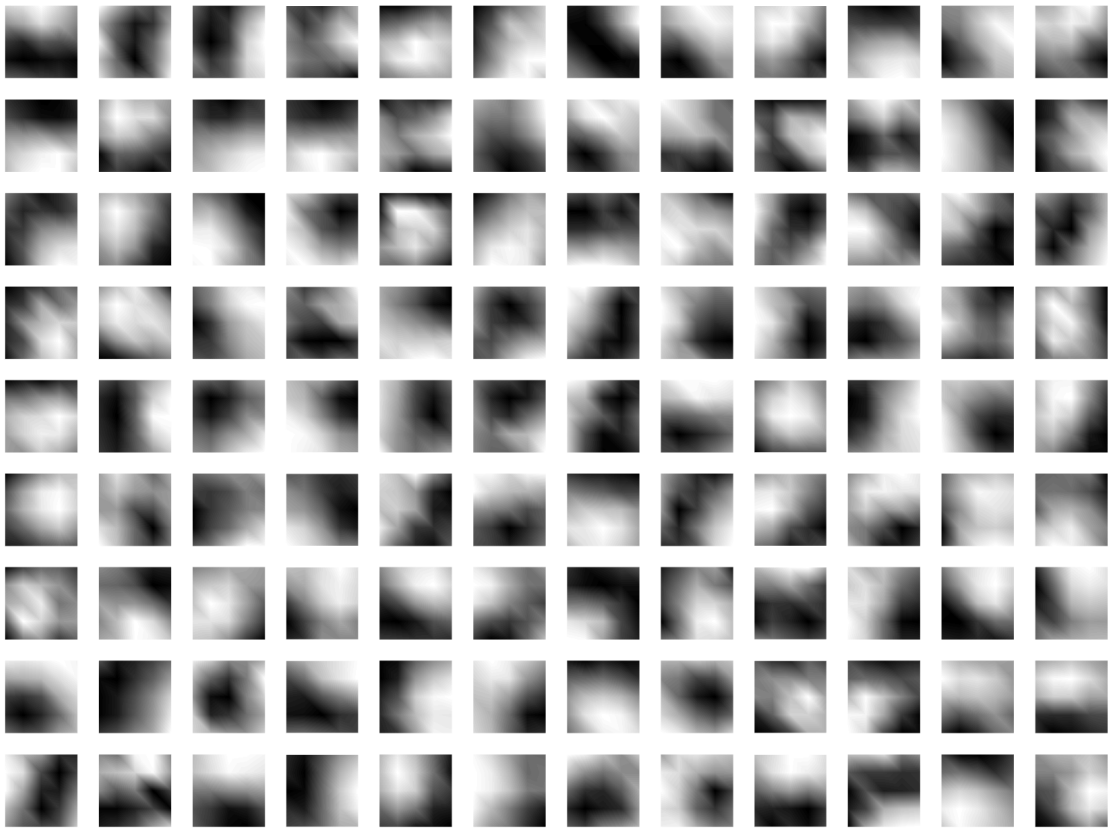
CHALMERS
UNIVERSITY OF TECHNOLOGY



Traffic sign classification with deep convolutional neural networks

Master's thesis in Complex Adaptive Systems

JACOPO CREDI

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis in Complex Adaptive Systems

Traffic sign classification with deep convolutional neural networks

JACOPO CREDI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Adaptive Systems research group
Chalmers University of Technology
Gothenburg, Sweden 2016

Traffic sign classification with deep convolutional neural networks
JACOPO CREDI

Supervisor: Luca Caltagirone, Department of Applied Mechanics
Examiner: Professor Mattias Wahde, Department of Applied Mechanics

Master's Thesis 2016:25
ISSN 1652-8557
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Adaptive Systems research group
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31 772 1000

Cover: Visualisation of the 108 filters in the first convolutional layer of a LeNet architecture trained with the GTSRB dataset.

Typeset in LaTeX
Printed by Chalmers Reproservice
Göteborg, Sweden 2016

Traffic sign classification with deep convolutional neural networks
JACOPO CREDI
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Adaptive Systems research group
Chalmers University of Technology

Abstract

In this work, a new library for training deep neural networks for image classification was implemented from the ground up, with the purpose of supporting GPU acceleration through OpenCL™, an open framework for heterogeneous parallel computing. The library introduced here is the first attempt at creating a C# deep learning toolbox, and can thus be more easily integrated with other projects under the .NET framework. The availability of cross-platform tools, covering as many developing environments as possible, can in fact accelerate the deployment of deep learning algorithms into a wide range of industrial applications, including advanced driver assistance systems and autonomous vehicles.

The library was tested on the German Traffic Sign Recognition Benchmark (GTSRB) data set, containing 51839 labelled images of real-world traffic signs. The performance of a classic deep convolutional architecture (LeNet) was compared to that of a deeper one (VGGNet), when trained with different regularisation methods. Dropout was observed to be particularly effective in counteracting overfitting for both models. Interestingly, the VGGNet model was observed to be more prone to overfitting, despite having a significantly lower number of parameters (∼462k) compared to the LeNet model (∼827k). This led to argue that architectural depth plays a crucial role in determining the capacity of a model, in accordance with some recent theoretical findings.

The best classification accuracy (96.9%) on the test GTSRB data was obtained using an ensemble of four deep convolutional neural networks, including both architectures and trained using both images converted to greyscale and the original RGB raw images.

Keywords: deep learning, convolutional neural networks, computer vision, machine learning, GPGPU, OpenCL.

Acknowledgements

First, I would like to express my gratitude to my supervisor, Luca, and my examiner, Mattias, for steering me in the right direction throughout this project and boldly reading through all my endless drafts. I also wish to thank all researchers and students in the VEAS division at Chalmers, for providing a nice and stimulating working environments during these months. Many thanks to all my friends, the old and the new, for always cheering me up and making me laugh when I needed it most. I would also like to express my gratitude to the people behind the Erasmus Mundus programme for allowing me and thousands of other students to understand that higher education truly should have no borders. Finally, special thanks to my family and Elena, for their everlasting love and support. This thesis is dedicated to you.

Jacopo Credi
Gothenburg, 12 June 2016

Contents

1
Introduction

The last few decades have seen a tremendous acceleration in the adoption of machine learning algorithms across an increasingly broad range of applications, many of which assist and simplify our everyday tasks. Spam filtering, speech understanding, face recognition, and e-commerce recommendations are only a few examples of applications in which machine learning methods are currently deployed.

In particular, with recent developments of general-purpose computing on graphics processing units (GPGPU) and the availability of large open data sets, training artificial neural networks (ANNs or NNs) with many hidden layers has become feasible. This approach to machine learning, now known as *deep learning*, has revolutionized research in computer vision, speech recognition and natural language processing, and is now rapidly being adopted in a large number of applications across a wide range of industrial sectors. Major technology companies such as Google, Microsoft, Facebook, Yahoo!, and IBM are currently re-thinking some of their core products and services to include deep learning-based solutions. At the same time, hardware producers such as Nvidia, Mobileye, Altera, Qualcomm, and Samsung have started research and development projects with the aim of efficiently implementing deep neural networks on chips or field-programmable gate arrays (FPGAs), in order to deploy state-of-the-art vision systems in smartphones, autonomous vehicles, and robots.

In the automotive industry, machine learning algorithms are playing an important role in the development of advanced driver assistance systems (ADAS) for improving the driver's safety and comfort. Deep learning is now considered by many automotive companies as one of the most promising emerging technologies, not only in the improvement of ADAS, but also in the broader context of autonomous vehicles, bound to revolutionize the entire automotive sector in the near future.

In particular, systems for automatic traffic sign recognition, such as Volvo's Road Sign Information (RSI), or Toyota's Road Sign Assist (RSA), have already been on the market for a few years and are key components of modern ADASs. Improving the accuracy of both the detection and classification of traffic sign would increase the effectiveness of current systems, and potentially play an important role in the development of future auto-pilot computer systems. There is therefore a strong interest among vehicle manufacturers to leverage recent developments in deep learning, namely deep Convolutional Neural Networks (CNNs, or ConvNets), which have achieved state-of-the-art performance in a wide range of computer vision tasks such as image classification, localisation, and detection [1–3].

1.1 Project goals

At the time of writing, GPGPU acceleration in all major deep learning libraries is supported through Nvidia's proprietary CUDA® platform and programming model, which makes these toolboxes heavily hardware-constrained. For this reason, the first goal of this project was to build a new .NET library for training convolutional networks from the ground up, using the OpenCL™ framework for GPGPU acceleration. As opposed to CUDA®, OpenCL™ is an open standard parallel programming framework for heterogeneous systems enabling the development of cross-platform, cross-vendor GPGPU solutions, and it is supported by major vendors, including Intel, IBM, AMD, and Nvidia.

The second goal of the project was then to train CNNs for traffic sign classification, using the German Traffic Sign Recognition Benchmark (GTSRB) data set [4], a publicly available data set for single-image, multi-class classification of traffic sign images. Building on previous works on this data set [5, 6], used as a source of inspiration for creating a baseline model, the project aimed at exploring different CNN architectures inspired by more recent works (e.g. [2, 3]), and at assessing the effect of new techniques and regularisation methods such as dropout [7] and batch normalisation [8].

1.2 Thesis outline

In this introductory chapter, the background and objectives of this thesis work were briefly outlined. The rest of the thesis is organized as follows:

- **Chapter 2** introduces deep learning methods and discusses recent advancements in image classification using CNNs, with a focus on traffic sign recognition.
- In **Chapter 3**, the Conv.NET libary for training convolutional neural networks in C# is presented. Rather than on implementation details, the description focuses on how the different available building blocks, optimisation and regularisation methods work.
- **Chapter 4** describes the GTSRB data set, two CNN architectures analysed in this work and other aspects of model selection.
- In **Chapter 5**, the results of training the two CNN architectures on the GTSRB data set are presented and discussed. In particular, the analysis focuses on the effect of using different regularisation methods and on how the two architectures differently benefit from using RGB over greyscale images for training.
- **Chapter 6** concludes this work with a discussion of its findings and contributions, points out limitations, and outlines directions for future research.

2
Deep learning

The design and development of algorithms allowing a machine to learn from large amounts of data and make predictions about the future is critically dependent on the process of extracting the most informative and non-redundant information from such data in their raw form. This process of *feature extraction* is traditionally carried out by humans and requires in general a great deal of expertise and labour, often including trial and error approaches. Increasingly, this process is being replaced by *representation learning*, a set of methods to automatically learn and discover new representations of the raw data, often outperforming traditional hand-crafted feature extraction [9]. In this chapter, a family of representation learning methods known as *deep learning* will be introduced.

2.1 Deep neural networks

Deep learning, a set of machine learning methods based on artificial neural networks (ANNs), is proving extremely successful in a wide range of tasks, particularly in the context of *supervised learning*, i.e. in inferring a function mapping given input data to desired given output values, in order to map new, unseen examples. Deep neural networks differ from other machine learning algorithms in that the representation of features is organized over multiple layers of nonlinear processing units, which transform the representation at one level into a slightly more abstract representation at a higher level. Typically, the first layers of a deep NN discover very simple features of the data (e.g. edges or colour gradients, if the data has the form of an image). More and more abstract and complex features are then automatically learned downstream in the network, by sequentially building them out of lower level ones, creating a hierarchy of representations.

Schmidhuber *et al.* [10] have recently reviewed deep learning methods extensively, while a more compact review, focused on recent advancements, can be found in LeCun *et al.* [11]. An excellent textbook on deep learning, written by Goodfellow *et al.* [12] is currently being published.

One key idea behind the strength of neural networks is that these hierarchically arranged features can to some extent be learned independently of one another, in a distributed fashion. A *distributed representation* of features is a highly desirable property in machine learning when dealing with high-dimensional data. In the case of a d-dimensional input space (e.g. $\mathbb{R}^d$), a distributed representation of n features can divide the space into $O(n^d)$ regions, corresponding to intersection of *half-spaces*. If such regions represent concepts, then exponentially many concepts can be distinguished using such a representation, since the different attributes of these concepts are *shared* [12].

It is worth mentioning that models of artificial neural networks have been known and used for decades. Pioneering work in neural computation dates back to 1943, with the introduction of McCulloch-Pitts neurons [13], simple threshold units with binary input and

output. The first feedforward neural network (FFNN) with adjustable synaptic weights, known as the *perceptron*, was introduced by Rosenblatt [14] in 1958 and used to build linear classifier machines with simple adaptive capabilities. A major breakthrough was the discovery that multilayer FFNNs can be trained using an algorithm known as backpropagation to compute the gradient of some objective function with respect to all model parameters, and then simply applying gradient descent to update such parameters (see Aside 2.1.1). This discovery is now attributed to Werbos [15] (PhD thesis, 1974), although several other researchers independently re-discovered it in the following years.

In 1989, Hornik *et al.* [16] showed that an FFNN with as few as a single hidden layer can approximate any function to any desired degree of accuracy. In light of this fundamental theoretical result, known as the *universal approximation theorem*, the whole idea of training neural networks with a large number of hidden layers may not sound as the optimal approach, as it arguably increases the complexity of the model. However, it was later discovered, mainly empirically, that approximating complex functions using deeper models can be much more *efficient* in terms of the total number of hidden units (and thereby of parameters) in the model. Intuitively, *compositionality*, i.e. the possibility of creating higher level features by combining lower level features, adds another exponential advantage to the statistical efficiency of a neural network, on top of the exponential advantage given by the distributed nature of the represented features. This insight is increasingly supported by theoretical results. For example, Delalleau and Bengio [17] have shown that functions in a certain class of polynomials in n variables can be represented by sum-product networks of depth $O(k)$ using $O(nk)$ hidden units, but require $O((n-1)^k)$ units in the case of a 1-hidden layer network.

2.2 Convolutional neural networks

Convolutional neural networks (CNNs, or ConvNets) are a type of feedforward neural networks designed to handle data organized in the form of arrays with some degree of spatial structure (i.e. locally correlated). A typical example is a color (RGB) image, which is essentially a stack of three 2D arrays, and can be seen as a 3D *tensor*, with the third dimension being the colour channel. The architecture of CNNs was born in the 1980s, with Fukushima's *neocognitron* [18], inspired by a neurophysiological model of mammals' visual primary cortex introduced by Hubel and Wiesel [19] in the 1960s. Many fundamental aspects of this early model are still used by modern CNNs.

At its core, a convolutional neural network is a stack of layers transforming a 3D input tensor to a 3D output tensor in a feedforward fashion. These layers perform linear or nonlinear transformation of their input, and these operations may or may not involve additional parameters and hyperparameters. Although this description may arguably apply to any form of FFNN, convolutional neural networks are different in that the extraction of features in the first layers, as well as the composition of such features into higher-level features, is performed through mathematical operations called discrete convolutions. The operating principles of CNNs are described in detail in Section 3.1.2.

Convolutional neural networks were first trained with backpropagation and gradient descent by LeCun *et al.* [20] in 1989, for the purpose of classifying handwritten digits. In the 1990s, CNNs-based algorithms were already used in commercial applications, such as reading cheques [21] (the CNN architecture known as "LeNet-5" used in this work is shown in Figure 2.1). However, at that time the computational power needed to train larger and

The backpropagation algorithm

The backpropagation algorithm is essentially an application of the chain rule of derivatives to the computation of the gradient of some loss or cost function L with respect to the parameters $\boldsymbol{\theta}^{(l)}$ of any layer l of a feedforward neural network in an inductive fashion. This makes it possible to use simple gradient descent (or its variants) to train a multilayer neural network.
Let

$$\nabla \boldsymbol{y}^{(l)} \coloneqq \nabla_{\boldsymbol{y}^{(l)}} L \coloneqq \Big(\frac{\partial L}{\partial y_1^{(l)}}, \dots, \frac{\partial L}{\partial y_{N_{\text{out}}^{(l)}}^{(l)}} \Big)^T$$

be the gradient of the loss function L with respect to output activations $\boldsymbol{y}^{(l)}$ in layer l. Assuming this gradient is *known*, the gradient with respect to the layer's parameters $\boldsymbol{\theta}^{(l)}$, i.e.

$$\nabla \boldsymbol{\theta}^{(l)} \coloneqq \nabla_{\boldsymbol{\theta}^{(l)}} L \coloneqq \Big(\frac{\partial L}{\partial \theta_1^{(l)}}, \dots, \frac{\partial L}{\partial \theta_{N_\theta^{(l)}}^{(l)}} \Big)^T$$

can be computed as

$$\frac{\partial L}{\partial \theta_j^{(l)}} = \sum_{i=1}^{N_{\text{out}}^{(l)}} \frac{\partial L}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial \theta_j^{(l)}} , \qquad \text{for } \; j = 1, \dots, N_\theta^{(l)} . \tag{2.1}$$

As long as $y_i^{(l)} = f(\boldsymbol{y}^{(l-1)}, \boldsymbol{\theta}^{(l)})$ is differentiable with respect to $\theta_j^{(l)}$ for all i, j, the gradient can be computed and used to update the parameters as

$$\boldsymbol{\theta}^{(l)} \leftarrow \boldsymbol{\theta}^{(l)} - \eta \nabla \boldsymbol{\theta}^{(l)} , \tag{2.2}$$

where η is a hyperparameter called the *learning rate*.
So far, $\nabla \boldsymbol{y}^{(l)}$ was assumed to be known. Now, let us consider two cases:

- If l is the output layer of the network, the gradient can simply be computed by taking the derivative of the loss function (as long as this is differentiable).
- If instead l is a hidden layer, then $\nabla \boldsymbol{y}^{(l)}$ can be computed from the gradient $\nabla \boldsymbol{y}^{(l+1)}$ of the layer above, again by applying the chain rule:

$$\frac{\partial L}{\partial y_k^{(l)}} = \sum_{i=1}^{N_{\text{out}}^{(l+1)}} \frac{\partial L}{\partial y_i^{(l+1)}} \frac{\partial y_i^{(l+1)}}{\partial y_k^{(l)}} , \qquad \text{for } \; k = 1, \dots, N_{\text{out}}^{(l)} , \tag{2.3}$$

  as long as $y_i^{(l+1)} = g(y_k^{(l)})$ is differentiable for all i, k.

By computing expressions 2.1 and 2.3 backwards, from the output layer to the input (hence the name backpropagation), all gradients can be computed, and the parameters of the network can be updated using expression 2.2 (or variants, as we shall see) so as to minimize the loss L.

Aside 2.1.1: A brief explanation of the backpropagation algorithm.

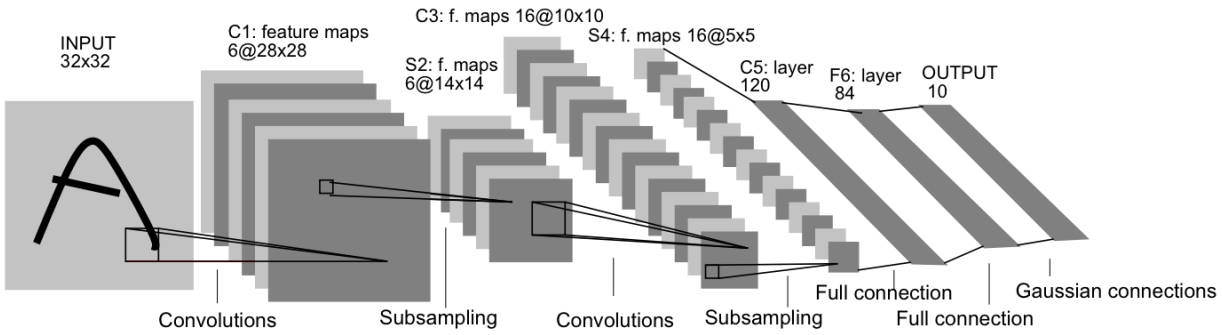


Figure 2.1: The architecture of LeNet-5, a CNN used by LeCun *et al.* [21] for document reading. Each plane is a feature map. Image reproduced with permission.

deeper models to solve more complex tasks was simply too large. This aspect, together with the rapid progress made in other areas of machine learning, hindered the diffusion of neural networks in the computer vision community for almost two decades.

A breakthrough result, published in 2006 by Hinton *et al.* [22], introduced an algorithm for greedy layer-wise pre-training algorithm to efficiently train particular kinds of deep NNs, and largely contributed in reviving the interest in deep learning. However, its recent success would have not been possible without two other essential ingredients. First, the appearance of large, open, and labelled data sets, such as the popular CIFAR10, CIFAR100, and ImageNet data sets of natural images. Secondly, the diffusion of cheap, massively parallel computing power in the form of GPUs (Graphics Processing Units) for general purpose computing, starting from the late 2000s.

Human-level performance in handwritten digit recognition on the benchmark MNIST database was achieved for the first time in 2011 by Cireşan *et al.* [23] by using a GPU-trained deep CNN. In the last 5 years, the popularity of CNNs in supervised image classification tasks has increased steadily, supported by an impressive series of successes. Most famously, in 2012 the ImageNet (ILSVRC) challenge, the largest contest in object recognition, with 1.2 million high-resolution images in 1000 classes, was won by Krizhevsky *et al.* [1] using a deep CNN (now known as "AlexNet"). To put this result into perspective, they achieved a top-5 classification error[1] of 15.3% on the test data, almost halving the second-best entry in the same competition (26.2%).

2.3 State-of-the-art in image classification

Since ILSVRC 2012, image classification competitions have consistently been won by teams employing CNNs with depth increasing over the years, as shown in Figure 2.2. AlexNet's architecture was similar to the basic LeNet architecture from the 1990s [21] (Figure 2.1), but deeper (5 convolutional layers and 3 fully-connected layers, with decreasing filter size) and larger (about 60 million parameters in total). In order to improve the network generalisation performance, Krizhevsky *et al.* employed two main techniques, nowadays widely used. First, they artificially augmented the training set by generating image translations and reflections along the horizontal axis, as well as jittering the RGB pixel intensities, to

[1]When measuring the top-5 classification score of an algorithm, the classification is considered correct if the target label is among the 5 best predictions of the algorithm, i.e. the 5 labels that the algorithm assigns the highest probability.

increase the robustness to noise of the learned features. Furthermore, they used a new powerful regularisation technique called *dropout* [7], described in detail in Section 3.2.2.

More recently, researchers have started exploring new architectures, with important differences from early models like LeNet. An example worth mentioning is the work of Szegedy *et al.* [24], a team based at Google, who engineered a particular kind of module (codenamed *Inception*) composed of several convolutional and pooling layers with filters of different size, running in parallel. Interestingly, they replaced the conventional fully-connected layers preceding the final classifier with a layer performing average pooling across the spatial dimensions, thus significantly reducing the number of parameters in the model (about 4 million in total). One of their architectures, a 22-layer CNN based on Inception modules and known as GoogLeNet, won the ILSVRC 2014 classification contest, achieving a top-5 ensamble error of 6.67% on the test data.

The second-best entry in the same challenge was a CNN by a team based at the Visual Geometry Group at Oxford University, and thus known as VGGNet (Simonyan and Zisserman [2]). The features learned by VGGNet models have proved to be extremely effective not only in classification (e.g. ImageNet), but also to transfer very well to other tasks, such as object localisation, image segmentation, and caption generation [2]. For this reason (and in part for the simplicity and homogeneity of this architecture, described in Section 4.2), VGGNet-inspired networks are currently among the preferred choice for image recognition tasks in the computer vision community. The drawback is that VGGNet-like architectures may have a large number of parameters (138 million, in the case of the ILSVRC 2014 runner-up entry) and are therefore computationally expensive to train and use for inference.

A recently developed technique called *batch normalisation*, introduced by Ioffe and Szegedy [8] in 2015, has proved very effective in making deep networks more robust to improper initialisation, thus speeding up the training. More generally, batch normalisation addresses a problem known as *internal covariate shift*, i.e. the change in the distribution of the input activations to each layer during the training process, due to model parameters being updated. As the name says, the method explicitly normalises the inputs to each layer over a training mini-batch (see Section 3.2.1), and it integrates this operation in the model architecture in a differentiable way, allowing gradient backpropagation. The downside of batch normalisation is that it is in general a slow operation, more difficult to execute in parallel than convolutions and matrix multiplications. However, this is greatly compensated by the increased robustness to initialisation, and also by the possibility of using considerably higher learning rates, thus requiring fewer iterations in order to train a model.

Even more recently, He *et al.* [3] (Microsoft Research Asia) showed that improving the performance of deep CNNs is not "as easy as stacking more layers" on top of each other. In fact, using ImageNet data they empirically showed that as the network depth increases, classification accuracy saturates and then starts degrading. Since the observed degradation also affects classification accuracy on the *training* set, the authors conjectured that the problem is not related to overfitting, but rather to exponentially low convergence rates. Hence, they proposed an interesting architecture with skip-connections performing identity mappings, in the hope that learning *residual functions* with respect to the identity function would successfully address the degradation problem. Their extremely deep *residual networks* (or "ResNets"), stacking up to 152 layers, have improved on the state-of-the-art in multiple benchmark data sets, not only in classification (1st place in ILSVRC 2015), but also in localisation, detection and segmentation challenges. [3]

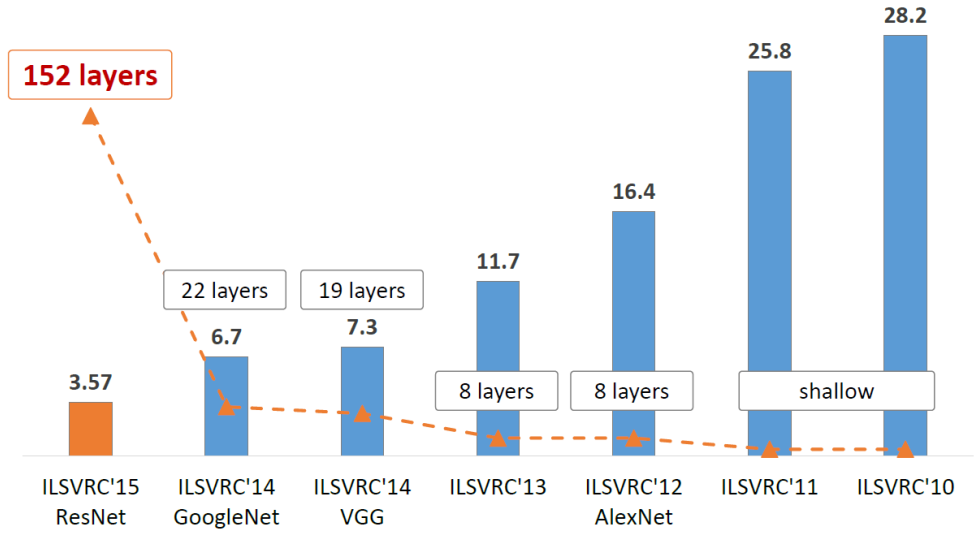


Figure 2.2: ImageNet Classification top-5 test error (in %) in the last six years. Image reproduced with permission by Kaiming He (Microsoft Research Asia).

2.4 Traffic sign recognition

Until the beginning of the decade, research in machine learning applied to the problem of traffic sign recognition has focused on the extraction of hand-crafted features for both the detection of a traffic sign instance and the subsequent classification into one of multiple classes. To this latter aim, extracted features were used to train classifiers, such as Bayes classifiers, logistic regression models, support vector machines (SVMs), or shallow NNs (see e.g. [25]).

Convolutional neural networks were first applied to this problem in the beginning of this decade, along with the appearance of large publicly available data sets of labelled traffic sign images. The first traffic sign classification challenge was held at the 2011 International Joint Conference on Neural Networks (IJCNN), using the GTSRB data set [4] (described in Chapter 4). The challenge was won by a team based at the IDSIA institute (Lugano, Switzerland) with an *ensemble* of 25 ConvNets (named by the authors *multi-column* deep neural network, or MCDNN), which achieved a classification rate of 99.46% on the test data (Cireşan *et al.*, 2012 [6]). This result surpassed human classification accuracy, estimated to be 98.84% on the same data.

Training several neural networks and using the resulting ensemble for inference is a common technique to obtain a higher classification accuracy than that of individual networks, and has thus become a common practice during challenges and competitions (see Section 3.2.2 for a more detailed discussion of the method of *model averaging*). Cireşan *et al.* used five different data preprocessing methods, and trained five networks in each case [6]. Apart from these techniques, the architecture of their CNNs, summarized in Table 2.1 (left column), is similar to that of LeNet-5 (Fig. 2.1), but features three convolutional layers with decreasing filter size.

Sermanet and LeCun [5] achieved the second place in the GTSRB challenge (98.31% accuracy) with a *multi-scale* convolutional neural network. Their architecture (see Table 2.1, middle column) was also similar to LeNet-5, featuring two convolutional layers (each followed by a max-pooling stage) and ending with two fully-connected layers. However, in their best performing networks, the output of the first pooling stage is directly fed into

the fully-connected classifier, in addition to that of the second stage. The claim of the authors was that such skip-connections would help combining global features extracted by the second stage with local and more detailed features learned by the first one.

More recently, Jin *et al.* [26] have obtained a new state-of-the-art classification accuracy of 99.65% on the GTSRB data, using an ensemble of 20 CNNs with a deeper architecture (Table 2.1, right column) and performing a thorough optimisation of the networks' hyperparameters. Finally, Haloi [27] has recently pushed the state-of-the-art on GTSRB to 99.81% using a very deep and complex architecture inspired to GoogLeNet [24]. However, no validation set nor cross-validation was used in this work (still unpublished), and therefore this result will not be taken into account in this study.

Layer	Cireşan *et al.* [6]	Sermanet & LeCun [5]	Jin *et al.* [26]
0	Input (48×48, RGB)	Input (32×32, grayscale)	Input (47×47, RGB)
1	Conv (7×7, 100 maps)	Conv (5×5, 108 maps)	Conv (5×5, 70 maps)
2	Nonlinearity (not specified)	Nonlinearity (rectified *tanh*)	Max-pooling (3×3, stride 2)
3	Max-pooling (2×2)	Max-pooling* (2×2)	Nonlinearity (ReLU)
4	Conv (4×4, 150 maps)	Conv (5×5, 108 maps)	Local normalisation
5	Nonlinearity (not specified)	Nonlinearity (rectified *tanh*)	Conv (3×3, 110 maps)
6	Max-pooling (2×2)	Max-pooling (2×2)	Max-pooling (3×3, stride 2)
7	Conv (4×4, 250 maps)	Fully-conn.* (100 units)	Nonlinearity (ReLU)
8	Nonlinearity (not specified)	Nonlinearity (rectified *tanh*)	Local normalisation
9	Max-pooling (2×2)	Fully-conn. (100 units)	Conv (3×3, 180 maps)
10	Fully-conn. (300 units)	Nonlinearity (rectified *tanh*)	Max-pooling (3×3, stride 2)
11	Nonlinearity (not specified)	Fully-conn. (43 units)	Nonlinearity (ReLU)
12	Fully-conn. (43 units)	Softmax	Local normalisation
13	Softmax		Fully-conn. (200 units)
14			Nonlinearity (ReLU)
15			Fully-conn. (43 units)
16			Softmax

Table 2.1: CNN architectures used in previous traffic sign classification works. In the architecture used by Sermanet and Lecun (middle column), layers marked by an asterisk are linked by a skip-connection.

2.4.1 A note on terminologies and conventions

There is currently no agreement on a common terminology for describing the arcihtecture of a CNN. For historical reasons, the first layers of a CNN, called *convolutional layers*, are sometimes regarded as composed of several different sub-stages performing multiple operations. A different scheme, called the *simple layer* terminology, considers every step of tensor processing as a layer in its own right, and thus the network as composed of a relatively large number of layers performing simple operations [12].

In this thesis (see e.g. Table 2.1), the latter approach is adopted, the reason being that a simpler terminology allows more modularity and flexibility in the construction of a model. Each simple layer can be used as a building block, regardless of what precedes and follows it. In fact, as previously discussed, the trend in the recent literature (e.g. [3, 24]) is towards going beyond the original CNN architectures used since the 1980s, and a more flexible terminology can be beneficial in this regard. The Conv.NET library, described in the next chapter, was thus implemented with this goal in mind.

3

The Conv.NET library

Several deep learning toolboxes have been released in recent years, thanks to a joint effort of academic departments (e.g. University of Montreal, UC Berkeley) and IT companies (e.g. Google, Microsoft, Facebook). However, while almost all of these toolboxes support GPU acceleration using Nvidia's proprietary CUDA® platform, none of them fully supports the OpenCL™ framework at the time of writing. For this reason, in this project a new deep learning library, named Conv.NET, was implemented from the ground up in C#, using the OpenCL™ framework for GPU acceleration through the open source OpenCL.NET bindings [28]. The reason behind the choice of C# as programming language is twofold. First, no C# deep learning library exist yet to the best of the writer's knowledge. Secondly, such a library could be more easily integrated with other projects written under the .NET framework, including ongoing projects in the Adaptive Systems research group at Chalmers University of Technology.

This chapter provides the reader with a brief description of the library: how to create a CNN for image classification, how the different building blocks work, and how the network can be trained and regularised.

3.1 Creating a network

As shown in code Listing 3.1, a `NeuralNetwork` object can be easily instantiated and added new building blocks using the `AddLayer()` method.

Each building block is implemented in its own class, derived from the base `Layer` class. In accordance to the simple layer notation, the base `Layer` class has two fields of class `Neurons`, called `inputNeurons` and `outputNeurons`, where the input and output activations of the layer are stored. Each derived class then implements different methods to transform input activations into output activations (*forward pass*), and to backpropagate error signals (also stored in dedicated fields of the `Neurons` class) backwards (*backward pass*), giving rise to different types of layers. As shown in Listing 3.1, each layer's parameters can be passed directly as arguments to the layer's constructor (see the XML documentation in the code).

When a layer is connected to an existing one, its `inputNeurons` field is pointed to the previous layer's `outputNeurons` object. In other words, denoting the input activations of layer l by $\mathbf{X}^{(l)}$ and its output activations by $\mathbf{Y}^{(l)} = F(\mathbf{X}^{(l)}; \boldsymbol{\theta}^{(l)})$, where $\boldsymbol{\theta}^{(l)}$ are the layer's parameters (if any), then $\mathbf{X}^{(l)} \equiv \mathbf{Y}^{(l-1)}$ for $l = 1, \dots, L$, where L is the number of layers in the network. In this notation, $\mathbf{Y}^{(0)}$ is the input tensor of the CNN, representing an image, and $\mathbf{Y}^{(L)}$ is the output tensor, representing a probability mass function over C possible classes, inferred by the network.

Similarly, denoting the gradient of the loss function L computed with respect to the input and output activations of layer l by $\nabla\mathbf{X}^{(l)} \doteq \nabla_{\mathbf{X}} L^{(l)}$ and $\nabla\mathbf{Y}^{(l)} \doteq \nabla_{\mathbf{Y}} L^{(l)}$, respectively, then $\nabla\mathbf{Y}^{(l-1)} \equiv \nabla\mathbf{X}^{(l)}$ for $l = 1, \dots, L$.

```
using Conv.NET;

/*
 * Creating a neural network with an input layer, 9 hidden layers,
 * and a 10-way output (softmax) layer
 */

NeuralNetwork network = new NeuralNetwork("NetworkName");

network.AddLayer(new InputLayer(1, 32, 32));
network.AddLayer(new ConvolutionalLayer(5, 32, 1, 0));
network.AddLayer(new ReLU());
network.AddLayer(new MaxPooling(2, 2));
network.AddLayer(new ConvolutionalLayer(5, 64, 1, 0));
network.AddLayer(new ReLU());
network.AddLayer(new MaxPooling(2, 2));
network.AddLayer(new FullyConnectedLayer(128));
network.AddLayer(new ReLU());
network.AddLayer(new FullyConnectedLayer(10));
network.AddLayer(new SoftMax());

/* Loading training and validation data (10 classes) */

DataSet trainingSet = new DataSet(10);
trainingSet.ReadData("PathToTrainingData");
trainingSet.ReadLabels("PathToTrainingLabels");

DataSet validationSet = new DataSet(10);
validationSet.ReadData("PathToValidationData");
validationSet.ReadLabels("PathToValidationLabels");

/* Setting training hyperparameters and training the network */

NetworkTrainer.LearningRate = 1e-5;
NetworkTrainer.MomentumCoefficient = 0.9;
NetworkTrainer.MiniBatchSize = 64;
NetworkTrainer.WeightDecayCoeff = 1e-4;
NetworkTrainer.EpochsBetweenEvaluations = 1;
NetworkTrainer.DropoutFullyConnected = 0.5;

NetworkTrainer.Train(network, trainingSet, validationSet);
```

Listing 3.1: Code snippet showing how to create and train a simple convolutional neural network using the Conv.NET library.

The notations $\mathbf{X}$ and $\nabla\mathbf{X}$ denote 4-dimensional tensors of size $M \times K_{\text{in}} \times H_{\text{in}} \times W_{\text{in}}$, where M is the size of the mini-batch used (i.e. the number of training examples passed into the network at each iteration - see Section 3.2.1), K_{in} is the input tensor depth, or number of channels (e.g. 3 in the case of a tensor representing a RGB image), and H_{in} and W_{in} are the input tensor's spatial dimensions (height and width of the image). Element $\mathsf{X}_{m,k,i,j}$ then denotes the input activation in spatial position (i, j), channel k, and example m in the mini-batch, whereas element $\nabla\mathsf{X}_{m,k,i,j}$ denotes its associated error value, to be computed with backpropagation. Similarly, $\mathbf{Y}$ and $\nabla\mathbf{Y}$ are tensors of size $M \times K_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}$, with self-explanatory notation. Only spatially square tensors are currently supported in the Conv.NET library, i.e. $H_{\text{in}} = W_{\text{in}}$ and $H_{\text{out}} = W_{\text{out}}$. Note that, as shown in Listing 3.1,

all these tensor dimensions need not be specified (apart from that of the input layer). In fact, they are automatically inferred during the network construction based on the layer parameters.

In the forward pass of the network, each layer transforms its input tensor $\mathbf{X}$ in a linear or nonlinear way, and with or without additional parameters $\boldsymbol{\theta}$, depending on the layer type. In the backward pass, the gradients with respect to both the input activations $\nabla\mathbf{X}$ and the parameters $\nabla\boldsymbol{\theta}$ (if any) are computed from $\nabla\mathbf{Y}$ using the backpropagation algorithm (see Aside 2.1.1). This can be mathematically summarised as follows:

$$\begin{aligned} \text{Forward pass:} \quad & \mathbf{Y}^{(l)} = F(\mathbf{X}^{(l)}; \boldsymbol{\theta}^{(l)}) \ . \\ \text{Backward pass:} \quad & \nabla\mathbf{X}^{(l)} = G(\nabla\mathbf{Y}^{(l)}; \boldsymbol{\theta}^{(l)}) \ , \\ & \nabla\boldsymbol{\theta}^{(l)} = H(\nabla\mathbf{Y}^{(l)}; \mathbf{X}^{(l)}) \ . \end{aligned} \tag{3.1}$$

Functions F, G, H depend on the type of layer l. In the rest of this section, the operations performed by each type of layer implemented in the library will be described in detail.

3.1.1 Input layer

The first layer of a neural network object must always be of type `InputLayer`. This is an auxiliary class performing the identity function in the forward pass (the backward pass has no meaning, and is thus not implemented), used to pass a mini-batch of examples into the network, through the method `FeedData()`.

In adding an `InputLayer` instance to an empty `NeuralNetwork`, the input tensor depth and spatial dimensions must be passed as arguments in the constructor. The mini-batch size (i.e. the first dimension of the tensor of activations) is instead set using a property of the static class `NetworkTrainer`, reflecting the fact that it is not a property of the neural network, but rather of the training procedure.

3.1.2 Convolutional layer

The convolutional layer, as the name implies, is the core component of a CNN, and its operating principles are inspired by those of hypercomplex processing cells in mammals' visual cortex (Hubel and Wiesel, 1962 [19]).

As shown in Figure 3.1, the output tensor $\mathbf{Y}$ of a convolutional layer can be seen as a stack of 2-dimensional arrays called *feature maps*, whose number determines the depth K_{out} of the output tensor and is one of the layer's hyperparameters.

Each unit $\mathsf{Y}_{m,k,i,j}$ in the k-th feature map is connected to units in a limited, local region of the input tensor $\mathbf{X}$, called the *receptive field* of the output unit (in grey in Figure 3.1). The receptive field is a sub-tensor of the input tensor, and its spatial dimensions are given by a hyperparameter F called the *filter size* or *kernel size*, whereas its depth always matches the input tensor depth H_{in}. Neighbouring output unit $\mathsf{Y}_{m,k,i+1,j+1}$ has a receptive field that is shifted by a certain *stride length* S (another hyperparameter) with respect to the previous one.

Commonly, the input tensor is framed with a border of zeros (*zero padding*), whose width P is another hyperparameter of the layer, in order to control the size of the output tensor $\mathbf{Y}$. In fact, its spatial dimensions are given by

$$H_{\text{out}} = \frac{H_{\text{in}} - F + 2P}{S} + 1 \ . \tag{3.2}$$

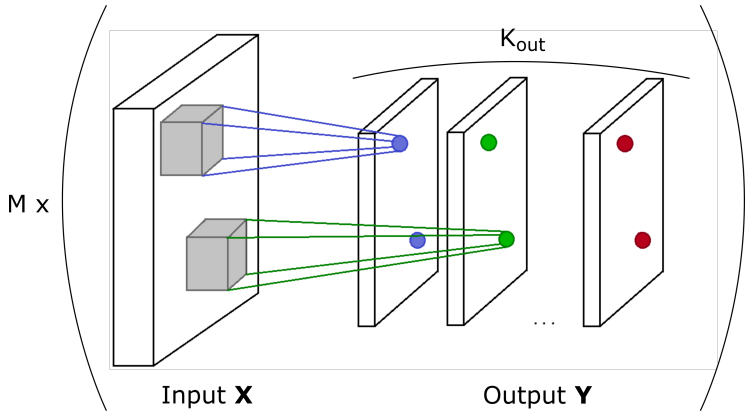


Figure 3.1: Graphical depiction of a convolutional layer.

Each connection between unit $\mathsf{Y}_{m,k,i,j}$ and the units in its receptive field is associated with a synaptic weight. However, all units in the same feature maps *share* the same set of $K_{\text{in}}F^2$ weights (plus one bias), known as a *filter*. The total number of parameters in a convolutional layer is therefore $K_{\text{out}}(K_{\text{in}}F^2+1)$, i.e. it does not depend on the input tensor spatial dimensions. This makes convolutional layers scalable to handle high-resolution images and intrinsically resistant to overfitting [29].

Besides keeping the number of parameters small, weight sharing is a fundamental property of convolutional layers, because it allows the network to detect the same *feature* in different positions of the input. This is clearly useful in the context of image classification, as the same edges, motifs, and parts can be present in different locations of an image. The operating principles of a convolutional layer therefore serves a two-fold purpose. First, the *local* nature of receptive fields is equivalent to introducing an infinitely strong prior over the parameters, explicitly saying that the layer should learn to capture only local spatial correlations in the input data. Secondly, as just discussed, *weight sharing* makes the detection of features translation-equivariant [12].

Mathematically, the operation performed in a convolutional layer is equivalent to a *discrete convolution*. Let $\mathbf{W}$ denote the 4-dimensional *kernel tensor* of the layer, where element $\mathsf{W}_{k,l,i,j}$ is the connection strength between a unit in channel k of the output and channel l of the input, with an offset of i rows and j columns between the two units. This tensor has size $K_{\text{out}} \times K_{\text{in}} \times F \times F$, where again F is the filter or kernel size.

The output tensor $\mathbf{Y}$ is then obtained by convolving the (padded) input tensor $\mathbf{X}$ and the kernel tensor $\mathbf{W}$, i.e.[1]

$$\mathsf{Y}_{m,k,a,b} = \sum_{l=0}^{K_{\text{in}}-1} \sum_{\substack{i=0 \\ a+i<H_{\text{in}}}}^{F-1} \sum_{\substack{j=0 \\ b+j<H_{\text{in}}}}^{F-1} \mathsf{X}_{m,l,a+i,b+j}\, W_{k,l,i,j} + b_k \ , \tag{3.3}$$

where b_k is the bias parameter of feature map k. This operation is equivalent to a local dot product, and is represented graphically in Figure 3.2.

[1]It is worth mentioning that it would be more precise to refer to the operation performed in Equation 3.3 as *cross-correlation*, instead of convolution. The two operations, however, are equivalent, in that they only differ by how the input tensor is accessed, and they return exactly the same result if the kernel tensor is *flipped* across its spatial dimensions. Usually, the cross-correlation operation is preferred in terms of implementation, as it simplifies the validity check of indices i and j.

Using the backpropagation algorithm, it is straightforward to compute the gradient $\nabla\boldsymbol{\theta} \coloneqq \nabla_{\boldsymbol{\theta}} L$ with respect to the layer's parameters (weights and biases):

$$\begin{aligned}\nabla \mathsf{W}_{r,s,t,v} \coloneqq \frac{\partial L}{\partial \mathsf{W}_{r,s,t,v}} &= \sum_{m,k,a,b} \frac{\partial L}{\partial \mathsf{Y}_{m,k,a,b}} \frac{\partial \mathsf{Y}_{m,k,a,b}}{\partial \mathsf{W}_{r,s,t,v}} \\ &= \sum_{m,k,a,b} \nabla \mathsf{Y}_{m,k,a,b} \frac{\partial}{\partial \mathsf{W}_{r,s,t,v}} \Big(\sum_{lij} \mathsf{X}_{m,l,a+i,b+j} \, W_{k,l,i,j} + b_k \Big) \\ &= \sum_{m,a,b} \nabla \mathsf{Y}_{m,r,a,b} \, \mathsf{X}_{m,s,a+t,b+v} \ , \end{aligned} \tag{3.4}$$

$$\begin{aligned}\nabla b_r \coloneqq \frac{\partial L}{\partial b_r} &= \sum_{m,k,a,b} \frac{\partial L}{\partial \mathsf{Y}_{m,k,a,b}} \frac{\partial \mathsf{Y}_{m,k,a,b}}{\partial b_r} \\ &= \sum_{m,k,a,b} \nabla \mathsf{Y}_{m,k,a,b} \frac{\partial}{\partial b_r} \Big(\sum_{lij} \mathsf{X}_{m,l,a+i,b+j} \, W_{k,l,i,j} + b_k \Big) \\ &= \sum_{m,a,b} \nabla \mathsf{Y}_{m,r,a,b} \ . \end{aligned} \tag{3.5}$$

The gradient with respect to the input activations (to be propagated backwards into the network in order to adjust parameters upstream) is instead:

$$\begin{aligned}\nabla \mathsf{X}_{m,r,s,t} \coloneqq \frac{\partial L}{\partial \mathsf{X}_{m,r,s,t}} &= \sum_{m',k,a,b} \frac{\partial L}{\partial \mathsf{Y}_{m',k,a,b}} \frac{\partial \mathsf{Y}_{m',k,a,b}}{\partial \mathsf{X}_{m,r,s,t}} \\ &= \sum_{m',k,a,b} \nabla \mathsf{Y}_{m',k,a,b} \frac{\partial}{\partial \mathsf{X}_{m,r,s,t}} \Big(\sum_{lij} \mathsf{X}_{m',l,a+i,b+j} \, W_{k,l,i,j} + b_k \Big) \\ &= \sum_{a,b} \nabla \mathsf{Y}_{m,k,a,b} \, \mathsf{W}_{k,r,s-a,t-b} \ , \end{aligned} \tag{3.6}$$

where again one must be careful to check the validity of the tensor indexing operations, i.e. $0 \leq a < H_{\text{out}}$, $0 \leq b < H_{\text{out}}$, $0 \leq s - a < F$, and $0 \leq t - b < F$.

One can note that Equation 3.4 is also a convolution, computed between the input tensor $\mathbf{Y}$ and the output tensor of error signals $\nabla\mathbf{Y}$ (known). Similarly, Equation 3.6 can be interpreted as a convolution between $\nabla\mathbf{Y}$ and the tensor $\mathbf{W}_{\text{flipped}}$, obtained by flipping all elements of the kernel tensor $\mathbf{W}$ across the spatial dimensions. To sum up, denoting the convolution operation with the symbol "$*$", the convolutional layer forward and backward passes can be written in the following compact form:

$$\boxed{\begin{aligned} \mathbf{Y} &= \mathbf{X} * \mathbf{W} && \text{(3.7a)} \\ \nabla\mathbf{W} &= \nabla\mathbf{Y} * \mathbf{X} && \text{(3.7b)} \\ \nabla\mathbf{X} &= \nabla\mathbf{Y} * \mathbf{W}_{\text{flipped}} \ , && \text{(3.7c)} \end{aligned}}$$

where the biases have been incorporated in the kernel tensor.

Convolution as a matrix multiplication

The convolution operation between two tensors $\mathbf{A}$ and $\mathbf{B}$ (Eq. 3.3 and Figure 3.2) essentially computes local dot products between sub-tensors of $\mathbf{A}$ and tensor $\mathbf{B}$. Efficient implementations of CNN libraries take advantage of this fact, reformulating both forward and backward pass operations of a convolutional layer as matrix multiplications. This can be done as follows (also see Figure 3.3 for a graphical representation):

1. First, each receptive field of size $K_{\text{in}} \times F \times F$ in the input tensor $\mathbf{X}$ is unrolled to a one-dimensional array of length $K_{\text{in}}F^2$. Since the number of receptive fields is H_{out}^2 (where H_{out} is given by Equation 3.2), placing all such arrays side by side as columns yields a $K_{\text{in}}F^2$-by-H_{out}^2 matrix. Let $\mathcal{X}$ denote this matrix of stretched receptive fields. This can be done for each example in the mini-batch, giving a set[2] of matrices $\{\mathcal{X}^{(m)}\}_{m=0,\ldots,M-1}$.
2. The kernel tensor $\mathbf{W}$ is similarly unrolled into a matrix, whose rows correspond to a stretched out filter. This results in a K_{out}-by-$K_{\text{in}}F^2$ matrix, denoted by $\mathcal{W}$.
3. Convolving the input tensor $\mathbf{X}$ with the kernel tensor $\mathbf{W}$ is now equivalent to multiplying the kernel matrix $\mathcal{W}$ and each receptive field matrix $\mathcal{X}^{(m)}$, and subsequently reshaping the resulting set of M matrices $\mathcal{Y}^{(m)}$ (each of size $K_{\text{out}} \times H_{\text{out}}^2$), into a tensor of size $M \times K_{\text{out}} \times H_{\text{out}} \times H_{\text{out}}$ (indeed the correct size of the output tensor $\mathbf{Y}$).
4. The backward pass becomes easier than using Equations 3.4-3.6: First, tensor $\nabla\mathbf{Y}$ must be reshaped into a set of M matrices of size $K_{\text{out}} \times H_{\text{out}}^2$, each denoted by $\nabla\mathcal{Y}^{(m)}$ (following the same notation). Multiplying matrices $\nabla\mathcal{Y}^{(m)}$ and $(\mathcal{X}^{(m)})^T$ yields the contribution of the m^{th} example to the gradient $\nabla\mathcal{W}$. The total gradient can be then computed by summing all these contributions.
5. Finally, multiplying matrix $\nabla\mathcal{W}^T$ and each matrix $\nabla\mathcal{Y}^{(m)}$ results in M matrices $\nabla\mathcal{X}^{(m)}$ of size $K_{\text{in}}F^2 \times H_{\text{out}}^2$, which must then be reshaped into a tensor of the same size as $\nabla\mathbf{X}$ (i.e. $M \times K_{\text{in}} \times H_{\text{in}} \times H_{\text{in}}$) using the inverse of the unrolling operation used in point 1.

To summarise, the operations described in Equations 3.7(a,b,c) can be replaced by the following ones:

$$\mathbf{X} \xrightarrow{unroll} \{\mathcal{X}^{(m)}\}_{m=0,\ldots,M-1} \tag{3.8a}$$
$$\mathcal{Y}^{(m)} = \mathcal{W}\mathcal{X}^{(m)} \quad \text{for} \quad m = 0, \ldots, M-1 \tag{3.8b}$$
$$\{\mathcal{Y}^{(m)}\}_{m=0,\ldots,M-1} \xrightarrow{reshape} \mathbf{Y} \tag{3.8c}$$
$$\nabla\mathcal{W} = \textstyle\sum_m \nabla\mathcal{Y}^{(m)}\ (\mathcal{X}^{(m)})^T \tag{3.8d}$$
$$\nabla\mathcal{X}^{(m)} = \nabla\mathcal{W}^T\ \nabla\mathcal{Y}^{(m)} \quad \text{for} \quad m = 0, \ldots, M-1 \tag{3.8e}$$
$$\{\nabla\mathcal{X}^{(m)}\}_{m=0,\ldots,M-1} \xrightarrow{roll-up} \nabla\mathbf{X} \tag{3.8f}$$

Note that the unrolling and rolling-up operations on the kernel tensor $\mathbf{W}$ are not included in the above equations, for the simple reason that in practice they need not be performed. Since the filters are never used in their tensor form, it is more convenient to store them directly in the matrix form $\mathcal{W}$.

The advantage of this approach lies in that implementing GPU-parallelised matrix multiplication routines, or using existing BLAS (Basic Linear Algebra Subprograms) libraries, is much easier than implementing convolutions efficiently. For this reason, in the Conv.NET

[2]The set of matrices $\{\mathcal{X}^{(m)}\}_{m=0,\ldots,M-1}$ can be conveniently stored in memory as one big matrix.

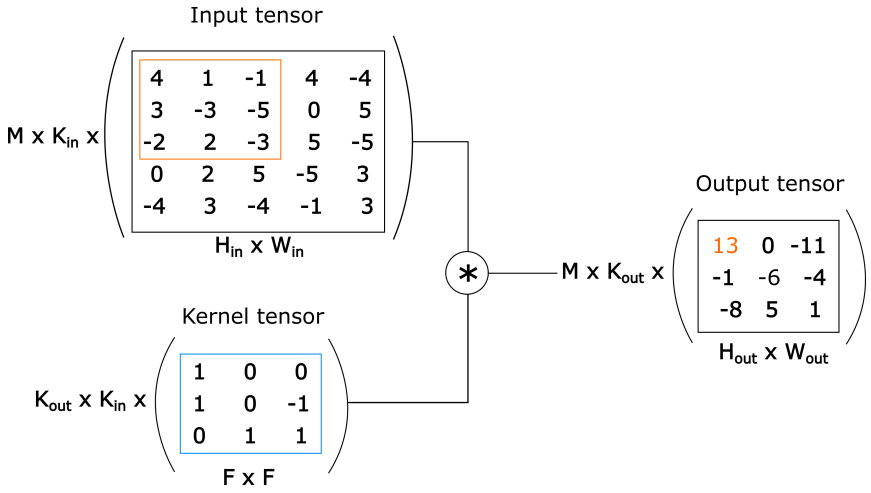


Figure 3.2: Graphical representation of the convolution operation between an input tensor (size $M \times K_{\text{in}} \times 5 \times 5$) and a kernel tensor (size $K_{\text{out}} \times K_{\text{in}} \times 3 \times 3$), stride of 1 and no padding.

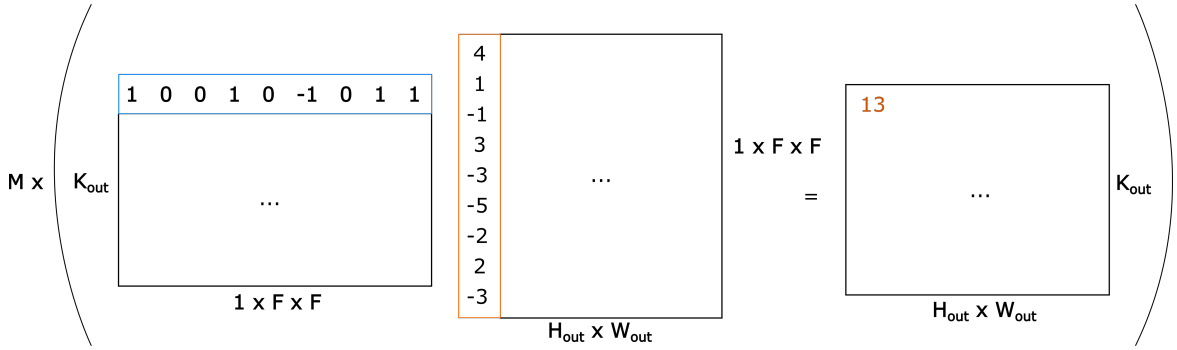


Figure 3.3: Reformulation of the convolution in Figure 3.2 as M matrix multiplications, where K_{out} has been assumed equal to 1, for better visualisation.

library all convolution operations are implemented as matrix multiplications. Although this requires several unrolling and reshaping operations, these are computationally very cheap and their impact is negligible compared to that of matrix multiplications [30].

The drawbacks of this approach is that in general it requires more memory, as each entry $\mathsf{X}_{m,l,i,j}$ appears in F^2 receptive fields, and thus each receptive field matrix $\mathcal{X}^{(m)}$ contains in principle F^2 copies of each entry of $\mathbf{X}$. Instead of creating and updating each $\mathcal{X}^{(m)}$ at each training iteration, the library creates a *mapping table* G_X from $\mathbf{X}$ to $\{\mathcal{X}^{(m)}\}$ when the layer is initialised. Then, instead of accessing each $\mathcal{X}^{(m)}$ (which effectively does not exist), the matrix multiplication subroutine directly accesses $\mathbf{X}$ through the mapping table G_X. The advantage of doing this is twofold. Firstly, it efficiently replaces both unrolling and rolling-up operations 3.8a and 3.8f. Secondly, it saves memory, as G_X is actually smaller than $\mathcal{X}$ by a factor M because the mapping is the same for every example in the mini-batch.

Zero-padding operations are sources of additional memory waste. In the Conv.NET library, the input tensor $\mathbf{X}$ is padded with zeros (when needed) into a new tensor $\mathbf{X}_{\text{pad}}$ before step 3.8(a), so that the above-mentioned mapping table G_X already takes padding into account. Part of this memory waste is recovered in the backward pass, by re-using the same memory buffer used for $\mathbf{X}_{\text{pad}}$ to store the tensor $\nabla\mathbf{X}_{\text{pad}}$, which is then un-padded into tensor $\nabla\mathbf{X}$. Furthermore, both padding and un-padding operations are performed by using a mapping table, created when the layer is initialised (similarly to the unrolling of receptive fields).

3.1.3 Nonlinearities

The output of a convolutional (or fully-connected) layer is usually passed through a nonlinear function f:

$$\mathsf{Y}_{m,k,a,b} = f(\mathsf{X}_{m,k,a,b}) \ .$$

Nonlinearity layers do not have parameters, and the backward pass simply consists in

$$\nabla\mathsf{X}_{m,k,a,b} = \nabla\mathsf{Y}_{m,k,a,b} \ f'(\mathsf{X}_{m,k,a,b}) \ .$$

Several options for f are available in the Conv.NET library, summarised in Table 3.1.

Activation function	Function form	Derivative	Conv.NET constructor
Hyperbolic tangent	$f(x) = \tanh(\beta x)$	$f'(x) = \beta(1 - (f(x))^2)$	`Tanh(double beta)`
Rectifier	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ 0 & \text{if} \quad x < 0 \end{cases}$	`ReLU()`
Exponential-rectifier	$f(x) = \begin{cases} x & \text{if} \quad x > 0 \\ \alpha(e^x - 1) & \text{if} \quad x \leq 0 \end{cases}$	$f'(x) = \begin{cases} 1 & \text{if} \quad x > 0 \\ \alpha e^x & \text{if} \quad x \leq 0 \end{cases}$	`ELU(double alpha)`

Table 3.1: Activation funtions available in the Conv.NET library

The hyperbolic tangent is a sigmoidal activation function, used since the earliest days of neural networks. Despite being still widely employed in shallow NNs, it is nowadays seldom used in the context of deep networks. In fact, a large input (both positive and negative) cause a sigmoidal unit to saturate, and thus to backpropagate a very small gradient signal, as the derivative $f'(x)$ is very close to zero in the saturation regime. This problem, known as *vanishing gradient*, can greatly slow down the training process in the case of very deep networks.

Replacing sigmoidal units with rectified linear units (ReLUs) has been empirically shown to accelerate the training of deep networks [1]. In fact, the rectifier function does not saturate for positive input, which may partially address the vanishing gradient problem. Furthermore, ReLUs are computationally very cheap, and promote sparse activations. For these reasons, they have been a very popular choice in recent years.

Finally, the exponential-rectifying function, introduced by Clevert *et al.* [31] in 2016, has also been shown to address the vanishing gradient problem, and to possess additional desirable properties. In fact, by saturating smoothly to -1 for negative values, it pushes the mean unit activation closer to zero, reducing the *bias shift effect* (similarly to batch normalisation, see Section 2.3) thereby speeding up the training process and acting as a regulariser.

3.1.4 Pooling layers

Pooling layers are used to downsample input spatial information, and they have been traditionally added between stages of convolutions. Each pooling unit in channel k of the output tensor computes a summary statistic of a patch of units in channel k of the input tensor. As a result, only spatial dimensions of the input tensor $\mathbf{X}$ are downsampled, whereas the depth is always preserved. The most important effect of pooling is that it makes the feature representation invariant to small translations and distortions, allowing to detect the presence of a feature regardless of its exact location in the input image. Two kinds of pooling layers are supported in the library, here briefly described.

Max-pooling

In this case, the downsampling consists in computing the maximum activation of a small patch of units in the input tensor. Mathematically:

$$\mathsf{Y}_{m,k,a,b} = \max\big(\{\mathsf{X}_{m,k,Sa+i,Sb+j}\}_{i=0,\dots,P-1,\ j=0,\dots,P-1}\big) \ ,$$

where P is the pooling width and S the stride. A common choice is performing 2×2 pooling with a stride of 2, so that each spatial dimension is halved. Currently, only $P = S = 2$ are supported in the Conv.NET library. In the backward pass, since each output activation only depends (linearly) on one input activation (i.e. the maximum one within the pooling patch), its corresponding gradient $\nabla\mathsf{Y}$ is simply redirected to that input location.

Global average pooling

This downsampling operation collapses an input tensor $\mathbf{X}$ of size $M \times K \times H \times H$ into an output tensor $\mathbf{Y}$ of size $M \times K \times 1 \times 1$, by taking the average of each feature map and collapsing spatial dimensions into a single unit. Mathematically:

$$\mathsf{Y}_{m,k,0,0} = \frac{1}{H^2} \sum_{i,j=0}^{H-1} \mathsf{X}_{m,k,i,j} \ .$$

The gradient can be simply backpropagated to the input as follows:

$$\nabla\mathsf{X}_{m,k,i,j} = \frac{1}{H^2} \nabla\mathsf{Y}_{m,k,0,0} \ ,$$

i.e. by distributing the error signal equally among the units of each input channel.

Global average pooling is a rather aggressive downsampling step, and it is sometimes performed at the end of all convolutional stages of a CNN and before the classifier (or sometimes even *instead* of the fully-connected layer preceeding the final softmax layer). This approach was popularized by Szegedy *et al.* [24], who employed global average pooling before the final 1000-way fully-connected layer with softmax activation in their GoogLeNet network winning ILSVRC 2014.

3.1.5 Fully-connected layer

Fully-connected layers are usually employed at the end of CNNs as classifiers. Their task is to process higher-level features learned by the underlying convolutional modules and learn to classify the input example based on such features.

In a fully-connected layer, every input unit is connected to every output unit, therefore the notion of spatial structure is lost and the multi-dimensional tensor notation is no longer meaningful. When a 4D tensor $\mathbf{X}$ is passed as input to a fully-connected layer, it is reshaped into a set of M one-dimensional arrays $\boldsymbol{x}^{(i)}$ containing all unrolled input activations corresponding to the i-th example of the mini-batch in $\mathbf{X}$ (see also Figure 2.1). The operations performed by the layer can then be written in terms of matrix multiplications. The forward pass takes the form

$$\boldsymbol{y}^{(i)} = W\ \boldsymbol{x}^{(i)} + \boldsymbol{b} \ , \qquad \text{for } i = 0, \dots, M-1 \ . \tag{3.9}$$

Matrix W and vector $\boldsymbol{b}$ above are respectively a N-by-$K_{\text{in}}H^2$ weight matrix and a N-by-1 vector of biases, where N is the number of output units (the only hyperparameter of the layer).

The gradients with respect to these parameters are computed as

$$\nabla W = \sum_i \nabla \boldsymbol{y}^{(i)} \, (\boldsymbol{x}^{(i)})^T \qquad \text{and} \qquad \nabla \boldsymbol{b} = \sum_i \nabla \boldsymbol{y}^{(i)} \; , \tag{3.10}$$

whereas the gradients with respect to each input vector $\boldsymbol{x}^{(i)}$ are computed as

$$\nabla \boldsymbol{x}^{(i)} = W^T \, \nabla \boldsymbol{y}^{(i)} \; , \qquad \text{for } i = 0, \ldots, M-1 \; . \tag{3.11}$$

In case of a classification problem with C classes, the network should end with a fully connected layer with $N = C$ output units, acting as the final classifier. Alternatively, a C-way fully-connected layer can be replaced by a convolutional layer with a filter size matching the spatial dimension of its input tensor and C feature maps. In fact, one can show that these two layers perform exactly the same operations, and can thus be used interchangeably.

3.1.6 Output layer: softmax

A feedforward neural network for classification into C classes typically ends with a *softmax* activation layer. Using the same notation as in the previous paragraph, let the input to this layer be a set of M one-dimensional vectors $\boldsymbol{x}^{(i)}$ of length C. Each entry $x_c^{(i)}$ of these vectors is then passed through the following squashing function:

$$y_c^{(i)} = \sigma(x_c^{(i)}) = \frac{\exp(x_c^{(i)})}{\sum_{c'=0}^{C-1} \exp(x_{c'}^{(i)})} \; .$$

Each output vector $\boldsymbol{y}^{(i)}$ then contains C values in $[0, 1]$ which add up to 1, and can be thus interpreted as a vector of probabilities over the C different classes. The index $c_{\max}^{(i)}$ corresponding to the maximum entry in $\boldsymbol{y}^{(i)}$ is the class inferred by the networks for the i-th image of the mini-batch.

3.2 Training a model

Training a neural network for classification means adjusting its internal parameters to increase its classification performance on the input data. The problem of training a neural network can thus be naturally formulated as an optimisation problem, given a suitable objective function.

3.2.1 Optimisation

Cross-entropy and mini-batch stochastic gradient descent

The classification performance of a deep NN can be measured in several ways, the most obvious of which is perhaps the classification accuracy, i.e. the number of correctly classified examples divided by the total number N of examples. Letting $\widetilde{c}^{(i)}$ denote the ground truth

label of example i, and $c_{\max}^{(i)}$ denote the class inferred by the network, the classification accuracy can be computed as $1 - E$, where E is the classification *error* of the network, i.e.

$$E = \frac{1}{N} \sum_{i=0}^{N-1} \begin{cases} 0 & \text{if } c_{\max}^{(i)} = \widetilde{c}^{(i)} \\ 1 & \text{otherwise} \end{cases} .$$

This performance measure, however, is unsuitable for gradient-based methods, as it is not differentiable. Furthermore, it is a binary measure: an inferred classification is either correct or incorrect, without any notion of *degree* of inference goodness. For these reasons, a differentiable cost or loss function is used instead.

The preferred cost function for classification problems is the *log-likelihood* or *cross-entropy* loss function, returning a smooth measure of the similarity between the output probability vector $\boldsymbol{y}^{(i)}$ (inferred by the network for the i-th example), and the one-hot probability vector $\boldsymbol{t}^{(i)}$ assigning probability 1 to the ground truth class $\widetilde{c}^{(i)}$, and 0 to all others. The cost function is as follows:

$$L = -\sum_{i=0}^{N-1} \sum_{c=0}^{C-1} t_c^{(i)} \log y_c^{(i)} = \sum_{i=0}^{N-1} \sum_{c=0}^{C-1} \begin{cases} \log y_c^{(i)} & \text{if } c = \widetilde{c}^{(i)} \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

A nice property of the cross-entropy cost function is that its gradient with respect to each input vector $\boldsymbol{x}^{(i)}$ of the *softmax* activation layer takes a very easy and interpretable form. Namely, for the c-th entry $\delta x_c^{(i)}$ of the j-th gradient vector, we have

$$\delta x_c^{(i)} \doteq \frac{\partial L}{\partial x_c^{(i)}} = \frac{\partial}{\partial x_c^{(i)}} \Big(\sum_{c=0}^{C-1} t_c^{(i)} \log y_c^{(i)} \Big) = \ldots = \begin{cases} y_c^{(i)} - 1 & \text{if } c = \widetilde{c} \\ y_c^{(i)} & \text{otherwise.} \end{cases} \tag{3.13}$$

This is a neat expression with an intuitive meaning: for each example i in the mini-batch, the derivative of L with respect to the c-th input activation of the softmax is simply equal to the error of the predicted probability for class c with respect to the c-th entry of the ground truth probability mass function (1 if c is the correct label, 0 otherwise).

By propagating these derivatives backwards into the network, we can compute the gradient of the loss function with respect to all parameters $\boldsymbol{\theta}^{(l)}$ of all layers $l = 1, \ldots, L-1$, and then update all such parameters in the direction of descending gradient:

$$\boldsymbol{\theta}^{(l)} \leftarrow \boldsymbol{\theta}^{(l)} - \eta \nabla \boldsymbol{\theta}^{(l)}. \tag{3.14}$$

This is the mini-batch version of Stochastic Gradient Descent (SGD). The hyperparameter η, known as the learning rate, controls the size of the steps taken in the direction specified by the gradients. Choosing a suitable value for η is of fundamental importance in training neural networks, because an excessively large learning rate can cause the loss function to diverge, whereas a very small learning rate yields a very slow learning.

Clearly, by using a mini-batch of size $M = 1$, mini-batch gradient descent reduces to the standard SGD (sometimes also called *online* update). Using a mini-batch size $M > 1$ has a twofold effect. First, it controls the degree of stochasticity in the update, as the gradient computed over a large mini-batch yields a better approximation of the "true" gradient (i.e. computed over the entire data set), whereas a small mini-batch leads to rather random steps in the loss function landscape. Second, it has a computational impact, because it corresponds to performing operations on large arrays less frequently, rather than more

frequent operations of smaller arrays, and therefore parallelised implementations can take advantage of massively parallel processing hardware (such as GPUs).

Typical values of the mini-batch size M are between one (*online* SGD update) and a few hundreds, with default values such as 32 or 64 usually working best. Using very large mini-batches, in fact, has the detrimental effect of requiring much more training iterations to reach the same error, since there are fewer parameter updates per epoch[3].

Parameter initialisation

Usually, parameters of a deep NN are initialised to small values, making sure to enforce *symmetry breaking* among units in the same layer [29]. In the Conv.NET library, parameters are initialised by following the practice introduced by He *et al.* [32]:

- The input weights to each unit in the network are initialised by sampling values from a Gaussian distribution with mean $\mu = 0$ and variance $\sigma^2 = 2/N_{\text{in}}$, where N_{in} is the number of inward connections of that unit. In the case of a fully-connected layers, this number equals the size of each input vector $\boldsymbol{x}^{(i)}$, whereas in the case of a convolutional layer N equals the size of the unit's receptive field, i.e. $K_{\text{in}}F^2$.
- All biases are initialised to zero.

Momentum update

Several variants of SGD have been proposed in the last few years, in the attempt to accelerate the optimisation process. The Conv.NET library implements one of such methods, called *momentum* update, which has been empirically shown to (almost always) yield a faster convergence rate than vanilla SGD [33]. The update rule is as follows:

$$\begin{aligned} \boldsymbol{v}^{(l)} &\leftarrow \mu \boldsymbol{v}^{(l)} - \eta \nabla \boldsymbol{\theta}^{(l)} \ , \\ \boldsymbol{\theta}^{(l)} &\leftarrow \boldsymbol{\theta}^{(l)} + \boldsymbol{v}^{(l)} \ . \end{aligned} \tag{3.15}$$

This update rule can be motivated by viewing the optimisation problem from a physical perspective and seeing the gradient $\nabla \boldsymbol{\theta}$ as the *acceleration* of the parameter update (rather than the speed, as in vanilla SGD). This acceleration is used to integrate the update *speed* $\boldsymbol{v}$, which, in turn, is used to integrate the *position* $\boldsymbol{\theta}$ of the point determined by the parameter configuration in the hilly landscape of the cost function. The hyperparameter μ, known as *momentum coefficient*, controls the amount of momentum that the point can build up while rolling on the landscape. Typical values are in $(0.5, 0.99)$. For $\mu = 0$, momentum update obviously reduces to the vanilla SGD update rule.

3.2.2 Regularisation

Usually, achieving a high classification performance on the training data is not sufficient, as in general there can be no guarantee that the same performance will be obtained with new, unseen data. Provided that the capacity of a model (i.e. the complexity of the functions that it can learn to approximate) is sufficiently high, near-perfect performance can be obtained on the training set, by learning very fine-grained idiosyncrasies of the data. However, such idiosyncrasies may be (likely) due to noise and may not reflect the

[3] A training epoch is defined as the number of iterations needed to use each example in the training set exactly once.

true underlying structure of the data, thereby leading to very poor predictive performance, once learned. This is the problem of *overfitting* as opposed to *generalisation*, perhaps the most fundamental concept in machine learning.

Deep neural networks usually have very high capacity and are therefore particularly prone to overfitting. Several methods, collectively known as *regularisation* methods, can be used to prevent overfitting when training high-capacity models.

Early stopping

One simple way of preventing overfitting is monitoring the network's classification performance (namely the cross-entropy loss) on a validation data set. A validation set is a set of examples which are neither used for training nor for the final evaluation of the model, and are considered to be representative of future test examples.

In the Conv.NET library, the state of the network, with all its weights, is saved to a binary file[4] every time the loss function, evaluated on the validation set, decreases with respect to the previous evaluation. When the cost function on the validation data stops decreasing, or even start *increasing*, overfitting is likely to be occurring, and further optimisation on the training set may be detrimental in terms of predictive performance.

Due to the stochasticity of the training process, however, the validation loss can remain almost constant or increase slightly for a few epochs before starting to decrease again. Therefore, it is usually a good idea not to interrupt the training as soon as the validation loss stops decreasing, but rather allowing it to continue for a certain number of epochs (denoted by *patience* in the library).

L2 penalty

Another common regularisation method available in the Conv.NET library is L2 penalty regularisation. The idea behind this method is to limit the capacity of the model by penalising large synaptic weights. This is achieved by adding to the loss function a penalty term consisting of the sum of the squared L2 norm of each weight vector in the network. Denoting the cross-entropy loss function (Eq. 3.12) by L_0, the L2 regularised loss function takes the following form:

$$L = L_0 + \frac{\lambda}{2} \sum_k w_k^2 \ , \tag{3.16}$$

where the sum is over all weights in all layers of the network, and λ is a hyperparameter. In the parameter update rule, the L2 penalty term translates into a linear term, as follows:

$$w_k \to w_k - \eta \Big(\frac{\partial L_0}{\partial w_k} + \lambda w_k \Big) \ ,$$

i.e. each weight decays towards zero with rate $\eta\lambda$, and for this reason L2 penalty is also referred to as *weight decay*. It can be shown that L2 regularisation is equivalent to placing a Gaussian prior $\mathcal{N}(0, 1/\lambda)$ on all weights of the network.

[4]The model can then be reloaded later, in order to resume training or evaluate it on different data. Furthermore, the number of training epochs between two consecutive evaluations can be specified by the user, as shown in Listing 3.1.

Model averaging

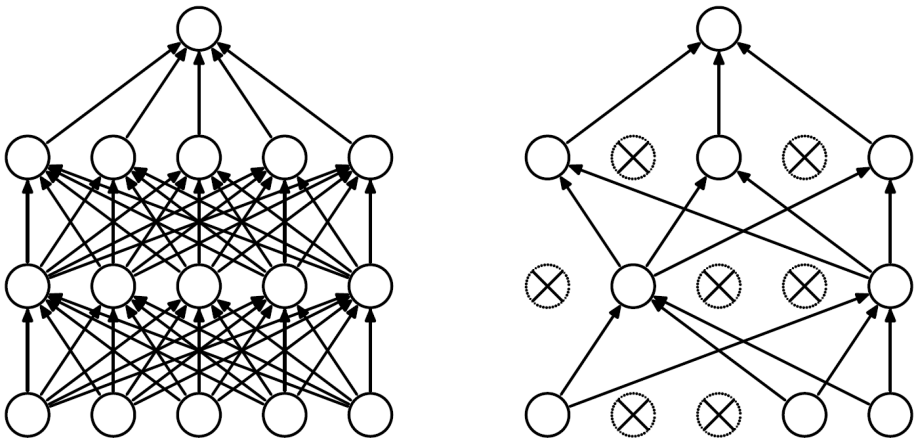
Model averaging is a general technique in machine learning that consists in training several models separately, either on the same data set or on different parts of the same data set, and then averaging their predictions at test time. The idea behind this method is that different models usually make slightly different errors (i.e. they overfit the training data in slightly different ways), and thus the predictive performance of their *ensemble* is usually superior to that of each individual model.

This technique works really well with neural networks. It is immediate to show that in the ideal case of uncorrelated errors, the expected squared error of an ensemble of k neural networks is smaller than the *average* error of the k models by a factor $1/k$ [12], although errors are never perfectly uncorreated in reality. The obvious drawback of model ensembling is that it can be very demanding in terms of memory and computation. This applies to both the training and, more importantly, to the inference process, where often runtime performance is crucial.

Dropout

Dropout is a recently published regularisation method (Srivastava *et al.*, 2014 [7]) that attempts to emulate model averaging with very large ensembles, with minimal computational impact. This is achieved by de-activating (*dropping out*) each unit in a hidden layer of a network at each training iteration, with some probability p (usually 0.5), as illustrated in Figure 3.4. This simple procedure is equivalent to training an exponentially large ensemble of thinned sub-networks (namely 2^{N_H} models, where N_H is the number of hidden units). At test time, dropout is not applied, so that its predictions approximate the averaged predictions of the ensemble of its sub-networks.

Dropout is not exactly equivalent to model averaging, as thinned sub-networks share parameters and as such are not independent. Moreover, dropout generally slows down the training process, as a large fraction (typically 1/2) of the parameters are effectively frozen at each itaration. However, dropout is still computationally much cheaper than training several models to average over, and it can be applied to virtually any type of model and with any training algorithm.



(a) Standard Neural Net (b) After applying dropout.

Figure 3.4: Graphical representation of dropout, from [7]. Reproduced with permission.

4

Methods

4.1 The GTSRB data set

The German Traffic Sign Recognition Benchmark (GTSRB) is a publicly available data set containing 51839 images of German road signs, divided into 43 classes [4]. A representative image for each class is shown in Figure 4.1. The data set was published during a competition held at the 2011 International Joint Conference on Neural Networks (IJCNN).

Images in the data set exhibit wide variations in terms of shape and colour among some classes, as well as strong similarities among others (e.g. different speed limit signs). The data pose several challenges to classification, including varying lighting and weather conditions, motion-blur, viewpoint variations, partial occlusions, physical damage and other real-world variabilities (some samples considered difficult to classify are shown in Figure 4.2). Furthermore, resolution is not homogeneous and as low as 15×15 pixels for some images (Figure 4.3, left panel). For these reasons, human performance on this data set is not perfect, and estimated at around 98.84% on average [4].

An additional challenge in training classification algorithms on the GTSRB data is that the 43 classes are not equally represented. The relative frequency of classes 0, 19, and 37, for example, is only 0.56%, significantly lower than the mean $1/43 = 2.33\%$ (Figure 4.3, right panel). Moreover, since the data set was created by extracting images from video footage, it contains a *track* of 30 images for each unique physical real-world traffic sign *instance*. As a result, images from the same track are heavily correlated.

Figure 4.1: A representative image for each of the 43 classes in the GTSRB data set. Reproduced with permission from Stallkamp *et al.* [4].

Figure 4.2: Some difficult samples in the GTSRB data.

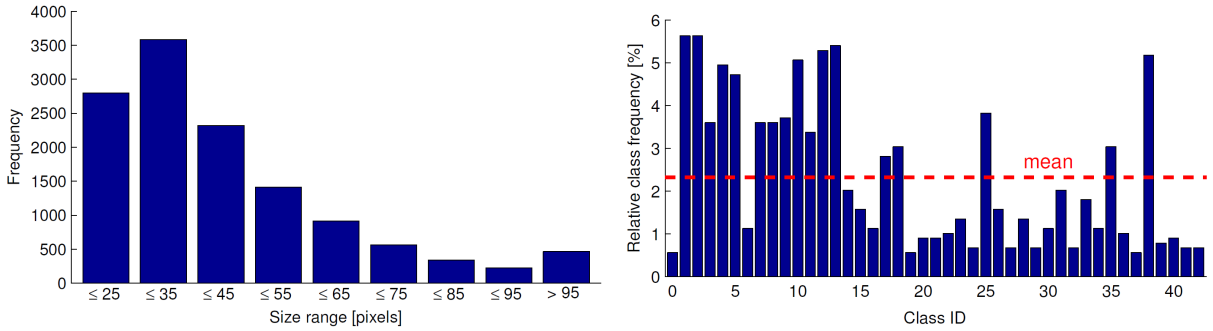


Figure 4.3: Left panel: distribution of the resolution of GTSRB test images. Right panel: relative class frequency in the training data. Both images reproduced with permission from Stallkamp *et al.* [4]

4.1.1 Data preprocessing

In accordance with the idea behind representation learning, GTSRB images were minimally preprocessed, and used to train CNNs almost in their raw form. The preprocessing steps descrbed in this section follow the approach used by Sermanet and LeCun [5] as closely as possible, for a better comparison.

First of all, a validation set was extracted from the training set by randomly picking one track (30 images) for each class, for a total of 1290 images. The remaining training set thus contains 37919 images[1]. This step was repeated twice, in order to perform holdout (2-fold) cross-validation (although with data sets of different size).

In each image, the provided region of interest (ROI) containing the traffic sign was cropped and then rescaled to 32×32 pixels. The resulting images were then either converted to greyscale (GS), following Sermanet and LeCun [5], or kept as raw RGB images.

Finally, each feature (i.e. pixel) of each image in the training set was normalised to zero mean and unit variance across the data set. Images were *not* normalised individually (across the image), but only across the data set. The same normalisation used on the training data was then applied to the validation and test data.

At the end of this procedure, we obtain 4 data sets, henceforth denoted by GS1, GS2, RGB1, RGB2 (with obvious notation).

[1]One image is missing from a track, in the original data

4.2 Model selection

4.2.1 CNN architectures

Two CNN architectures were designed, trained and compared in this work. The baseline model (left column) is a classic CNN architecture, whose core structure is essentially the same as that of the LeNet architectures introduced in the late 1980s by LeCun *et al.* [20]. Henceforth, this model will be referred to as the LeNet-like model, or simply LeNet. The network has 10 hidden layers, 6 of which contain parameters, for a total of about 822k $\sim$ 827k parameters (depending on whether GS or RGB images are used). The architectural details (number of feature maps, hidden units, etc.), outlined in Table 4.1, reflect those of the best model in the 2011 paper by Sermanet and LeCun [5] (see Table 2.1 in Chapter 2), apart from the use of skip connections.

Layer	LeNet (baseline) model	Tensor size	Parameters
0	Input (GS or RGB)	1 or $3 \times 32 \times 32$	-
1	Conv (5×5, 108 maps)	$108 \times 28 \times 28$	2.8k (GS) or 8.2k (RGB)
2	Nonlinearity	$108 \times 28 \times 28$	-
3	Max-pooling (2×2, stride 2)	$108 \times 14 \times 14$	-
4	Conv (5×5, 108 maps)	$108 \times 10 \times 10$	534.7k
5	Nonlinearity	$108 \times 10 \times 10$	-
6	Max-pooling (2×2, stride 2)	$108 \times 5 \times 5$	-
7	Fully-conn. (100 units)	$100 \times 1 \times 1$	270.1k
8	Nonlinearity	$100 \times 1 \times 1$	-
9	Fully-conn. (100 units)	$100 \times 1 \times 1$	10.1k
10	Nonlinearity	$100 \times 1 \times 1$	-
11	Fully-conn. (43 units) + softmax	$43 \times 1 \times 1$	4.3k
		Total parameters:	822k (GS) / 827k (RGB)

Table 4.1: Detailed architecture of the LeNet (baseline) model. The tensor size does not account for the mini-batch size, independent on the architecture. Unless stated otherwise, stride is 1 and padding 0.

This baseline model was then compared to a different architecture, inspired by the VGGNet by Simonyan and Zisserman [2] and thus referred to as VGGNet-like, or simply VGGNet. There are several important differences between this second model, described in detail in Table 4.2, and the baseline LeNet architecture. First, the VGGNet is a much deeper model, with 19 hidden layers, 9 of which contain parameters.

Notably, however, it has 461k $\sim$ 462k parameters in total, i.e. significantly fewer than the LeNet network, due to the use of smaller (3×3) filters in all convolutional layers. The combination of small filters with stacks of convolutional layers (without subsampling layers in between) is in fact one of the peculiarities of VGG-inspired architectures, enabling larger *effective* receptive fields with the use of fewer parameters. Another difference is that the spatial dimensions of the tensor of activations are preserved by convolutions (with the use of zero-padding), and only reduced by pooling, while the number of feature maps is doubled at the same time.

Layer	VGGNet model	Tensor size	Parameters
0	Input (GS or RGB)	1 or $3 \times 32 \times 32$	-
1	Conv (3×3, 32 maps, padding 1)	$32 \times 32 \times 32$	0.3k (GS) or 0.9k (RGB)
2	Nonlinearity	$32 \times 32 \times 32$	-
3	Conv (3×3, 32 maps, padding 1)	$32 \times 32 \times 32$	9.2k
4	Nonlinearity	$32 \times 32 \times 32$	-
5	Max-pooling (2×2, stride 2)	$32 \times 16 \times 16$	-
6	Conv (3×3, 64 maps, padding 1)	$64 \times 16 \times 16$	18.5k
7	Nonlinearity	$64 \times 16 \times 16$	-
8	Conv (3×3, 64 maps, padding 1)	$64 \times 16 \times 16$	36.9k
9	Nonlinearity	$64 \times 16 \times 16$	-
10	Max-pooling (2×2, stride 2)	$64 \times 8 \times 8$	-
11	Conv (3×3, 128 maps, padding 1)	$128 \times 8 \times 8$	73.9k
12	Nonlinearity	$128 \times 8 \times 8$	-
13	Conv (3×3, 128 maps, padding 1)	$128 \times 8 \times 8$	148k
14	Nonlinearity	$128 \times 8 \times 8$	-
15	Max-pooling (2×2, stride 2)	$128 \times 4 \times 4$	-
16	Fully-conn. (100 units)	$100 \times 1 \times 1$	160k
17	Nonlinearity	$100 \times 1 \times 1$	-
18	Fully-conn. (100 units)	$100 \times 1 \times 1$	10.1k
19	Nonlinearity	$100 \times 1 \times 1$	-
20	Fully-conn. (43 units) + softmax	$43 \times 1 \times 1$	4.3k
		Total parameters:	461k (GS) / 462k (RGB)

Table 4.2: Detailed architecture of the VGGNet model. The tensor size does not account for the mini-batch size, independent on the architecture. Unless stated otherwise, stride is 1 and padding 0.

	LeNet model (parameters GS/RGB)	VGGNet model (parameters GS/RGB)
Feature extractor	537k / 542k	287k / 288k
Classifier	285k	174k

Table 4.3: Comparison between the two architectures in terms of the number of parameters in the feature-extraction stage and in the classification stage.

4.2.2 Choice of activation function

In Tables 4.1 and 4.2 the particular type of nonlinear activation function used after convolutional and fully-connected layers is not specified. Although the rectifying-linear function (ReLU) has been the preferred choice in the past few years, the effect of replacing it with the recently introduced exponential-linear function (ELU) in terms of training speed was investigated by evaluating the cross-entropy cost after one training epoch. Results are shown in Section 5.1.

5
Results and discussion

5.1 Hyperparameter selection

Training neural networks, and in particular deep NNs, requires setting several hyperparameters that affect both the optimisation and the regularisation aspect of the training process.

5.1.1 Optimisation

The hyperparameters affecting optimisation are the learning rate η, the momentum coefficient μ, and the mini-batch size M. The standard value of $\mu = 0.9$ has been repeatedly observed to work well in practice [1, 2, 24], and was thus kept fixed throughout all experiments.

The effects of the mini-batch size M have been discussed in Section 3.2.1. This hyperparameter was chosen with the aim of obtaining the best computational efficiency, by measuring the number of images processed per second during training of the LeNet architecture for different values of M. Specifically, a grid search on the logarithmic scale between $M = 1$ and $M = 128$ was carried out, using different devices supporting OpenCL™, as shown in Figure 5.1. Using a NVIDIA® GeForce GTX 980Ti graphics card (the device to be used for training), the computational efficiency was observed to increase steadily as a function of M, but saturated for $M > 64$. Therefore, the value $M = 64$ was chosen and kept fixed throughout all training runs.

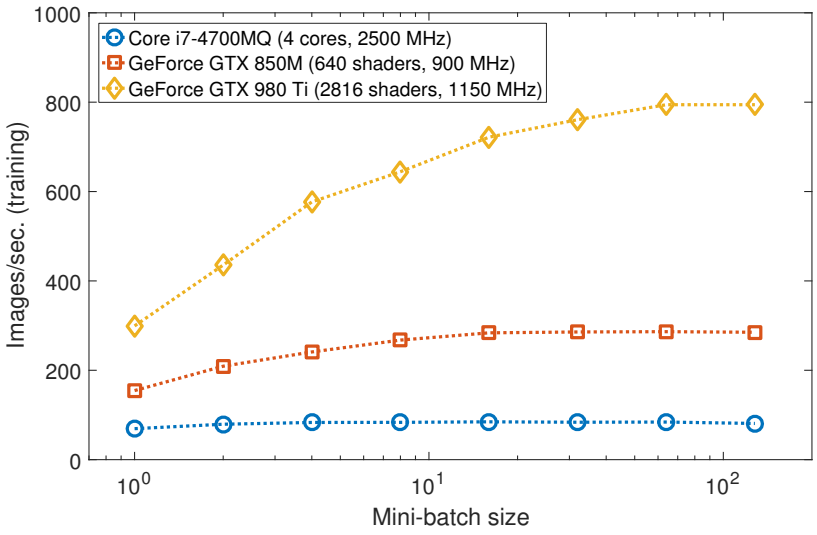
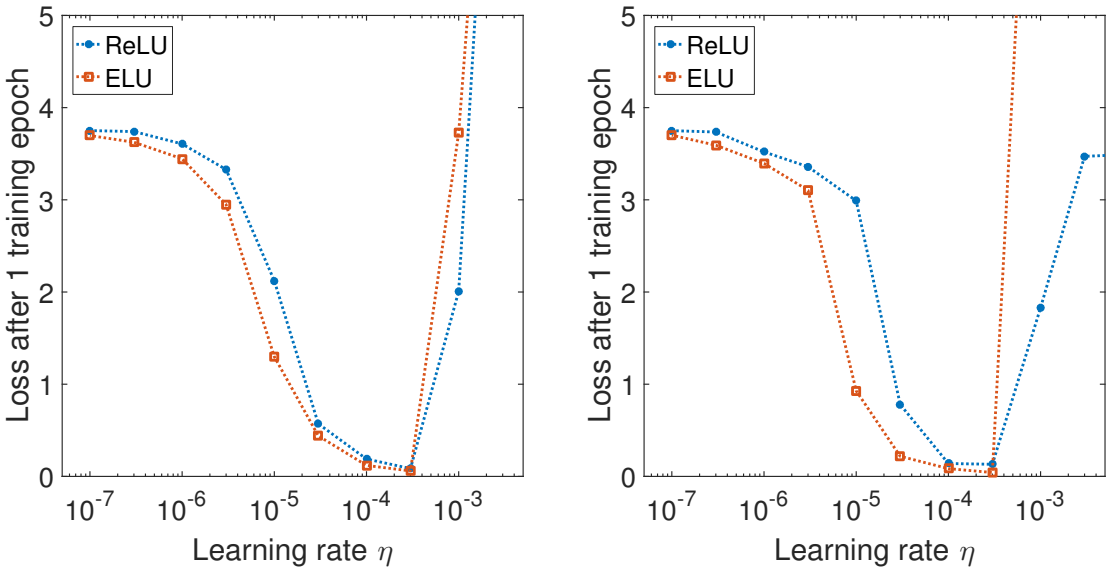


Figure 5.1: Images processed per second during training as a function of the mini-batch size M. Different curves correspond to different devices supporting OpenCL™.

It is important to fix the mini-batch size before optimising the learning rate η, as these two parameters are strongly interdependent. Let $\|\nabla_{\boldsymbol{\theta}^{(l)}} L\|$ be the norm of the gradient with respect to the parameters in a given layer l, when using a mini-batch size of M. If the mini-batch size is doubled, then the expected norm of the same gradient would be $2\|\nabla_{\boldsymbol{\theta}^{(l)}} L\|$ (see Eq. 3.1), and we would need to reduce the learning rate by a half in order to make update steps of the same length.

Having fixed $M = 64$, the learning rate space was explored by carrying out a grid search in the logarithmic scale between $\eta = 10^{-7}$ and $\eta = 10^{-2}$. In Figure 5.2, the value of the cost function after one training epoch is plotted as a function of the learning rate, for both the LeNet model (left panel) and the VGGNet model (right panel). These experiments were repeated using both ReLUs (blue dots) and ELUs (red squares) throughout the architecture, for both models.



(a) LeNet model **(b)** VGGNet model

Figure 5.2: Cost function after one training epoch as a function of the learning rate η, with ReLUs (blue dots) or ELUs (red squares). **(a)**: LeNet model. **(b)**: VGGNet model.

In both plots, three regimes can be clearly distinguished. For very small learning rates ($\eta \lesssim 10^{-5}$), loss decrease is minimal and learning is expected to be very slow. For intermediate values ($10^{-5} \lesssim \eta \lesssim 3 \cdot 10^{-4}$), learning speed accelerates sharply. Finally, for $\eta \gtrsim 3 \cdot 10^{-4}$, the gradient descent method becomes unstable, until the cost function eventually diverges in just one training epoch. This last regime, for obvious reasons, has to be avoided, although it has been suggested [33] that the optimal η may lie close to the divergence point. In practice, if learning speed is not a major problem, it is usually better to select a learning rate yielding a training speed that is neither *too* slow, nor *too* fast. This choice reduces the importance of designing a sophisticated strategy for *annealing* the learning rate, and essentially allows to keep η fixed (although fine-tuning the network with a smaller η can sometimes lead to slight improvements). For these reasons, a learning rate of $\eta = 10^{-5}$ was selected and kept fixed for both architectures.

Another aspect worth noting in Figure 5.2 is that ELUs consistently outperform ReLUs in terms of learning speed, in both the slow and fast learning regimes. In other experiments (here not displayed) using ReLUs, units were found to learn to always output zero (i.e. never activate) after only a few training epochs. This was indeed found to be a known

problem (the "dying ReLUs" problem), usually caused by a large magnitude gradient flowing backwards through the ReLU and producing a large negative bias shift. After this has happened, the input to the unit is likely to be negative, and as a result the ReLU seldom activates in the forward pass. Since the gradient of an inactive ReLU is zero, it is very unlikely that the unit will be able to update its input parameters and thus recover. This problem does not affect ELUs, as their gradient is non-zero for negative input, and the units can thus slowly recover with time, if needed. For these two reasons, the exponential-linear activation function was chosen as the nonlinearity to use in both models, and all results henceforth showed are obtained using ELUs.

5.1.2 Regularisation

Regularisation methods implemented in the Conv.NET library have been discussed in Section 3.2. A combination of methods was used to regularise the two models under investigation. In particular, early stopping was always used. When using this method, one does not have to worry about setting the right number of training epochs, as training will stop if overfitting starts (after the *patience* buffer has run out), and it can always be resumed if needed. For all runs in the following sections, a patience of 25 epochs was used.

Choosing a good value for the L2 penalty strength λ can be more tricky. After carrying out a grid search over the interval $(10^{-6}, 10^{-3})$ (albeit with no cross-validation), the value $\lambda = 10^{-5}$ was found to yield the best performance on the validation set, for the LeNet model. To allow a fair comparison, the same value of λ was then used for the VGGNet architecture as well.

Finally, dropout probability was set to the recommended value of 0.5, whose optimality in almost all cases is both empirically and theoretically supported [7, 12]. In the results presented in this chapter, dropout is only applied to the fully-connected layers of the CNNs, following the most common practice.

5.2 Training the networks

As discussed in Section 3.2.2, when training a neural network it is important to monitor the cost function evaluated on both the training and the evaluation data. Figure 5.3a shows a clear example of overfitting: While the training cost (red diamonds) smoothly decreases, the validation cost (blue circles) stops decreasing after $20 \sim 25$ epochs, and eventually starts increasing. Monitoring the classification accuracy (or, equivalently, the error, shown in Figure 5.3b), can also be useful, although early stopping should be based on the cost function, as it is the objective function that the gradient descent method is directly minimising. In Figure 5.3b, the LeNet architecture was trained on data set GS1 with no regularisation, which causes it to overfit the training set.

5.2.1 Comparison of regularisation methods

The two models were trained using the different regularisation methods described in Section 3.2.2 and data sets GS1 and GS2, as Sermanet and LeCun [5] reported better results when converting GTSRB images to greyscale. The cross-entropy cost evaluated on the validation set when training the LeNet model with different regularisation methods is shown in Figure 5.4. The same runs were repeated using the VGGNet model, and the resulting

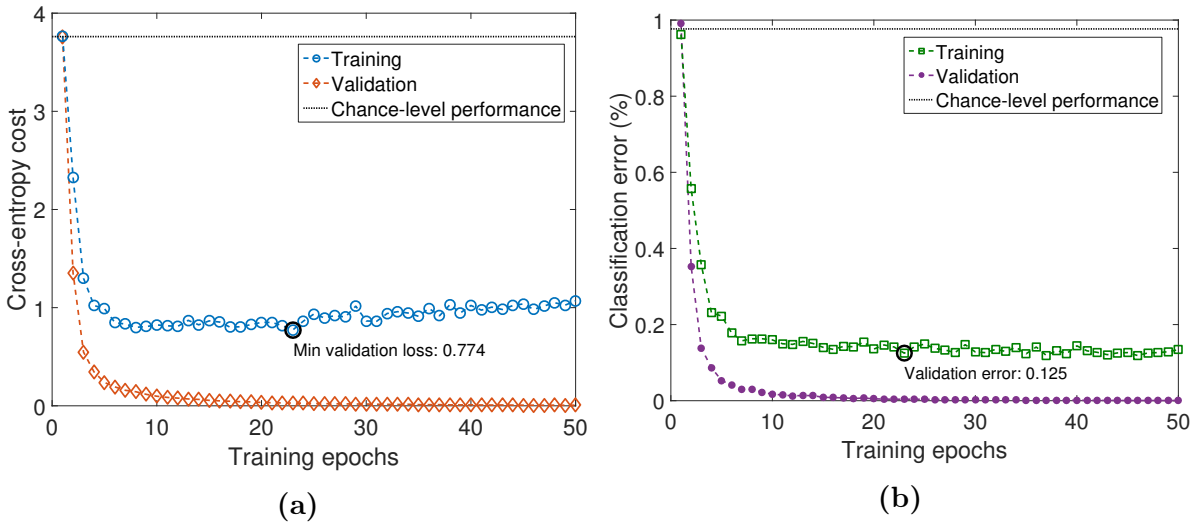


Figure 5.3: A training run of the LeNet model on data set GS1. Hyperparameters: $\eta = 10^{-5}, \mu = 0.9, M = 64$, no regularisation. **(a)**: Cross-entropy cost function. **(b)**: Classification error. The black dotted lines represent chance-level performance.

curves are shown in Figure 5.5. All training runs shown here were obtained using data set GS1 (curves obtained with data set GS2 are very similar, and therefore not shown). In both plots, the y-axis is in logarithmic scale for better visualisation.

These curves allow a first qualitative analysis of the effect of the two different regularisers. In particular, both models exhibited strong overfitting without regularisation (blue circles). This problem was partially addressed by using L2 penalty (red curves), which appeared to be slightly more effective in the case of the VGGNet model. Dropout, however, had a much greater impact (green crosses) for both architectures. With dropout, the networks learned more slowly, since parameters in the fully-connected layers are updated less frequently (by a factor of 2, on average) and parameters in the convolutional layers receive smaller gradients. However, both networks reached significantly lower validation cost after just ∼20 training epochs, and this keept decreasing for much longer[1]. As shown in the next section, this translated into a much better generalisation performance. Interestingly, the two regularisation methods did not appear to work well together (purple curves), compared to the case where only dropout was used, particularly in the case of the deeper model.

It is worth noting that the amplitude of the oscillations in the curves appears amplified in these plots because of the use of logarithmic scale on the y-axis. The curve with blue circles in Figure 5.4, for example, looks more noisy than that in Figure 5.3a, although they are actually the same curve. By taking the scale into account, it can be noted that the cost function exhibited significantly smaller oscillations when using dropout.

[1]Here only the first 100 epochs are displayed, for clarity, but networks were trained for at least 200 epochs, when using dropout.

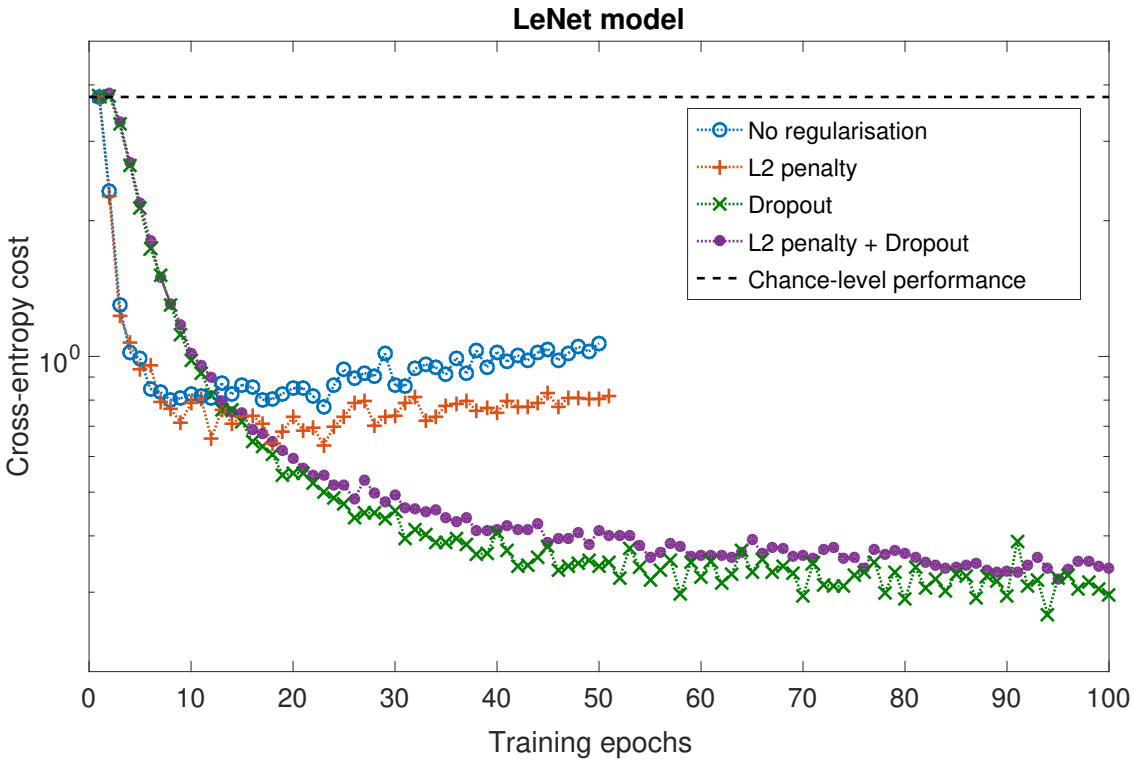


Figure 5.4: Cost function on the validation set when training the LeNet model with different regularisation methods and data set GS1. The cost on the training set is not shown. Hyperparameters used: $\eta = 10^{-5}$, $\mu = 0.9$, $M = 64$.

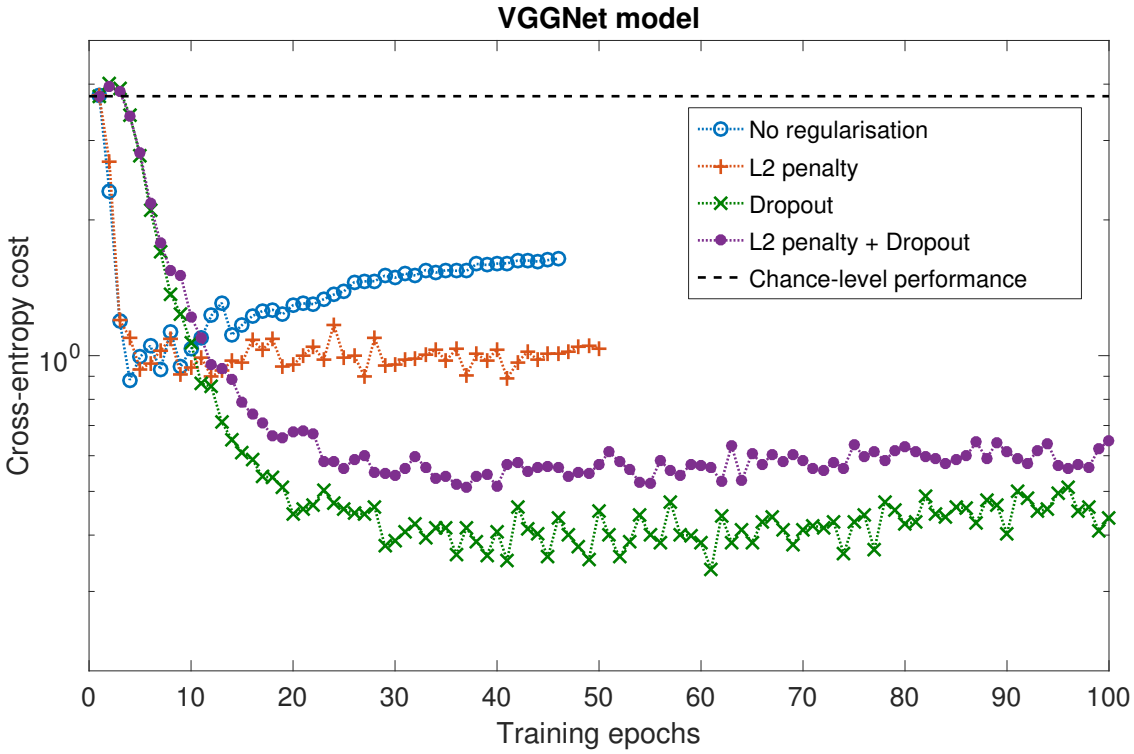


Figure 5.5: Cost function on the validation set when training the VGGNet model with different regularisation methods and data set GS1. The cost on the training set is not shown. Hyperparameters used: $\eta = 10^{-5}$, $\mu = 0.9$, $M = 64$.

5.3 Generalisation performance

In order to assess the predictive performance of a classification algorithm, it must be evaluated on a test set, i.e. a separate data set containing examples that have neither been used for training the algorithm, nor for choosing hyperparameters, nor for determining when to stop training. After training the LeNet and the VGGNet models with different regularisation methods, the networks were finally evaluated on the official GTSRB test set, obtaining the classification accuracies displayed in Table 5.1.

	LeNet	**VGGNet**
No regularisation	93.2 ± 0.5	90.4 ± 1.5
L2 penalty	92.6 ± 0.1	91.7 ± 0.1
Dropout	$\mathbf{95.7 \pm 0.1}$	$\mathbf{94.8 \pm 0.4}$
L2 penalty + Dropout	94.8 ± 0.3	93.3 ± 0.3

Table 5.1: Classification accuracy (in %) on the test greyscale data for the two models trained with different regularisation methods. Displayed values are 2-fold averages (data sets GS1 and GS2).

Several remarks can be made about these results. First, as was qualitatively observed in the previous section, dropout outperformed L2 penalty in terms of model regularisation, and the best accuracy for both architectures was obtained by only using dropout, without weight decay. Only the VGGNet architecture was observed to benefit from L2 penalty, whereas the LeNet classification performance on test data decreased slightly when this regulariser was used. The reason for this behaviour is not entirely clear, and probably a more robust strategy for selecting the hyperparameter λ, such as k-fold cross-validation carried out for both architectures, would shed light on this phenomenon. The same argument applies as to why the two regularisation methods do not appear to work well in synergy. It has been recently argued [33] that L2 regularisation could be dropped altogether when using early stopping, as they essentially play the same role, with the latter allowing a much easier hyperparameter selection. Since early stopping was always used in this work, our results seem to confirm the validity of this recommendation.

In general, regularisation appears to be much more important for the VGGNet than for the LeNet model. The former, in fact, performed significantly worse than the latter without regularisation (-2.8% accuracy on average), but the performance gap shrinked when dropout was used (-0.9% on average). This result was rather surprising. Since the LeNet contains almost 80% more parameters than the VGGNet, one would expect its capacity to be larger, making it more prone to overfitting. However, the obtained results suggest that the opposite may be true.

The natural explanation of this phenomenon is that the capacity of a CNN does not only depend on the number of its parameters, but also on its *depth*. Although the final stage of the two models is exactly the same (two hidden fully-connected layers with 100 units each, followed by a 43-way fully-connected layer with softmax activation), the feature extraction stage of the VGGNet is more than twice as deep (15 layers, including 6 nonlinearities) as that of the LeNet model (6 layers, 2 of which nonlinearities). In light of the above, it appears plausible that regularisation is more important for the VGGNet architecture.

5.3.1 Using colour information

In order to find strategies for improving the accuracy of a classification algorithm, it can be useful to go back to the original data and understand the characteristics of the examples that the algorithm is classifying incorrectly. This was done for the best performing networks (i.e. those trained with dropout only) and revealed that a large fraction of the misclassified images were indeed very hard to classify even for humans, mostly due to extremely poor resolution. However, some misclassified examples exhibiting motion-blur or partial occlusions were instead easily recognisable to the human eye.

This motivated us to re-train the networks using colour (RGB) images instead of greyscale (GS) images, in the hope that, when properly regularised, they would learn how to effectively use the additional information contained in the three colour channels, thereby improving their classification performance. Both the LeNet and the VGGNet models were then re-trained using data sets RGB1 and RGB2, with dropout applied to fully-connected layers and no L2 penalty. The cross-entropy cost curves obtained when training the two architectures are shown in Figures 5.6a and 5.6b, respectively, whereas the classification accuracy evaluated on the test set in the different cases is displayed in Table 5.2.

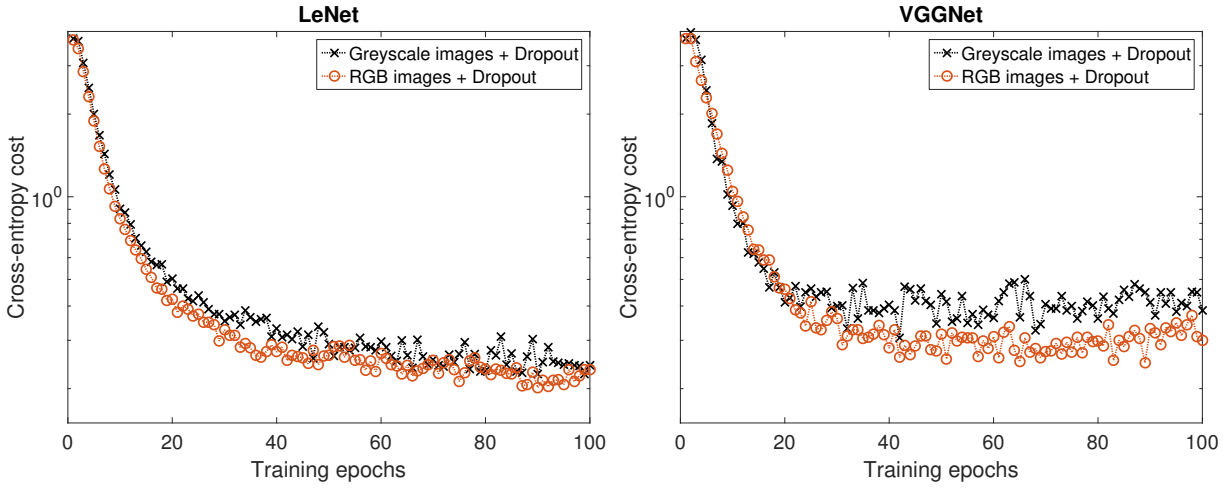


Figure 5.6: Cross-entropy validation cost during training on either GS or RGB images. **(a)**: LeNet model. **(b)**: VGGNet model. Hyperparameters: $\eta = 10^{-5}, \mu = 0.9, M = 64$ in both cases. Only the first 100 training epochs are shown (200 in total).

Interestingly, using colour information yielded a better performance for both architectures. A random selection of 16 images that were correctly classified by the LeNet model trained with colour images, but misclassified when ignoring colour information, is shown in Figure 5.7a. Qualitatively, it appears that when colour information was made available, the CNN learned to exploit the presence of peculiar features such as the yellow blob in the middle of "priority road" traffic signs and the coloured circles in the "traffic lights ahead" danger signs.

This result is in contrast to that of Sermanet and LeCun [5], who reported better classification accuracies when training their models with greyscale images, although not consistently. It is worth noting that Sermanet and LeCun converted images to the YUV colour scheme, whereas in this work the raw RGB images were used. Furthermore, they connected different filters in the first convolutional layer to different channels (namely 100

	LeNet	**VGGNet**
Greyscale + dropout	95.7 ± 0.1	94.8 ± 0.4
RGB + dropout	96.2 ± 0.1	95.7 ± 0.5
Ensemble of 2 CNNs (GS + RGB)	96.6 ± 0.1	96.2 ± 0.1
Ensemble of 4 CNNs (GS + RGB, LeNet + VGGNet)	**96.9 ± 0.2**	

Table 5.2: First and second rows: Classification accuracy (in %) on the test data for the two models trained with dropout and either GS or RGB images. Third row: Ensembles of two models with the same architectures, one trained with GS images and the other with RGB images. Fourth row: Ensemble of four models, combining GS and RGB images and the two architectures. All values are 2-fold averages.

to the Y channel, and 8 to the U and V colour channels), whereas here all filters were connected to all channels, according to current recommendations [12]. Finally, while in this work no form of data augmentation was used to regularise the networks, Sermanet and LeCun made heavy use of synthetic data, obtained by adding various forms of perturbations including translation, rescaling, and rotation of the original images.

(a) **(b)**

Figure 5.7: **(a)**: A sample of 16 GTSRB test images randomly selected among those correctly classified by the LeNet model trained with RGB images, but misclassified by the same network trained with GS images. **(b)**: A random selection of 16 images incorrectly classified by an ensemble of two LeNet CNNs trained on both GS and RGB images.

5.3.2 Model ensembles

Finally, the classification accuracy of ensembles of CNNs was evaluated. As shown in the third row of Table 5.2, the performance of the ensemble of two networks with the same architecture, trained with either greyscale or colour images, was found to be superior to that of individual networks, regardless of the architecture. This result is not surprising, as it confirms the effectiveness of model averaging as a regularisation method [12]. However, it is

interesting to note that the performance gap between the two architectures reduced further (if compared to the results in Table 5.1), reinforcing the idea that the deeper model has a larger capacity (despite using fewer parameters) and as a results it benefits more from regularisation. As an experiment, the two ensembles {LeNet(GS) + LeNet(RGB)} and {VGGNet(GS) + VGGNet(RGB)} were merged into a 4-network ensemble, obtaining a classification accuracy of 96.9%, as shown in the last row of Table 5.2.

A random selection of 16 images incorrectly classified by the {LeNet(GS) + LeNet(RGB)} ensemble are shown in Figure 5.7b. Interestingly, one of the major remaining challenges is classifying very dark images, or more generally images exhibiting very poor contrast (see e.g. the "priority road" sign in the third row of Figure 5.7b). Simple preprocessing techniques for contrast adjustment such as histogram equalisation (used on this data set by e.g. Cireşan *et al.* [6]) could help addressing this problem. An alternative idea could be to artificially augment the training data not only by translating, scaling, and rotating images, but also by applying random darkening and overexposure. This could potentially help the network to learn contrast-invariant features, increasing the classification robustness of underexposed or overexposed images.

6

Conclusions and future work

Research in deep learning is moving at extremely fast pace, in both academia and industry, and while some commercial applications based on deep learning methods have already appeared on the market in the past few years, their number and impact is predicted to grow enormously in the near future. For these reasons, it is important to provide researchers and engineers with the tools needed to quickly develop deep learning solutions, by covering as many platforms and developing environments as possible.

The Conv.NET class library, implemented in this project from the ground up, is the first attempt, to the best of the writer's knowledge, at creating a lightweight toolbox for deep learning under the .NET framework. Using this library, deep convolutional neural networks for image classification can be easily created and trained within C# projects, and quickly integrated with other .NET applications. Furthermore, both the training and inference processes can be greatly accelerated using a GPU or any other computational device supporting OpenCL™, an open framework for heterogeneous parallel computing.

In order to bench-test the Conv.NET library, in the second part of this work different CNNs were trained for the task of classifying traffic signs, using the German Traffic Sign Recognition Benchmark data set. Specifically, the attention was focused on two network architectures, respectively denoted by LeNet and VGGNet. The best single-model classification accuracy (96.2%) on the official GTSRB test data was obtained by training a LeNet architecture with dropout and early stopping on RGB images. By averaging the prediction of an ensemble of 4 CNNs, including two LeNet and two VGGNet architectures, trained on both greyscale and colour images, the classification accuracy on the test set increased to 96.9%.

These results are still quite far from state-of-the-art performance (99.65% [26]) on this data set . However, they were obtained with relatively little effort in terms of fine-tuning and hyperparameter optimisation and, importantly, without any form of artificial data augmentation. In fact, improving over the state-of-the-art was out of the scope of this project, and perhaps not particularly meaningful *per se*. Instead, the data set was used for benchmarking the impact of different features implemented in the Conv.NET library throughout the project, and in particular of different regularisation methods.

In general, dropout [7] was observed to consistently outperform L2 penalty at regularising both types of architecture. In fact, results seem to suggest that L2 penalty has seldom beneficial effects in terms of classification performance, provided that early stopping is used (see Figures 5.4 and 5.4). This agrees with recommendations in Bengio, 2012 [33], although weight decay remains nowadays widely used in training CNNs for image classification. Understanding when L2 penalty is beneficial and how to tune the decay hyperparameter λ without resorting to computationally expensive methods such as cross-validation remains an active area of research [12].

Remarkably, the VGGNet architecture ($\sim$462k parameters) was found to be more prone to overfitting than the LeNet model ($\sim$827k parameters), with the latter consistently ob-

taining a slightly higher classification accuracy than the former on the test data. Interestingly, the performance gap between the two models, relatively large (2.8%) in the case of no regularisation, was observed to shrink as more and more powerful regularisation methods were used and more information was made available to the network (0.4% in the case of single-architecture ensembles). This led to argue that the VGGNet model, despite the lower number of parameters, may have a significantly larger capacity than that of the LeNet.

The proposed, natural explanation for this behaviour is that the representational capacity of a CNN does not only depend on the number of parameters in the model, but also on the model's depth. In other words, estimating the complexity of the functions that a network can learn to approximate by only considering the number of parameters it contains is misleading, because it does not take into account the effect of feature *compositionality*, crucially dependent on the model's depth. Theoretical studies have just started to shed light on this key idea behind the success of deep learning [12], and further developments are certainly needed in order to fully understand it. In retrospective, since the VGGNet architecture is about twice as deep as that of LeNet (19 vs 10 hidden layers, using the simple layer nomenclature), it should not be surprising that it is more severely affected by overfitting and benefits more from regularisation.

On the one hand, it could be argued that the capacity of the LeNet model may simply be better suited for the complexity of the GTSRB data set. On the other hand, developments in deep learning over the recent years have moved in the direction of using increasingly deep and complex architectures, while devising and using better strategies for regularisation at the same time [2, 3, 8, 26]. In light of the results obtained in this study with increasingly powerful regularisation methods, we can expect the VGGNet to eventually perform better than the LeNet, once properly regularised. To this aim, several strategies can be experimented and combined, including:

- Artificial data set augmentation. This may include standard methods for generating synthetic images during training, such as applying translations, rotations, horizontal flipping, and colour jittering to the original images. However, given the wide variations of lighting conditions in the GTSRB data set, it would be interesting to assess the effect of brightness and/or contrast perturbation of images as a novel form of data augmentation method.
- Using batch normalisation (Ioffe and Szegedy, 2015 [8]), a very promising technique to address multiple problems affecting the training of deep architectures. This method has empirically proved not only to greatly accelerate the learning speed, but also to improve the predictive performance of deep CNNs by acting as a regulariser.
- A more thorough exploration of the hyperparameter space. This applies in particular to the regularisation strength λ in the L2 penalty term, usually selected using k-fold cross validation or through random search [12].
- Applying dropout to the entire network (potentially including the input layer), rather than to the final (fully-connected) stage only. The effectiveness of this approach is controversial [7], and seems to depend on the data to be classified.

All of the above strategies can lead to incremental improvements in terms of classification accuracy and apply to both CNN models investigated in this work, although, as discussed, their effect would arguably be more important for the deeper model (VGGNet).

An alternative approach would be to explore a completely different kind of architecture, perhaps inspired by the recently introduced residual network framework [3], featuring skip-connections throughout the network and making heavy use of batch normalisation.

Clearly, traffic sign classification with deep convolutional neural networks is just one example of the successful application of deep learning methods to a real-world problem. The Conv.NET library was implemented to be flexible and potentially be used for solving other classification tasks with deep neural networks. Most components needed to train CNNs for image classification are available and fully functioning, but several additional features are under development (such as batch normalisation, residual networks and a graphical user interface), in the hope that the library may serve as a starting point for future projects, both in the open source community and at the Adaptive Systems research group at Chalmers University of Technology.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *ArXiv preprint arXiv:1409.1556*, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", *ArXiv preprint arXiv:1512.03385*, 2015.

[4] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition", *Neural Networks*, vol. 32, pp. 323–332, 2012.

[5] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks", in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, IEEE, 2011, pp. 2809–2813.

[6] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber, "Multi-column deep neural network for traffic sign classification", *Neural Networks*, vol. 32, pp. 333–338, 2012.

[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[8] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *ArXiv preprint arXiv:1502.03167*, 2015.

[9] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives", *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1798–1828, 2013.

[10] J. Schmidhuber, "Deep learning in neural networks: An overview", *Neural Networks*, vol. 61, pp. 85–117, 2015.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[12] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning", Book in preparation for MIT Press, 2016, [Online]. Available: `http://www.deeplearningbook.org`.

[13] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[14] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[15] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences", 1974.

[16] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators", *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[17] O. Delalleau and Y. Bengio, "Shallow vs. deep sum-product networks", in *Advances in Neural Information Processing Systems*, 2011, pp. 666–674.

[18] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[19] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex", *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[20] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network", in *Advances in neural information processing systems*, Citeseer, 1990.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[22] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets", *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[23] D. C. Cireşan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification", in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, 2011, p. 1237.

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[25] A. de la Escalera, L. E. Moreno, M. A. Salichs, and J. M. Armingol, "Road traffic sign detection and classification", *IEEE Transactions on Industrial Electronics*, vol. 44, no. 6, pp. 848–859, Dec. 1997.

[26] J. Jin, K. Fu, and C. Zhang, "Traffic sign recognition with hinge loss trained convolutional neural networks", *Intelligent Transportation Systems, IEEE Transactions on*, vol. 15, no. 5, pp. 1991–2000, 2014.

[27] M. Haloi, "Traffic sign classification using deep inception based convolutional networks", *ArXiv preprint arXiv:1511.02992*, 2015.

[28] A. Balasubramaniam, OpenCL.Net: open source .NET bindings to OpenCL, License: EPL-1.0. Version 2.2.9 (23 October 2013). Available on NuGet: `https://www.nuget.org/packages/OpenCL.Net/`.

[29] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[30] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing", in *Tenth International Workshop on Frontiers in Handwriting Recognition*, Suvisoft, 2006.

[31] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)", *ArXiv preprint arXiv:1511.07289*, 2015.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026–1034.

[33] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures", in *Neural Networks: Tricks of the Trade*, Springer, 2012, pp. 437–478.