Fast Convolutional Neural Network Training and Classification on CUDA GPUs

master thesis in computer science

by

Daniel Strigl and Klaus Kofler

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Dr. Stefan Podlipnig,
Institute of Computer Science

Innsbruck, 4 February 2010

Certificate of authorship/originality

We certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

We also certify that the thesis has been written by us. Any help that we have received in our research work and the preparation of the thesis itself has been acknowledged. In addition, we certify that all information sources and literature used are indicated in the thesis.

Daniel Strigl and Klaus Kofler, Innsbruck on the 4 February 2010

Abstract

Machine learning algorithms are usually very computational intensive and rather complex in their implementation, especially the well performing ones. In case of neural networks it is mainly the training time which is often very time-consuming. Training of neural networks using a voluminous training dataset can take several days or even weeks. These two points are also valid for the so-called *Convolutional Neural Networks* (CNNs), invented by Yann LeCun in the early 1990s. However, CNNs deliver state-of-the-art performance on two-dimensional pattern recognition and classification tasks in a broad area of applications.

This work aims at mastering both major drawbacks of CNNs, namely the time-consuming training and the implementation complexity. Therefore, a flexible, high performance but easy-to-use library for CNNs was developed. This library hides the implementation complexity from the user since CNNs can easily be constructed by composing the single types of layers needed for the network. Furthermore, some implementations of well performing networks described in the literature are delivered with this library.

The second main goal of this thesis is to reduce the training time of CNNs and evaluating the performance gains that can be achieved using *GPGPU computing* in this area of application. To reduce the training time of a CNN our library aims at using the advantages of today's parallel processors. First a reference implementation for x86 multicore CPUs has been designed. In a second step an implementation for CUDA enabled NVIDIA GPUs has been developed. The implementations were used to perform benchmarks in terms of classification rate as well as execution speed using known networks. This work will demonstrate that today's GPUs bear a serious advantage over traditional CPUs in terms of execution speed on this particular kind of machine learning algorithms, reaching speedups of up to 25 times.

Zusammenfassung

Maschinelle Lernalgorithmen sind in der Regel sehr rechenintensiv und komplex in der Implementierung. Dies gilt insbesondere für die leistungsfähigen Algorithmen. Im Fall von Neuronalen Netzen ist es besonders das Training das sehr zeitaufwändig ist. Das Trainieren von Neuronalen Netzen mit Hilfe einer umfangreichen Trainingsmenge kann einige Tage bis Wochen beanspruchen. Diese beiden Nachteile gelten auch für die in den frühen 90igern von Yann LeCun erfundenen *Convolutional Neural Networks* (CNN). Allerdings liefern CNNs die mitunter besten Ergebnisse bei Problemen der zweidimensionalen Mustererkennung und -klassifizierung.

Diese Arbeit zeigt wie die beiden zentralen Nachteile von CNNs, das zeitaufwändige Training sowie die komplexe Implementierung, minimiert werden können. Es wurde eine flexible, sehr performante aber trotzdem einfach zu handhabende Bibliothek für CNNs entwickelt. Diese Bibliothek verbirgt die Komplexität vor dem Anwender, da CNNs einfach durch das Zusammenfügen der einzelnen Typen von Schichten erstellt werden können. Aufbauend auf dieser Implementierung wurden bekannte leistungsstarke Netzwerke aus der Literatur realisiert.

Das zweite Ziel dieser Arbeit ist die Reduzierung der Trainingszeit von CNNs und die Evaluierung der Performancezuwächse die durch den Einsatz von *GPGPU Computing* auf diesem Anwendungsgebiet erreicht werden können. Um die Trainingszeit zu verkürzen nutzt unsere Bibliothek die Vorteile heutiger paralleler Prozessoren. Zuerst wurde eine Referenzimplementierung für x86 Multicore-CPUs entwickelt. In einem weiteren Schritt wurde eine Implementierung für CUDA-fähige GPUs erstellt. Diese beiden Implementierungen wurden verwendet um mehrere Messungen mit bekannten Netzwerken, unter Berücksichtigung der Klassifizierungsrate als auch der Ausführungsgeschwindigkeit, durchzuführen. Es wird gezeigt, dass bei dieser Art von maschinellen Lernalgorithmen heutige GPUs einen beträchtlichen Geschwindigkeitsvorteil gegenüber traditionellen CPUs besitzen und eine bis zu 25-fache Beschleunigung erreichen.

Acknowledgments

We want to thank the following people that contributed to our thesis in one or the other way. First, we would like to express our deepest appreciation and gratitude to Dr. Stefan Podlipnig, not only for supervising this thesis but also for providing us with the needed hardware (see Table 6.1 on page 86) and for being a great lecturer. We would like to thank Dr. Alfred Strey for introducing us to the neural network theory and for the helpful suggestions and hints during the work on this thesis. Furthermore, we would like to thank Daniel Keysers (Google Switzerland), Alex Graves (Technische Universität München), Son Lam Phung (University of Wollongong), Fok Hing Chi Tivive (University of Wollongong), Michael Jahrer (commendo research & consulting), Patrice Simard (Microsoft Research), and Clément Farabet (New York University) for answering our questions and for providing us with some resources. Thanks also go to our colleagues, the IT-Boyz$^{\text{IBK}}$ = { scher, $^{*\text{partymartin}*}$marcom, dani[1], athon, $^{*\text{lol}*}$adrian[1], gigi, holzi, fritz[1], hausi, gregor, michi, herbert, and everybody we forgot to mention }, for the common lunch hours and coffee breaks and for making the Institute of Computer Science at the University of Innsbruck a nice place to study. We are also deeply grateful to our parents for all the support they gave us and are still giving us. Finally, I (Daniel) would like to thank Yvonne for her love, encouragement, and patience and my daughter Joana[2] just for being there and for enriching my life in so many ways.

Thanks all of you.

This work was partially funded by the Fakultäten Servicestelle of the University of Innsbruck under the Förderungsstipendium.

[1] see also Figure 7.13(a) on page 116

[2] see also Figure 7.13(b) on page 116

Publications

- D. Strigl, K. Kofler, and S. Podlipnig. Performance and Scalability of GPU-based Convolutional Neural Networks. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010)*, Pisa, Italy, February 2010. (to appear)

Acronyms

1D	One-dimensional (1-dimensional)
2D	Two-dimensional (2-dimensional)
3D	Three-dimensional (3-dimensional)
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
API	Application Programming Interface
bprop	Backpropagation
CBCL	Center for Biological and Computational Learning
CE	Cross-Entropy
Cg	C for graphics
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DP	Double Precision
DRAM	Dynamic Random Access Memory
FMA	Fused Multiplication Addition
FP	Floating-Point
fprop	Forward Propagation
FPU	Floating-Point Unit
GLSL	OpenGL Shading Language

GNU	Recursive acronym for "GNU's Not Unix!"[3]
GPGPU	General-Purpose computation on Graphics Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HLSL	High Level Shading Language
HPC	High Performance Computing
IEEE	Institute of Electrical and Electronics Engineers
IDM	Image Distortion Model
MAD	Multiplication Addition
MIT	Massachusetts Institute of Technology
MLP	Multilayer Perceptron
MNIST	Modified NIST (handwritten digit database)
MSE	Mean-Squared-Error
NIST	National Institute of Standards
NN	Neural Network; Nearest Neighbor
OCR	Optical Character Recognition
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
PCA	Principal Component Analysis
PGM	Portable Gray Map
RBF	Radial Basis Function
RNN	Recurrent Neural Network
ROP	Raster Operation Processor
RPROP	Resilient Backpropagation

[3] `http://www.gnu.org/gnu/manifesto.html`

SDK	Software Development Kit
SFU	Special Function Unit
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SP	Scalar (or Streaming) Processor; Single Precision
SSE	Streaming SIMD Extensions
SVM	Support Vector Machine
TDP	Termal Design Power
Tex	Texture
TPC	Texture/Processor Cluster

Contents

Chapter 1

Introduction

Neural networks perform very well on pattern recognition and classification tasks with a large amount of training data. For image classification, like optical character recognition (OCR), *Convolutional Neural Networks* (CNNs) deliver state-of-the-art performance [1]. CNNs are a derivative of *multilayer perceptrons* (MLPs) optimized for two-dimensional pattern recognition. The area of applications for CNNs is widespread. They are used for handwriting recognition [1, 2], face, eye and license plate detection [3–9], and in non-vision applications such as semantic analysis [10].

The biggest drawback of CNNs, besides a complex implementation, is the long training time. Since CNN training is very compute- and data-intensive, training with large datasets may take several days or weeks. The huge number of floating-point operations and relatively low data transfer in every training step makes this task well suited for *GPGPU* (General Purpose GPU) computation on current *GPUs* (Graphic Processing Units). The main advantage of GPUs over CPUs is the high computational throughput at relatively low cost, achieved through their *massively parallel architecture*.

In the past using the GPU for general purpose calculations required a deep understanding of the hardware architecture, where the problems had to be implemented using a graphics API like OpenGL or DirectX. Therefore, the algorithms had to be transformed into a graphics pipeline friendly format. With the emergence of *CUDA* (Compute Unified Device Architecture) [11] and the so-called *unified shader architecture* [12] things have changed. The CUDA language bears resemblance to the C programming language and is therefore much more common for most programmers than the graphics API languages. This results in a shorter training period, faster adoption and higher efficiency. Furthermore, unified shaders are better adapted to perform general computations than earlier architectures.

In contrast to other classifiers like *Support Vector Machines* (SVMs) where several parallel implementations for CPUs [13] and GPUs [14] exist, similar efforts for CNNs are missing. Therefore, we implemented a high performance but

still flexible library in C++ and CUDA to accelerate the training and classification process of arbitrary CNNs. Due to the fact that the ideal parameters of a neural network (network structure, initial value range, learning method, learning rate, etc.) can only be determined by testing and evaluating, shortening the training time often leads to better results. Our goal was to demonstrate the performance and scalability improvement that can be achieved by shifting the computation-intensive tasks of a CNN to the GPU.

We started with a straight forward implementation without any manual parallelization or vectorization. To fairly compare the GPU with the CPU variant of our library, we optimized this implementation using functions from Intel's *performance libraries* IPP (Integrated Performance Primitives) [15] and MKL (Math Kernel Library) [16]. Those libraries take the full advantage of the newest *Streaming SIMD Extensions* (SSE) of the CPU. These enhancements resulted in a quite fast implementation.

The GPU implementation using CUDA exchanges the mathematical vector and matrix operations with functions either from NVIDIA's CUBLAS library [17] if appropriate functions are available there or our own implementations otherwise. Each kernel-function performs one mathematical operation, e.g. a matrix-vector multiplication or the summation of all elements in a vector. In order to not affect the C++ class structure of the CPU implementation each CUBLAS or self implemented function (kernel-call) is embedded in a templatized C++ function. This means that after every operation on the GPU the program control is passed back to the CPU, even if no data transfer between CPU and GPU is necessary. Although this introduces some small overheads the general reusable structure of our library is preserved.

Finally, in order to demonstrate the potential of CNNs in the area of pattern recognition and classification we implemented two *real-world applications* using our library: an application for *handwritten character recognition* and a *face detection system* to detect and locate human faces in an image. These two applications present the advantage of CNNs over other types of neural networks (e.g. MLPs) through their *built-in invariance* against certain transformations of the input patterns. Furthermore, it shows how CNNs can be used to solve practical problems.

According to the statements in [18] which emphasis *the need for open source software in machine learning* we made our library [19] public available[1].

[1] `http://cnnlib.sourceforge.net`

1.1 Outline

This thesis consists of eight chapters with an introduction in Chapter 1, the rest of the thesis is organized as follows:

- Chapter 2 gives a brief introduction to artificial neural networks. The first part introduces neural networks in general and describes the relationship of biological and artificial neural networks. The second part introduces the probably simplest kind of artificial neural network, namely the multilayer perceptron. This is followed by an outline on other types of neural networks. The next part describes how neural networks are trained using gradient-based error backpropagation. The chapter closes with a description of all error and activation functions used in this thesis.

- Chapter 3 describes the details and peculiarities of Convolutional Neural Networks (CNNs). It starts with an introduction to CNNs and their network architecture, followed by a detailed description of the mathematical model. The third part explains how CNNs can be trained with the standard backpropagation algorithm. The chapter concludes with an overview of the different types of CNNs proposed over the last years.

- Chapter 4 introduces the Compute Unified Device Architecture (CUDA) which has been used to implement the GPU-based part of our library. This chapter starts with an introduction to the history of GPGPU computing and introduces the architecture of current and future CUDA supporting hardware. Furthermore, it explains the threading model of CUDA as well as it's different APIs and the different types of memory that can be used.

- Chapter 5 deals with the description of our implementation. It gives an overview of already existing CNN implementations, followed by a description of the implementation procedure of our library. The next part explains the applied techniques to simplify and accelerate our implementation. Furthermore, it covers the performance enhancements of our library by parallelization on x86 multicore CPUs and GPUs. The chapter closes with a description of the software architecture of our implementation.

- Chapter 6 shows the benchmark results we gained during the work on this thesis. The first part describes how the used numerical precision affects the classification result of a CNN. In the second part we compared the execution time of our library with a reference implementation from the New York University. The last part covers various benchmarks to show

our CNN implementation's performance with a focus on the comparison of scalability between our CPU and GPU implementation.

- Chapter 7 describes how neural networks build with our library perform on different tasks. The performance of various CNNs in terms of classification rates on handwritten digit recognition are demonstrated in the first part, while the second part describes how CNNs can be used to detect and locate human faces in digital images.

- Finally, Chapter 8 gives a conclusion of our work as well as an outlook on future research in this area.

The additional appendix at the end of this thesis is built of following parts: The first part (A) explains in detail the databases used in this thesis for testing and evaluation, followed by the description of a method to enlarge these datasets by virtual training images in order to improve the accuracy of a neural network (B). In the next part (C) we describe the bug in the CUDA architecture we discovered during the work on this thesis. Finally, the last part (D) closes with a collection of source code examples for CUDA.

1.2 Responsibilities

This thesis was written by two persons, Daniel and Klaus. Table 1.1 shows how the responsibilities in this work were distributed.

Contents	**Author**
• Chapter 1	Daniel and Klaus
• Chapter 2	Klaus
• Chapter 3	Daniel
• Chapter 4	Daniel and Klaus
• Chapter 5	
▷ Section 5.1	Daniel and Klaus
▷ Section 5.2	Daniel
▷ Section 5.3	Daniel
▷ Section 5.4	Klaus
▷ Section 5.5	Klaus
• Chapter 6	
▷ Section 6.1	Klaus
▷ Section 6.2	Klaus
▷ Section 6.3	Daniel
• Chapter 7	
▷ Section 7.1	Klaus
▷ Section 7.2	Daniel
• Chapter 8	Daniel and Klaus
• Appendix A	Klaus
• Appendix B	Daniel
• Appendix C	Daniel and Klaus
• Appendix D	Daniel and Klaus

Table 1.1: Work-sharing in the thesis.

Chapter 2

A Review of Artificial Neural Networks

As mentioned in the previous chapter, neural networks are powerful machine learning algorithms leading to good results in a large area of applications. However, the internals of neural networks are not easy to understand and rather complex. This chapter provides all needed background material to understand simple neural networks and delivers literature hints to gain a deeper knowledge of neural networks.

The first two sections introduce neural networks in general and describe the relationship of biological and artificial neural networks. Section 2.3 introduces the probably simplest kind of artificial neural network, namely the multilayer perceptron, while Section 2.4 gives an outline on other types of neural networks. This is followed by Section 2.5 which describes how neural networks are trained using gradient-based error backpropagation. The chapter closes with a description of all error and activation functions used in this thesis in Section 2.6 and Section 2.7, respectively.

2.1 Biological Neural Networks

The name *neural network* (NN) derives from the cells that can be found in the brain of nearly every being. In a biological neural networks a *neuron* is one single, *excitable cell* [20]. A schematic illustration of a neuron cell in a mammalian brain is shown in Figure 2.1. A peculiarity of NNs is the very high degree of interconnection. While every single neuron has only one single output (axon) there are several inputs (dendrites) from sensors or other neurons. If a sensor or neuron gets excited it "fires", which means that it generates a certain potential on its output using biochemical reactions. One single neuron can only make a *binary decision* (to fire or not to fire). This implies that a huge number of neurons is needed to perform any meaningful task[1]. Artificial NNs, as described in the following section, have been developed to simulate the structure and functional aspects of the biological NNs.

[1] According to [20] a human brain for example contains more than 100 billion neurons.

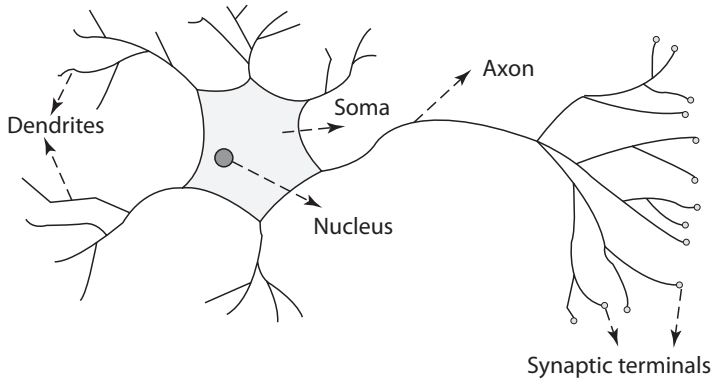


Figure 2.1: A neuron cell in a mammalian brain (from [21], January 2010).

2.2 Artificial Neural Networks

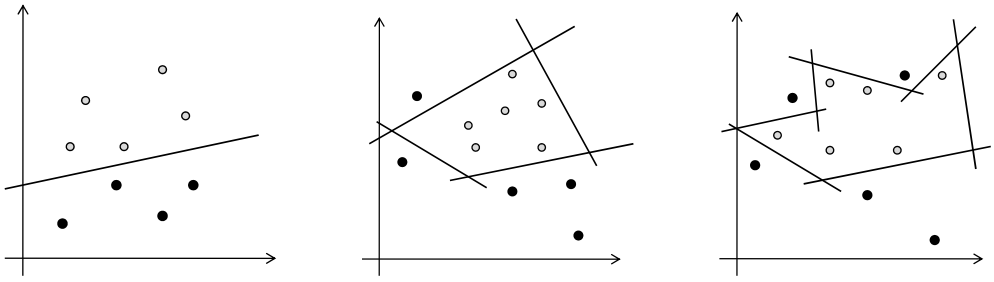
In computer science *artificial neural networks* (ANNs), in short neural networks, are usually used for pattern recognition or classification tasks [22, 23]. Such a program bears to reassemble a living brain. Therefore, they form a *network* of many neurons. They are one of the most commonly used *machine learning* algorithms. Machine learning implies that the program has to be trained before it can perform any real-world application. To be able to train a NN a *training dataset* is needed which consists of *preclassified examples* that should be representative for all possible inputs the network later may has to deal with.

There exist several types of neurons to build an ANN. One of the oldest and simplest are the so-called *Rosenblatt perceptrons* introduced by Rosenblatt in 1957 [24]. Such a neuron is a binary classifier, consisting of several inputs and one output. The output y is calculated by the following function

$$y = \begin{cases} 1, & \text{if } \mathbf{x} \cdot \mathbf{w} + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

where $\mathbf{x}$ is a vector representing the *excitement* of the incoming connections and vector $\mathbf{w}$ contains the *weights* of the incoming connections. The statement $\mathbf{x} \cdot \mathbf{w}$ denotes the dot product of these two vectors and b is the *bias term*, a threshold that determines how many input connections have to be excited at the same time to make the neuron fire.

One single Rosenblatt perceptron is able to solve all binary, linear separable classification problems as shown in Figure 2.2(a). To solve classification problems that are not linear separable, several layers of perceptrons have to be arranged in a series. The characteristic of such *multilayer perceptron networks* (MLPs, see Section 2.3) is that the neurons of a layer are only connected in a

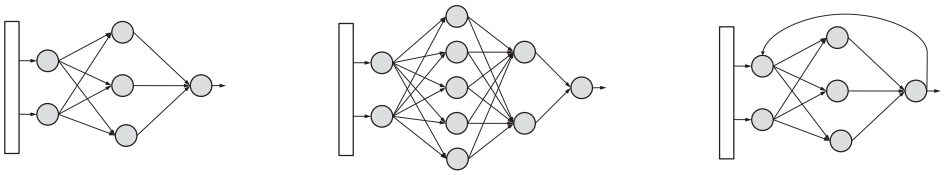


(a) Linear separable classification problem solvable by one single perceptron.

(b) Classification problem where one class can be enclosed by a convex polygon, solvable by a two-layer perceptron network.

(c) Example of a two class classification problem, solvable only by a MLP with three or more layers.

Figure 2.2: Examples for different classification problems.



(a) Two-layer perceptron NN.

(b) Three-layer perceptron NN.

(c) Recurrent neural network.

Figure 2.3: Some examples of multilayer neural networks.

feed-forward structure to neurons in their preceding layer, but not to other neurons of their own or following layers. MLP networks with two layers as shown in Figure 2.3(a) can classify convex polygons (see Figure 2.2(b)), networks consisting of three (as the one in Figure 2.3(b)) or more layers can classify a set of arbitrary shapes (see Figure 2.2(c)). The rectangles in Figure 2.3(a), 2.3(b), and 2.3(c) represent some sensors, e.g. the values of the pixels in an image recognition tasks. They form the *input layer*, sometimes also described as the 0th layer. The last layer is called the *output layer*, while the layers in between form the so-called *hidden layers*. A single layer network does not have any hidden layers, but an input and an output layer. To distinguish between more than two classes, the output layer of the network contains usually one neuron for each single class (an *one-of-n coding*, also known as *place code* or *grandmother cell code*[2]). During the classification the pattern is then assigned to the class

[2]This expression comes from the joke that there is a single neuron in the brain that represents your grandmother (`http://sciencehouse.wordpress.com/2008/07/18/neural-code`).

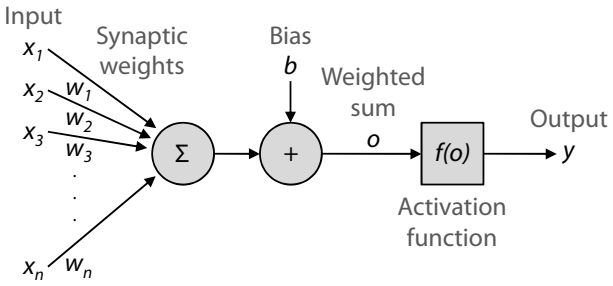


Figure 2.4: Structure of a Rosenblatt perceptron with a continuous, differentiable activation function.

which's neuron has the highest excitation.

The major drawback of the Rosenblatt neuron's output function (2.1) is that it's *derivation* is not defined. However, a well defined derivation of the output function is indispensable for all *gradient-based backpropagation learning* algorithms [22, 23], which are the most widely used and probably most effective learning algorithms for multilayer NNs. Therefore, the output function of the Rosenblatt perceptron is usually extended according to Figure 2.4 with

$$y = f(o), \quad o = \mathbf{x} \cdot \mathbf{w} + b \tag{2.2}$$

where o is the *weighted sum* of the neuron's inputs and f is a so-called *activation* (*squasher* or *transfer*) *function* which's derivation is well defined. If f is the identity the neuron is called *linear*. The function f is usually a *sigmoid function*, e.g. the tangens hyperbolicus or some variation of it where the derivation is well defined (see also Section 2.7).

2.3 Multilayer Perceptrons

The probably oldest, simplest and widely used type of ANN is the *multilayer perceptron* (MLP). It consists of one or more hidden layers and an output layer organized in a feed-forward structure as shown in Figure 2.3(a) and 2.3(b). In this structure the activation of the neurons is propagated layer-wise from the input to the output layer.

All layers of a MLP network are *fully connected layers*, where every neuron of such a layer is connected to all neurons in the previous layer. Thus, the output $y^{(l)}(j)$ of neuron j in layer l is calculated according to

$$y^{(l)}(j) = f^{(l)}\left(o^{(l)}(j)\right) \tag{2.3}$$

where $f^{(l)}$ denotes the activation function of layer l and $o^{(l)}(j)$ the weighted sum input to neuron j in layer l. The weighted sum input $o^{(l)}(j)$ is given by

$$o^{(l)}(j) = \sum_{i=0}^{N^{(l-1)}-1} y^{(l-1)}(i) \cdot w^{(l)}(i,j) + b^{(l)}(j) \tag{2.4}$$

where $N^{(l-1)}$ defines the number of neurons in the preceding layer $l-1$ while $w^{(l)}(i,j)$ is the weight for the connection from neuron i in layer $l-1$ to neuron j in layer l and $b^{(l)}(j)$ the bias of neuron j in layer l. The most commonly used activation functions of a MLP are described in detail in Section 2.7.

2.4 Other Network Topologies

There exist a lots of network variants, differing in the way their neurons are connected. Common examples are *Convolutional Neural Networks* (CNNs, described in Chapter 3) and *Recurrent Neural Networks* (RNNs) [25, 26]. In the latter ones neurons of layer l have additional inputs from neurons in the same layer or in a following layer $l+n$ as shown in Figure 2.3(c). This adds some kind of memory to the network as results depend on the previous states of it. This can be an advantage if the network is used for example for predicting time series, but it does not make any sense in pattern recognition because we analyze single images and every result is totally independent from previous ones. Therefore, in all networks implemented in this thesis the neurons of a layer are only connected with neurons in the preceding layer (called *feed-forward structure*).

2.5 Neural Network Training

Training of NNs can be divided in *supervised*, *unsupervised*, and *reinforcement learning*. Supervised learning [27, pages 9–41] means that there exist a preclassified training dataset which is trained by the network. In contrast to that, unsupervised learning [27, pages 485–586] uses a training dataset which's desired output is unknown. It has to be determined by the training algorithm (this training method is often used for *clustering tasks* [23, 28]). When using reinforcement learning the desired output is unknown at the training's beginning. It is determined during the training by the principle of *trial and error*, rewarding improvement and penalizing decline. This thesis will focus on supervised learning because only this training variant is used in this work.

In the following Subsection 2.5.1 we will focus on the error backpropagation algorithm since it is the most widely used and probably most effective learning

algorithm for feed-forward NNs, especially for MLPs. In Subsection 2.5.2 we present some more sophisticated gradient-based training algorithms used in this thesis. Subsection 2.5.3 explains how to initialize the trainable parameters of a NN. This is followed by a description of a data preprocessing method to enhance the performance of a NN in Subsection 2.5.4. Finally, Subsection 2.5.5 characterizes a break condition for NN training used in the scope of this thesis.

2.5.1 Error Backpropagation

During the *error backpropagation training*, initially presented by Rumelhart et al. [29], the weights of the network are adapted by applying the *gradient descent technique* in the *weight space* in order to minimize the error on the training dataset. It has to be noted that the backpropagation algorithm does not guarantee to find a *global minimum* which is an inherent problem of all gradient descent learning methods. However, there are several approaches in the literature to overcome this problem [30–33].

In the following parts we will explain in short the standard backpropagation algorithm applied to MLPs, while a more detailed description can be found in [22, pages 140–147] and [23, pages 151–173].

Network Error Computation

In all supervised training methods a pattern k of the training dataset is fed into the network to produce some output. This process is called *forward propagation* or *forward pass*. During the so-called *backward propagation* or *backward pass* the error at the network's output is calculated and propagated backwards through the network[3]. This error specifies the distance between the *actual* and the *desired output* according to some metric and can be calculated as follows

$$E^k = ||\mathbf{y}^k - \mathbf{d}^k||_m \tag{2.5}$$

where $\mathbf{y}^k$ is a vector containing the excitations of all neurons in the output layer and $\mathbf{d}^k$ denotes the training vector which contains the desired output (also known as *training signal*) for this pattern. Such a metric can be for example the *mean-squared-error* (MSE, see Section 2.6.1) or the *cross-entropy* (CE, see Section 2.6.2). In case of the MSE the error is calculated according to

$$E^k = \frac{1}{2 \cdot N^{(L)}} \cdot \sum_{j=0}^{N^{(L)}-1} \left(y^{(L),k}(j) - d^k(j) \right)^2 \tag{2.6}$$

[3] Hence the name "error backpropagation".

where $N^{(L)}$ denotes the number of neurons in the output layer L. In this context such a metric is called *error function* and represents either the error of one training example or the sum of errors produced by all training examples, depending if the training is performed in online or offline mode.

Error Sensitivity Computation

After determining the error on the network's output the *error sensitivity* (sometimes also called *local gradient*) of each layer can be calculated. The error sensitivity $\delta^{(l),k}(j)$ of a neuron j in layer l is defined as

$$\delta^{(l),k}(j) = \frac{\partial E^k}{\partial o^{(l),k}(j)} \tag{2.7}$$

which is the partial derivate of the error function E with respect to the weighted sum input $o^{(l),k}(j)$ to neuron j. Using the *chain rule of differentiation* [22, pages 142–143], we can express the error sensitivity of each neuron j in the output layer L as follows

$$\delta^{(L),k}(j) = f'^{(L)}(o^{(L),k}(j)) \cdot e^k(j) \tag{2.8}$$

where the symbol $f'^{(L)}$ denotes the derivate of the activation function in the output layer L and $e^k(j)$ describes the element j of the error vector calculated by the error function (see Section 2.6). In case of the MSE this is defined as

$$e^k(j) = y^{(L),k}(j) - d^k(j). \tag{2.9}$$

This error is then backpropagated through the whole network to gain the error sensitivity $\delta^{(l),k}(j)$ for each single neuron j in the hidden layers $l < L$ according to

$$\delta^{(l),k}(j) = f'^{(l)}\left(o^{(l),k}(j)\right) \cdot \sum_{i=0}^{N^{(l+1)}-1} \delta^{(l+1),k}(i) \cdot w^{(l+1)}(j,i), \tag{2.10}$$

where $w^{(l+1)}(j,i)$ denotes the weight on the connection from neuron j in layer l to neuron i in layer $l+1$.

Error Gradient Computation

In order to minimize the error on the training dataset by the gradient descent technique we have to calculate the *steepest descent* of the *error surface* [23] in the weight space. This is given by the opposite direction of the *error gradient*, which is defined as the partial derivate of the error function E with respect to the weights $w^{(l)}(i,j)$ and the biases $b^{(l)}(j)$ of all layers. It can be obtained using the

error sensitivity $\delta^{(l),k}(j)$ and the output $y^{(l-1),k}(i)$ of the corresponding neuron i in the preceding layer $l-1$:

$$\frac{\partial E^k}{\partial w^{(l)}(i,j)} = \delta^{(l),k}(j) \cdot y^{(l-1),k}(i) \tag{2.11}$$

$$\frac{\partial E^k}{\partial b^{(l)}(j)} = \delta^{(l),k}(j) \tag{2.12}$$

Weight Update

To make the actual output more similar to the desired one, a *correction value* of each weight

$$\Delta w^{(l),k}(i,j) = -\eta \cdot \frac{\partial E^k}{\partial w^{(l)}(i,j)} \tag{2.13}$$

and bias

$$\Delta b^{(l),k}(j) = -\eta \cdot \frac{\partial E^k}{\partial b^{(l)}(j)} \tag{2.14}$$

has to be calculated, where η denotes the *learning rate* with $0 < \eta < 1$ (often decreased during training time [27, page 397]). This learning rate η defines the step length in the *negative gradient direction*.

After a certain number of forward and backward propagations the weights and biases of the network are then updated according to

$$w^{(l)}(i,j) = w^{(l)}(i,j) + \sum_{k=1}^{P} \Delta w^{(l),k}(i,j) \tag{2.15}$$

and

$$b^{(l)}(j) = b^{(l)}(j) + \sum_{k=1}^{P} \Delta b^{(l),k}(j) \tag{2.16}$$

where P denotes the number of backward propagations that are *batched* together. If the update is performed after every presented pattern the training is called *online* or *stochastic backpropagation*, if it is performed only once for the whole training dataset it is called *offline* or *batch backpropagation*.

One sweep through the entire training dataset is referred as *training epoch* and is usually performed several times on the network. To determine when to stop the training process is not an easy task and there are several approaches in the literature to overcome this problem. One of it, the *early stopping method*, is described in Subsection 2.5.5. When training in online mode it is also recommended to shuffle the training patterns (randomizing the order of the patterns

in the training set) in each epoch [34, pages 15–16]. This helps to reduce the probability to get stuck in a *local minima*.

The Algorithms 1 and 2 summarize the standard online and offline backpropagation algorithm for MLPs. In this algorithms the symbol ϵ determines the convergence criteria and n_{max} denotes the maximum number of training epochs, while K is the number of patterns in the training dataset.

2.5.2 Other Training Algorithms

Besides the standard backpropagation we also used two other gradient-based learning algorithms in this thesis, namely the *gradient descent with momentum term* [22, 35] and the *resilient backpropagation* (RPROP) [36]. Both of this algorithms are an enhancement of the standard backpropagation in order to improve the gradient descent method, e.g. to accelerate the convergence of the algorithm or to avoid to get stuck in a local minima.

Gradient Descent with Momentum Term

This learning algorithm takes into account previous weight changes in form of a so-called *momentum term* [35] to improve the convergence of the algorithm.

The correction values for the weight and bias update in (2.15) and (2.16) are calculated according to

$$\Delta w^{(l)}(i,j)^{(t)} = \underbrace{\alpha \cdot \Delta w^{(l)}(i,j)^{(t-1)}}_{\text{momentum term}} - \eta \cdot \frac{\partial E^k}{\partial w^{(l)}(i,j)} \tag{2.17}$$

and

$$\Delta b^{(l)}(j)^{(t)} = \underbrace{\alpha \cdot \Delta b^{(l)}(j)^{(t-1)}}_{\text{momentum term}} - \eta \cdot \frac{\partial E^k}{\partial b^{(l)}(j)} \tag{2.18}$$

where α is the *momentum rate* with $0 \leq \alpha < 1$. The variables $\Delta w^{(l)}(i,j)$ and $\Delta b^{(l)}(j)$ in these two equations are noted as functions of iteration t, where the values of the current iteration t depend on the values of the previous iteration $t-1$. The algorithm must therefore store the previous update values.

One of the properties of this algorithm is that the weight changes will get bigger if there are several consecutive weight updates in the same direction (Figure 2.5(a)) and will get smaller if the weight update oscillates (Figure 2.5(b)) or is equal to zero. This results in an improvement of the *convergence speed* which is described in detail in [22, pages 267–268] and [23, pages 186–192].

Algorithm 1 The standard online backpropagation algorithm for MLPs.

Initialize all weights w and biases b of the MLP with small random values.
Set the learning rate η to a small positive value.
$n = 1$

repeat

  /* Online backpropagation */
  for $k = 1$ to K **do**
    /* Forward pass */
    Propagate the pattern x^k through the network to obtain $y^{(L),k}$.
    /* Backward pass */
    Compute the error e^k for the $N^{(L)}$ output neurons:
      $e^k(j) = y^{(L),k}(j) - d^k(j)$ /* for MSE */
    Compute $\delta^{(L),k}$ for the $N^{(L)}$ output neurons:
      $\delta^{(L),k}(j) = f'^{(L)}(o^{(L),k}(j)) \cdot e^k(j)$
    Compute $\delta^{(l),k}$ for the $N^{(l)}$ neurons of the hidden layers:
      $\delta^{(l),k}(j) = f'^{(l)}\left(o^{(l),k}(j)\right) \cdot \sum_{i=0}^{N^{(l+1)}-1} \delta^{(l+1),k}(i) \cdot w^{(l+1)}(j,i)$
    Compute $\partial E^k / \partial w^{(l)}(i,j)$ for the $N^{(l)}$ neurons of each layer:
      $\partial E^k / \partial w^{(l)}(i,j) = \delta^{(l),k}(j) \cdot y^{(l-1),k}(i)$
    Compute $\partial E^k / \partial b^{(l)}(j)$ for the $N^{(l)}$ neurons of each layer:
      $\partial E^k / \partial b^{(l)}(j) = \delta^{(l),k}(j)$
    /* Weight and bias update */
    Update all weights $w^{(l)}$ of each layer:
      $w^{(l)}(i,j) = w^{(l)}(i,j) - \eta \cdot \partial E^k / \partial w^{(l)}(i,j)$
    Update all biases $b^{(l)}$ of each layer:
      $b^{(l)}(j) = b^{(l)}(j) - \eta \cdot \partial E^k / \partial b^{(l)}(j)$
  end for

  /* Compute the error over the hole training dataset */
  for $k = 1$ to K **do**
    Propagate the pattern x^k through the network to obtain $y^{(L),k}$.
  end for
  $E = 1/\left(2 \cdot N^{(L)} \cdot K\right) \cdot \sum_{k=1}^{K} \sum_{j=0}^{N^{(L)}-1} \left(y^{(L),k}(j) - d^k(j)\right)^2$ /* for MSE */

  $n = n + 1$
until $E < \epsilon$ or $n > n_{max}$

Algorithm 2 The standard offline backpropagation algorithm for MLPs.

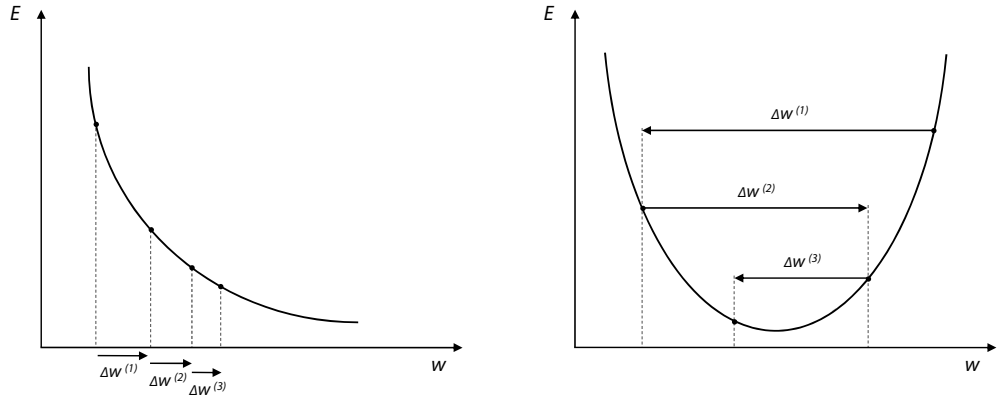
Initialize all weights w and biases b of the MLP with small random values.
Set the learning rate η to a small positive value.
$n = 1$

repeat

  /* Offline backpropagation */
  for $k = 1$ to K **do**
    /* Forward pass */
    Propagate the pattern x^k through the network to obtain $y^{(L),k}$.
    /* Backward pass */
    Compute the error e^k for the $N^{(L)}$ output neurons:
      $e^k(j) = y^{(L),k}(j) - d^k(j)$ /* for MSE */
    Compute $\delta^{(L),k}$ for the $N^{(L)}$ output neurons:
      $\delta^{(L),k}(j) = f'^{(L)}(o^{(L),k}(j)) \cdot e^k(j)$
    Compute $\delta^{(l),k}$ for the $N^{(l)}$ neurons of the hidden layers:
      $\delta^{(l),k}(j) = f'^{(l)}\left(o^{(l),k}(j)\right) \cdot \sum_{i=0}^{N^{(l+1)}-1} \delta^{(l+1),k}(i) \cdot w^{(l+1)}(j,i)$
    Compute $\partial E^k / \partial w^{(l)}(i,j)$ for the $N^{(l)}$ neurons of each layer:
      $\partial E^k / \partial w^{(l)}(i,j) = \delta^{(l),k}(j) \cdot y^{(l-1),k}(i)$
    Compute $\partial E^k / \partial b^{(l)}(j)$ for the $N^{(l)}$ neurons of each layer:
      $\partial E^k / \partial b^{(l)}(j) = \delta^{(l),k}(j)$
  end for
  /* Weight and bias update */
  Update all weights $w^{(l)}$ of each layer:
    $w^{(l)}(i,j) = w^{(l)}(i,j) - \eta \cdot \sum_{k=1}^{K} \partial E^k / \partial w^{(l)}(i,j)$
  Update all biases $b^{(l)}$ of each layer:
    $b^{(l)}(j) = b^{(l)}(j) - \eta \cdot \sum_{k=1}^{K} \partial E^k / \partial b^{(l)}(j)$

  /* Compute the error over the hole training dataset */
  for $k = 1$ to K **do**
    Propagate the pattern x^k through the network to obtain $y^{(L),k}$.
  end for
  $E = 1/\left(2 \cdot N^{(L)} \cdot K\right) \cdot \sum_{k=1}^{K} \sum_{j=0}^{N^{(L)}-1} \left(y^{(L),k}(j) - d^k(j)\right)^2$ /* for MSE */

  $n = n + 1$
until $E < \epsilon$ or $n > n_{max}$



(a) Several consecutive weight updates in the same direction (based on Figure 7.5 in [22]).

(b) Oscillation of the weight updates (based on Figure 7.6 in [22]).

Figure 2.5: Different behaviors of the weight updates during the gradient descent.

Resilient Backpropagation

The *resilient backpropagation* (RPROP), first proposed by Riedmiller and Braun [36], performs the weight updates using just the learning rate and the sign of the partial derivative of the error function with respect to each weight.

The step size $\Delta^{(l)}(i,j)$ of the weight update doesn't depend on the gradient magnitude but is increased or decreased depending on the *sign of the gradient* according to

$$\Delta^{(l)}(i,j)^{(t)} = \begin{cases} \eta^+ \cdot \Delta^{(l)}(i,j)^{(t-1)} & , \quad \text{if } \frac{\partial E^k}{\partial w^{(l)}(i,j)}^{(t-1)} \cdot \frac{\partial E^k}{\partial w^{(l)}(i,j)}^{(t)} > 0 \\ \eta^- \cdot \Delta^{(l)}(i,j)^{(t-1)} & , \quad \text{if } \frac{\partial E^k}{\partial w^{(l)}(i,j)}^{(t-1)} \cdot \frac{\partial E^k}{\partial w^{(l)}(i,j)}^{(t)} < 0 \\ \Delta^{(l)}(i,j)^{(t-1)} & , \quad \text{else} \end{cases} \tag{2.19}$$

where $0 < \eta^- < 1 < \eta^+$.

Once the step size for each weight is adapted, the update value $\Delta w^{(l)}(i,j)$ for the weight $w^{(l)}(i,j)$ in layer l is calculated as follows:

$$\Delta w^{(l)}(i,j)^{(t)} = \begin{cases} -\Delta^{(l)}(i,j)^{(t)} & , \quad \text{if } \frac{\partial E^k}{\partial w^{(l)}(i,j)}^{(t)} > 0 \\ +\Delta^{(l)}(i,j)^{(t)} & , \quad \text{if } \frac{\partial E^k}{\partial w^{(l)}(i,j)}^{(t)} < 0 \\ 0 & , \quad \text{else} \end{cases} \tag{2.20}$$

The update value $\Delta b^{(l)}(j)$ for the bias $b^{(l)}(j)$ in layer l is applied in the same way as the update values for the weights and is therefore not illustrated separately.

The benefit of this algorithm is that it accelerates learning in flat regions of the error function as well as near of a local minimum. While the gradient descent with momentum term can be performed in online and offline mode, the RPROP is an offline learning algorithm. An improved version of the RPROP learning algorithm was presented in [37].

2.5.3 Initialization of the Weights

The trainable parameters of a neural network are normally initialized with random numbers. A method to automatically choose the *distribution* of this numbers was presented by LeCun et al. They suggest to initialize the weights of a neuron (node) depending on the number of incoming connections, described as follows in [34, page 20]:

"The weights should be randomly drawn from a distribution (e.g. uniform) with mean zero and standard deviation

$$\sigma_w = m^{-1/2} \tag{2.21}$$

where m is the *fan-in* (the number of connections feeding into the node)."

2.5.4 Input Data Normalization

As stated in [34, page 16] neural networks perform better if the input vectors have a *mean* of 0 and a *standard deviation* of 1 over the training dataset. In order to fulfill this property the training dataset has to be *normalized*[4]. Normally, each component (feature) of the input vector is normalized separately across the given set of training data (see also [22, page 298] and [38, page 15]). For simplicity we regard the hole input vector as one entity shifting the mean of the entire dataset to 0 and its standard deviation to 1. This avoids some of the problems when normalizing each feature of the input vector separately (e.g. division by zero for a constant feature across the hole dataset).

First of all the mean m of the entire training set S and its standard derivation σ have to be calculated as follows:

$$m = \frac{1}{|S|} \cdot \sum_{x \in S} \frac{1}{|x|} \cdot \sum_{i=0}^{|x|-1} x(i) \tag{2.22}$$

[4]Some additional information about normalization, standardization and rescaling of the input data of a neural network can be found at: `http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html`

$$\sigma = \sqrt{\frac{1}{|S|} \cdot \sum_{x \in S} \frac{1}{|x|} \cdot \sum_{i=0}^{|x|-1} (x(i) - m)^2} \tag{2.23}$$

The elements of the normalized input vector $\hat{\mathbf{x}}$ can then be calculated as follows:

$$\hat{x}(i) = \frac{x(i) - m}{\sigma}, \quad i = 0, 1, \ldots, |x| - 1 \tag{2.24}$$

This transformation has to be applied to any input feature vector presented to the neural network, even if they stem from the validation or test dataset. Note that validation and test dataset (see Subsection 2.5.5) must be normalized using the statistics (mean and standard derivation) computed from the training dataset. Normalization has no effect on the information in the dataset, but it shifts the values to a range which is more suitable for the symmetric activation functions (e.g. tanh) of a neural network (see also Section 2.7).

2.5.5 Early Stopping as Stopping Criterion

When to stop the training is a quite difficult question when using neural networks. Minimizing the error on the training dataset often leads to *overtraining* the network, which means that the network is specialized to much on the training dataset instead of the problem in general [39]. Therefore, the authors in [40] introduced the *early stopping method*. Using it the training dataset is split into two parts: a larger one that is used to train the network and a so-called *validation set*. The error on the validation set is evaluated at regular intervals during the training. If it does not shrink for a certain number of epochs the training is terminated and the weights that minimize the error on the validation set are considered as the final ones. Figure 2.6 illustrates the early stopping method, where the vertical dashed line indicates the epoch with the best performance on the validation set.

The early stopping method has also some drawbacks. First, the validation patterns have to be removed from the training dataset. This can reduce the network's recognition performance. Second, there is no well defined rule which determines how big the validation set should be or how long one should wait for a new minimum of the error on the validation set. This can only be discovered by testing which can take a lot of time. Usually, the validation set contains of about 5% to 20% of the initial training dataset.

Some databases, such as the MNIST database (Appendix A.1), occupy a separate *test dataset*. This dataset can be used to compare the performance of

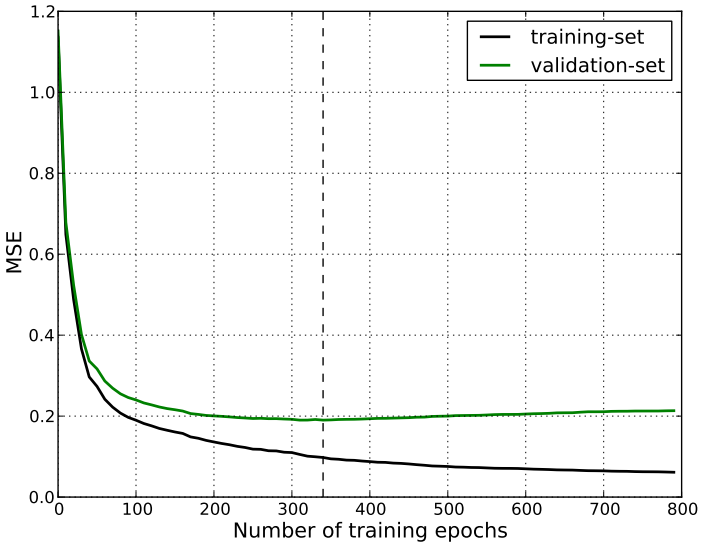


Figure 2.6: Illustration of the early stopping method. The vertical dashed line indicates the training epoch at which the smallest error on the validation set was obtained. These training curves were recorded during a face/non-face classification task.

different classification techniques[5]. The best performing parameters of a neural network should never be evaluated using this dataset. Instead, a subset of the training dataset (validation set) should be used. The performance of a neural network on the test dataset should only be determined by a *separate run* when the training is complete. Using the test dataset to evaluate the best performing parameters of a neural network leads to an effect known as "training on the testing data" (see also [41]). In this case the classifier is too much adapted on the test dataset and leads to an *excessively optimistic estimation* of the error rate.

2.6 Error Functions

The *error function* of a NN is the metric used in (2.5) to calculate the difference between an actual and the desired output of a network during the training phase. An error function generates an error E^k which considers all output neurons and an error vector $\mathbf{e}^k$ containing a separate error for each output neuron which is

[5] A summary of the error rates of different classification techniques on the MNIST test dataset can be found in Table A.1 and online at: `http://yann.lecun.com/exdb/mnist`

backpropagated through the network. In this thesis we used two different error functions: the *mean-squared-error* and the *cross-entropy*, described in detail in the following two subsections.

2.6.1 Mean-Squared-Error

The *mean-squared-error* (MSE) [22] is used in a wide range of applications, not only for NNs. It is also used in many regression and interpolation applications [27].

The overall error E^k is calculated according to

$$E^k_{mse} = \frac{1}{2 \cdot N^{(L)}} \cdot \sum_{j=0}^{N^{(L)}-1} \left(y^{(L),k}(j) - d^k(j) \right)^2 \tag{2.25}$$

where $y^{(L),k}(j)$ denotes the excitation of neuron j in the output layer L when presenting pattern k to the network and $d^k(j)$ is the desired output for this neuron. The backpropagated error vector $\mathbf{e}^k$ in (2.8) is given by

$$e^k(j) = y^{(L),k}(j) - d^k(j), \quad j = 0, 1, \ldots, N^{(L)} - 1. \tag{2.26}$$

One of its advantages is the simpleness, both of implementation and computation. The typical values for the desired output vector $\mathbf{d}^k$ is a value close to the maximum of the squasher function $f^{(L)}$ in (2.3) for the index that corresponds to the desired class and a value close to the minimum of it for all other values (e.g. for the tanh, $d^k(j) \in \{-1, +1\}$ for $j = 0, 1, \ldots, N^{(L)} - 1$).

2.6.2 Cross-Entropy

The *cross-entropy* (CE) [22] is often used when a neural network is trained to estimate the *posterior probabilities* of a class membership. Therefore, one output neuron is allocated for each class, i.e. an one-of-n coding is used. The estimated probabilities $p^k(j)$ of the network must be in the range from 0 to 1. When applying the CE to a network with multiple outputs, each of those represents the probability for the corresponding *class-affiliation*. Thus, the following constraints must be satisfied:

$$p^k(j) \geq 0, \quad j = 0, 1, \ldots, N^{(L)} - 1 \quad \text{and} \quad \sum_{j=0}^{N^{(L)}-1} p^k(j) = 1 \tag{2.27}$$

When using the CE for well defined classification tasks (as we did in this work) the corresponding desired output vector for the pattern $\mathbf{x}^k$ is given by

$$d^k(j) = \begin{cases} 1, & \text{if } \mathbf{x}^k \text{ belongs to class } j+1 \\ 0, & \text{else.} \end{cases} \tag{2.28}$$

The CE error function for a *multi-class problem* (when the network has more than one output) is defined as

$$E_{ce}^k = -\sum_{j=0}^{N^{(L)}-1} d^k(j) \cdot \ln p^k(j) \tag{2.29}$$

while for a *binary classification task* with two classes and a network with one single output it is calculated according to

$$E_{ce}^k = -\left(d^k \cdot \ln y^{(L),k} + \left(1 - d^k\right) \cdot \ln\left(1 - y^{(L),k}\right)\right). \tag{2.30}$$

The only constraint for the latter case is that the single output has to be inside the interval from 0 to 1, which can be satisfied by using the log-sigmoid function (Subsection 2.7.2) as squasher function $f^{(L)}$ in the output layer of the network. To satisfy the constraints in (2.27) for the case with multiple classes in (2.29) a so-called *softmax function* [22, 42] must be applied to the network output $\mathbf{y}^{(L),k}$ in order to gain the estimated probabilities in $\mathbf{p}^k$:

$$p^k(j) = \frac{\exp\left(y^{(L),k}(j)\right)}{\sum_{i=0}^{N^{(L)}-1} \exp\left(y^{(L),k}(i)\right)}, \quad j = 0, 1, \ldots, N^{(L)} - 1 \tag{2.31}$$

The backpropagated error in (2.8) is given by

$$e^k = y^{(L),k} - d^k \tag{2.32}$$

for the binary classification case with one output neuron and by

$$e^k(j) = p^k(j) - d^k(j), \quad j = 0, 1, \ldots, N^{(L)} - 1 \tag{2.33}$$

for the multi-class case with more than one output neuron.

In theory the CE has a clear advantage over the MSE: instead of calculating the error for each output neuron separately, the result of the CE considers the relation between all neuron's outputs. A detailed description of CE and other error functions can be found in [22, pages 194–252] while an investigation of the efficacy of CE and MSE used to train feed-forward NNs can be found in [43].

2.7 Activation Functions

An *activation function* transforms the excitement level of a neuron, which is the weighted sum of its inputs and the bias, into an output (2.3). Most activation functions have a squashing effect, therefore they are often called *squasher functions*. A nonlinear activation function is the only possibility to introduce nonlinearity into a neural network and is therefore indispensable to learn nonlinear functions[6]. The following sections give an overview over the activation functions used in this thesis.

2.7.1 Identity

Although this is the simplest activation function it is used very seldom. The weighted sum input $o^{(l)}(j)$ to neuron j in layer l in (2.3) is passed to the output without any modification, therefore

$$f(x) = x \tag{2.34}$$

as shown in Figure 2.7(a). It does not make much sense to use this function in inner layers because it is a linear function, but it may be appropriate for the output layer, as for example when using CE as error function on a multi-class classification problem (see Subsection 2.6.2). Neurons using the *identity* as activation function are called *linear*.

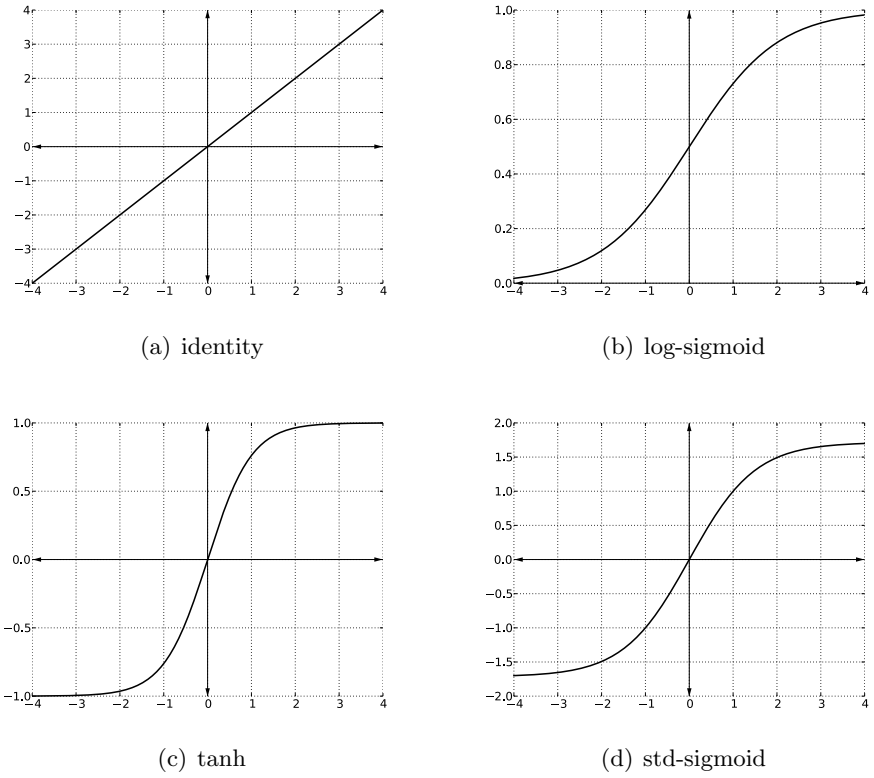
2.7.2 Log-Sigmoid

The *log-sigmoid* function [22, page 126] is nonlinear which makes it applicable for nonlinear problems. It is calculated according to

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{2.35}$$

and as Figure 2.7(b) clearly shows it is always positive which means it is asymmetric. Asymmetric squasher functions are not optimal for gradient-based back-propagation learning algorithms because, according to [34], they train slower than symmetric ones. However, the log-sigmoid scales the output between 0 and 1 which can make it a good choice for the output layer in combination with certain error functions as stated in Subsection 2.6.2.

[6] See also the online article *Why use activation functions?* at `http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-10.html`.



(a) identity

(b) log-sigmoid

(c) tanh

(d) std-sigmoid

Figure 2.7: Graphical representation of the activation functions (identity (a), log-sigmoid (b), tanh (c), and std-sigmoid (d)) used in this thesis.

2.7.3 Tangens Hyperbolicus

The *tangens hyperbolicus* (tanh) [22, page 127] is the probably most widely used squasher function for neural networks consisting of perceptrons. It is defined as

$$f(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{2.36}$$

and illustrated in Figure 2.7(c). It is both, nonlinear and symmetric and as noted in (2.36) it can be build from the exponential function. Therefore, it is a good choice for an activation function. The tanh is quite computational expensive, but for the use in neuronal networks usually an approximation of it by polynomials is precise enough (see also [44]). When using the tanh in the output layer, the normally used values for the desired output $\mathbf{d}^k$ are $+1$ and -1.

2.7.4 Std-Sigmoid

In [45] the author introduced a slight variation of the tangens hyperbolicus function called *std-sigmoid* that should be better suited for the use as activation function:

$$f(x) = 1.7159 \cdot \tanh\left(\frac{2}{3} \cdot x\right) \tag{2.37}$$

As Figure 2.7(d) shows it is a symmetric and nonlinear function with the following three properties: $f(\pm 1) = \pm 1$, the second derivative has its maximum at $x = 1$, and the effective gain is close to 1 [34]. The target values' amount is usually smaller than the function's maximum and minimum (e.g. $\{-1, +1\}$), respectively. This has the advantage that one will probably not reach the function's saturation during the training phase [34].

Chapter 3

Convolutional Neural Networks

While MLP networks, as discussed in the previous chapter, perform well on rather simple classification and recognition tasks, they have some drawbacks when it comes to real-world applications. For example, the topology of two-dimensional input patterns is completely ignored and offers therefore little or no invariance to shifting, scaling, and other forms of distortion [2]. To overcome some of this drawbacks Yann LeCun[1] introduced a new type of neural networks in the early 1990s, namely the *Convolutional Neural Networks* (CNNs) [46]. These networks were largely inspired by Hubel and Wiesel's [47] discovery of locally-sensitive, orientation-selective neurons in the cat's visual system [48].

This chapter provides a brief description of this neural network type. It starts with an introduction to CNNs and their network architecture in Section 3.1, followed by a detailed description of the mathematical model in Section 3.2. Section 3.3 gives a description how CNNs can be trained with the standard backpropagation algorithm. Finally, Section 3.4 concludes this chapter with an overview of the different CNN types proposed over the last years.

3.1 Introduction

The traditional approach for two-dimensional pattern recognition is based on a *feature extractor*, the output of which is fed into a neural network [22, pages 296–298]. This feature extractor is usually static, independent of the neural network and not part of the training procedure. It is not an easy task to find a "good" feature extractor because it is not part of the training procedure and therefore it can neither adapt to the network topology nor to the parameters generated by the training procedure of the neural network.

CNNs make this difficult task part of the network and act as a *trainable feature extractor* with some degree of shift, scale, and deformation invariance [2]. This type of neural networks is a derivative of MLP networks (Section 2.3) optimized for two-dimensional pattern recognition or classification (usually of images).

[1] `http://yann.lecun.com`

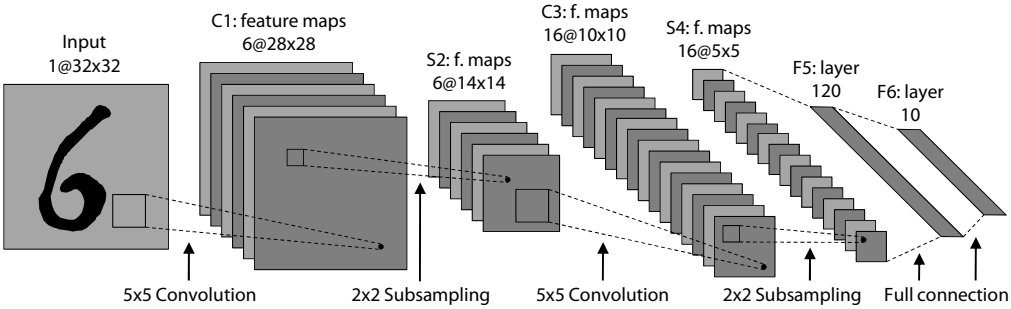


Figure 3.1: Architecture of a CNN used in this thesis (a variant of the LeNet5 network proposed by LeCun et al. in [2] for handwritten digit recognition; see also Subsection 7.1.1).

The area of applications for CNNs is widespread, they are used for handwriting recognition [1, 2], face, eye and license plate detection [3–9], and in non-vision applications such as semantic analysis [10].

A CNN consists of three different types of layers: *convolutional layers*, *subsampling layers* (optional), and *fully connected layers*. These layers are arranged in a feed-forward structure as shown in Figure 3.1. The convolutional layers are responsible for the feature extraction (edges, corners, end points or non visual features in other signals), using the two key concepts of local *receptive fields* and *shared weights*. Since the exact location of the detected features are less important than the relative position to other features, the succeeding subsampling layer performs a local averaging and subsampling. This reduces the shift and distortion sensibility of the detected features. The fully connected layer acts as a normal classifier similar to the layers in traditional MLP networks. As shown in Figure 3.1 the convolutional (C1, C3) and subsampling layers (S2, S4) are considered as 2D layers, whereas the fully connected layers (F5, F6) are considered as 1D layers.

A detailed description of the architectural concepts behind these layers is given in [2], while a brief explanation of the mathematical model of these layers together with a description how the standard backpropagation training algorithm can be applied to CNNs [34] are given in the following sections.

3.2 Mathematical Model

This section will describe the functional aspects of a CNN, where the used notation is summarized in Table 3.1. The symbol l denotes the layer index inside a CNN and goes from 1 to L, where L is the number of network layers and $M^{(l)}$ the number of feature maps inside the layer l.

Description	Symbol
Layer index	l
Number of layers	L
Number of feature maps in layer l	$M^{(l)}$
Number of neurons in layer l	$N^{(l)}$
Activation function of layer l	$f^{(l)}$
Convolutional layer	$C^{(l)}$
Subsampling layer	$S^{(l)}$
Fully connected layer	$F^{(l)}$
Input image size	$W^{(0)} \times H^{(0)}$
Input image	$\mathbf{y}_0^{(0)}$ or $\mathbf{x}$
Input image pixel at column c and row r	$y_0^{(0)}(c,r)$ or $x(c,r)$
Size of feature maps in layer l	$W^{(l)} \times H^{(l)}$
Feature map n in layer l	$\mathbf{y}_n^{(l)}$
Feature map element at column c and row r	$y_n^{(l)}(c,r)$
Output of neuron i in layer l	$y^{(l)}(i)$
Size of the convolution kernel of convolutional layer $C^{(l)}$	$k_c^{(l)} \times k_r^{(l)}$
Convolution kernel from feature map m in layer $l-1$ to feature map n in convolutional layer $C^{(l)}$	$\mathbf{w}_{m,n}^{(l)}$
Convolution kernel element at column i and row j	$w_{m,n}^{(l)}(i,j)$
Horizontal step size of the convolution in convolutional layer $C^{(l)}$	$h^{(l)}$
Vertical step size of the convolution in convolutional layer $C^{(l)}$	$v^{(l)}$
Set of feature maps in layer $l-1$ that are connected to feature map n in convolutional layer $C^{(l)}$	$V_n^{(l)}$
Size of the subsampling kernel of subsampling layer $S^{(l)}$	$s_c^{(l)} \times s_r^{(l)}$
Weight of feature map n in subsampling layer $S^{(l)}$	$w_n^{(l)}$
Weight for the connection from neuron i in layer $l-1$ to neuron j in the fully connected layer $F^{(l)}$	$w^{(l)}(i,j)$
Bias for feature map n in convolutional layer $C^{(l)}$ or subsampling layer $S^{(l)}$	$b_n^{(l)}$
Bias of neuron j in the fully connected layer $F^{(l)}$	$b^{(l)}(j)$

Table 3.1: Architectural notation for a CNN (based on Table 1 in [49]).

3.2.1 Convolutional Layer

The *convolutional layers* are the core of any CNN and take over the part of a traditional feature extractor for two-dimensional images. The advantage over traditional feature extractors is that the convolutional layers are fully integrated in the training process and can adapt to the given problem. Furthermore, convolutional layers increase the robustness of a neural network against the distortion of the input patterns [2].

A convolutional layer consists of several two-dimensional planes of neurons, the so-called *feature maps*. Each neuron of a feature map is connected to a small subset of neurons inside the feature maps of the previous layer, the so-called *receptive fields* (see also Figure 3.4). The receptive fields of neighboring neurons overlap and the weights of these receptive fields are shared through all the neurons of the same feature map. The feature maps of a convolutional

layer and its preceding layer are either *fully* or *partially connected* (either in a predefined way or in a randomized manner).

First, the convolution between each input feature map (or the input image) and the respective *convolution kernel* is computed. Corresponding to the connectivity between the convolutional layer and its preceding layer these convolution outputs are then summed up together with a trainable scalar, known as the bias term. Finally, the result is passed through an activation function (e.g. tanh). An illustration of this process is given in Figure 3.2.

The output $\mathbf{y}_n^{(l)}$ of a feature map n in a convolutional layer l is given by

$$y_n^{(l)}(c,r) = f^{(l)}\left(\sum_{m\in V_n^{(l)}}\sum_{(i,j)\in R^{(l)}} w_{m,n}^{(l)}(i,j)\cdot y_m^{(l-1)}(c\cdot h^{(l)}+i, r\cdot v^{(l)}+j) + b_n^{(l)}\right) \tag{3.1}$$

where $R^{(l)} = \left\{(i,j)\in\mathbb{N}^2 \mid 0\le i<k_c^{(l)}; 0\le j<k_r^{(l)}\right\}$, $k_c^{(l)}$ and $k_r^{(l)}$ are the width and the height of the convolution kernels $\mathbf{w}_{m,n}^{(l)}$ of layer l, and $b_n^{(l)}$ is the bias of feature map n in layer l. The set $V_n^{(l)}$ contains the feature maps in the preceding layer $l-1$ that are connected to feature map n in layer l. For example, $V_2^{(l)} = \{3,4,6\}$ means that the feature maps 3, 4, and 6 of layer $l-1$ are connected to feature map 2 of layer l. The values $h^{(l)}$ and $v^{(l)}$ describe the horizontal and vertical step size of the convolution in layer l (usually 1), while $f^{(l)}$ is the activation function of layer l.

The size of a layer's output feature maps is directly related to the size of the input feature maps, the kernel size and the step size of the convolution. The number of feature maps can be chosen independently of those parameters. The width $W^{(l)}$ and the height $H^{(l)}$ of the output feature maps of layer l are given by the formulas

$$W^{(l)} = \frac{W^{(l-1)} - \left(k_c^{(l)} - h^{(l)}\right)}{h^{(l)}} \qquad \text{and} \qquad H^{(l)} = \frac{H^{(l-1)} - \left(k_r^{(l)} - v^{(l)}\right)}{v^{(l)}} \tag{3.2}$$

where $W^{(l-1)}$ and $H^{(l-1)}$ denote the size of the feature maps of the preceding layer $l-1$ (the input feature maps) or the input image.

3.2.2 Subsampling Layer

To reduce the size of consecutive feature maps, a *subsampling layer* is usually placed between two convolutional layers. This type of layer reduces the outputs of a certain number of adjacent neurons (the *subsampling kernel*; normally a square of 2×2 neurons) of a feature map in the previous layer to one single value. Afterwards it multiplies this value with a single weight, adds a bias

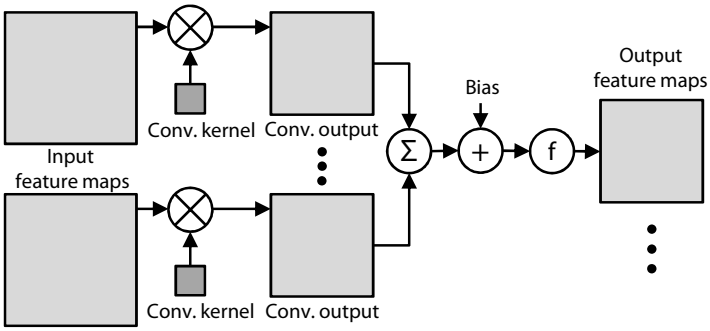


Figure 3.2: Illustration of the operating principle of a convolutional layer inside a CNN (based on Figure 2 in [49]).

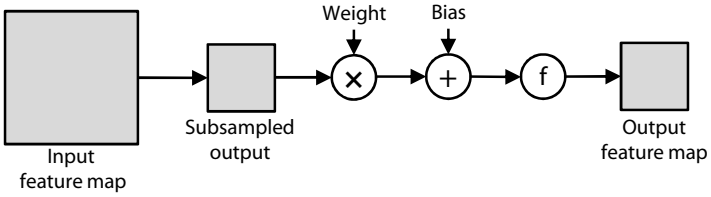


Figure 3.3: Illustration of the operating principle of a subsampling layer inside a CNN (based on Figure 3 in [49]).

and passes the result through an activation function to obtain the result of the output feature map. Subsampling layers have the same number of feature maps as the preceding convolutional layer, where each feature map of the subsampling layer is always connected to the corresponding one in the previous convolutional layer (1-to-1 connection).

The output $\mathbf{y}_n^{(l)}$ of a feature map n in the subsampling layer l is calculated according to

$$y_n^{(l)}(c,r) = f^{(l)}\left(w_n^{(l)} \cdot \sum_{(i,j)\in S^{(l)}} y_n^{(l-1)}(c \cdot s_c^{(l)} + i, r \cdot s_r^{(l)} + j) + b_n^{(l)}\right) \tag{3.3}$$

where $S^{(l)} = \left\{(i,j) \in \mathbb{N}^2 \mid 0 \leq i < s_c^{(l)}; 0 \leq j < s_r^{(l)}\right\}$, $s_c^{(l)}$ and $s_r^{(l)}$ define the width and the height of the subsampling kernel of layer l and $b_n^{(l)}$ is the bias of feature map n in layer l. The value $w_n^{(l)}$ is the weight of feature map n in layer l and $f^{(l)}$ the activation function of layer l.

Figure 3.3 illustrates the process that is performed by a subsampling layer. As subsampling layers are optional a simpler CNN can be build by replacing the consecutive convolutional and subsampling layers with a convolutional layer that uses a step size greater than 1 (for an example see Subsection 7.1.2 and [1]).

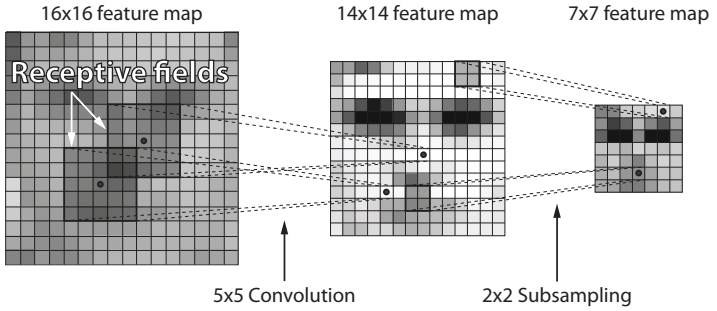


Figure 3.4: Illustration of a 5×5 convolution followed by a 2×2 subsampling inside a CNN, taken from a face/non-face classification task (based on Figure 3.5 in [50]).

Since this layer performs a subsampling of the input feature maps the size of the output feature maps is reduced according to the following two formulas

$$W^{(l)} = \frac{W^{(l-1)}}{s_c^{(l)}} \qquad \text{and} \qquad H^{(l)} = \frac{H^{(l-1)}}{s_r^{(l)}} \tag{3.4}$$

where $s_c^{(l)}$ and $s_r^{(l)}$ usually have a value of 2, which halves the size of the input feature maps.

The process of a 5×5 convolution followed by a 2×2 subsampling inside a CNN is illustrated in Figure 3.4.

3.2.3 Fully Connected Layer

After the convolutional and subsampling layers one or more *fully connected layers* follow, where the last layer acts as output layer forming the *final network response*. Although the output layer can be constructed from any type of neurons, e.g. sigmoidal neurons or radial basis function (RBF) neurons (see also [2, page 8], where Euclidean RBF units have been used), we will focus on using *sigmoidal neurons* in this thesis.

Fully connected layers are always used in a CNN and are similar to the layers in a standard MLP network (described in Section 2.3), where they act as a normal classifier. In those layers the outputs of all neurons in layer $l-1$ are connected to every neuron in layer l and the output $y^{(l)}(j)$ of neuron j is calculated as defined in (2.3) and (2.4).

If the preceding layer $l-1$ of a fully connected layer l is a convolutional or subsampling layer one has to consider the two-dimensional feature maps (matri-

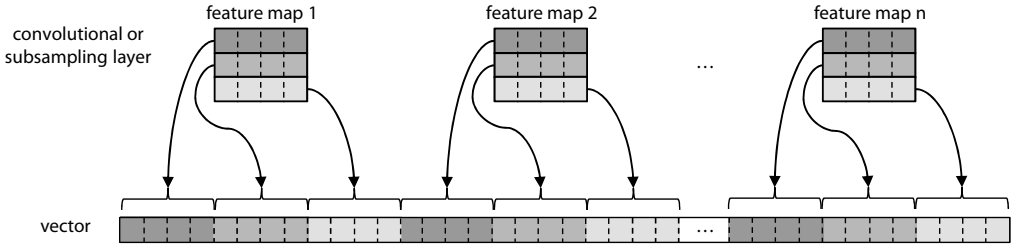


Figure 3.5: Rearrangement of the feature maps of a convolutional or subsampling layer into a single vector, so that it can be used in the succeeding (1D) fully connected layer.

ces) of those network layers as a single contiguous vector, e.g. concatenating the single rows of all feature maps to one single vector, as illustrated in Figure 3.5:

$$\begin{aligned}\left\{y_n^{(l)}(c,r);\quad n=0,1,\ldots,M^{(l)}-1, c=0,1,\ldots,W^{(l)}-1, r=0,1,\ldots,H^{(l)}-1\right\}\\ \longrightarrow \left\{y^{(l)}(j);\quad j=0,1,\ldots,N^{(l)}-1=M^{(l)}\cdot W^{(l)}\cdot H^{(l)}-1\right\}\end{aligned} \tag{3.5}$$

with

$$y^{(l)}(j) = y_{n'}^{(l)}(c',r'),\quad j=0,1,\ldots,N^{(l)}-1 \tag{3.6}$$

where

$$\begin{aligned} n' &= \left\lfloor j/\left(W^{(l)}\cdot H^{(l)}\right)\right\rfloor,\\ r' &= \left\lfloor \left(j-n'\cdot W^{(l)}\cdot H^{(l)}\right)/W^{(l)}\right\rfloor,\\ c' &= j - W^{(l)}\cdot\left(r'+n'\cdot H^{(l)}\right),\quad \text{and}\\ N^{(l)} &= M^{(l)}\cdot W^{(l)}\cdot H^{(l)}. \end{aligned}$$

An example for a possible composition of convolutional, subsampling, and fully connected layers is shown in Figure 3.1.

3.3 Network Training

Since CNNs are standard feed-forward networks they can use the same training algorithms and error functions as other neural networks of this type, e.g. the MLP networks (Section 2.3). The *backpropagation algorithm* for CNNs is almost identical to the one for standard MLP networks (see Subsection 2.5.1). However, due to the different *connection structure* and the *weight sharing* in the convolutional and subsampling layers it is a little bit more complex and therefore described in detail in the following parts. The various definitions and

Description	Symbol
Training image index in the dataset	k
Training image k or network input	$\mathbf{x}^k$ or $\mathbf{y}_0^{(0),k}$
Training image pixel at column c and row r	$x^k(c,r)$ or $y_0^{(0),k}(c,r)$
Desired output vector for sample k	$\mathbf{d}^k$
Element j of the desired output vector $\mathbf{d}^k$	$d^k(j)$
Overall error for training pattern $\mathbf{x}^k$	E^k
Backpropagated error vector for training pattern $\mathbf{x}^k$	$\mathbf{e}^k$
Element j of the backpropagated error vector $\mathbf{e}^k$	$e^k(j)$
Derivate of the activation function $f^{(l)}$ of layer l	$f'^{(l)}$
Weighted sum input to neuron (c,r) in feature map n of a convolutional or subsampling layer l ($C^{(l)}$, $S^{(l)}$)	$o_n^{(l),k}(c,r)$
Weighted sum input to neuron j in a fully connected layer $F^{(l)}$	$o^{(l),k}(j)$
Output of neuron (c,r) in feature map n of a convolutional or subsampling layer l ($C^{(l)}$, $S^{(l)}$)	$y_n^{(l),k}(c,r)$
Output of neuron j in a fully connected layer $F^{(l)}$	$y^{(l),k}(j)$
Error sensitivity of neuron (c,r) in feature map n of a convolutional or subsampling layer l ($C^{(l)}$, $S^{(l)}$)	$\delta_n^{(l),k}(c,r)$
Error sensitivity of neuron j in a fully connected layer $F^{(l)}$	$\delta^{(l),k}(j)$

Table 3.2: Notation for the CNN backpropagation (based on Table 2 in [49]).

notations used in this section are summarized in Table 3.2. The symbol k denotes the index of the training image $\mathbf{x}^k$ in the dataset. Parts of the equations in this section are based on the work in [49] and [50].

In order to minimize the error on the training dataset using the *gradient descent method* the *error gradient* has to be calculated. This gradient can then be used to train the CNN applying the (online or batch) backpropagation algorithm (Subsection 2.5.1) or a variant of it, e.g. gradient descent with momentum term or resilient backpropagation (Subsection 2.5.2).

3.3.1 Error Sensitivity Computation

The error gradient can be computed through the *error sensitivity*, which is defined as the partial derivate of the error function (Section 2.6) with respect to the weighted sum input to a neuron.

For a neuron (c,r) in feature map n of a convolutional or subsampling layer l, the error sensitivity is given by

$$\delta_n^{(l),k}(c,r) = \frac{\partial E^k}{\partial o_n^{(l),k}(c,r)} \tag{3.7}$$

while for a neuron j in a fully connected layer l, it is defined as

$$\delta^{(l),k}(j) = \frac{\partial E^k}{\partial o^{(l),k}(j)} \tag{3.8}$$

where $o_n^{(l),k}(c,r)$ and $o^{(l),k}(j)$ denote the weighted sum input to a neuron in a 2D (convolutional or subsampling) layer and 1D (fully connected) layer, respectively.

Using the *chain rule of differentiation* [22, pages 142–143] and depending on the composition of the different layers in a CNN the error sensitivity can be calculated according to the following equations:

- **Output layer (L)**

  The error sensitivity $\delta^{(L),k}(j)$ of the neurons in the output layer L can be expressed in the same way as for a MLP network using equation

  $$\delta^{(L),k}(j) = f'^{(L)}\left(o^{(L),k}(j)\right) \cdot e^k(j), \quad j = 0, 1, \ldots, N^{(L)} - 1 \tag{3.9}$$

  where the backpropagated error $e^k(j)$ depends on the used error function (Section 2.6):

  – MSE case:

  $$e^k(j) = y^{(L),k}(j) - d^k(j) \tag{3.10}$$

  – CE case:

  $$e^k(j) = p^k(j) - d^k(j) \tag{3.11}$$

- **Other fully connected layers ($l < L$)**

  In the other (hidden) fully connected layers $l < L$ the error sensitivity $\delta^{(l),k}(j)$ of the neurons can be calculated according to

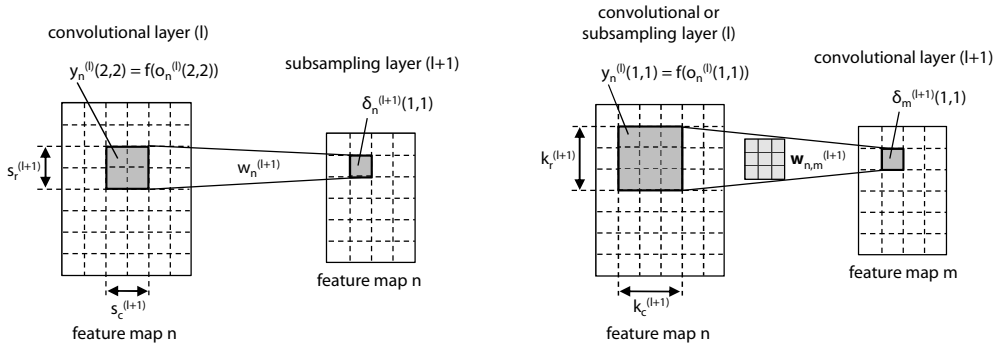
  $$\delta^{(l),k}(j) = f'^{(l)}\left(o^{(l),k}(j)\right) \cdot \sum_{i=0}^{N^{(l+1)}-1} \delta^{(l+1),k}(i) \cdot w^{(l+1)}(j,i) \tag{3.12}$$

  where $j = 0, 1, \ldots, N^{(l)} - 1$.

- **A convolutional or subsampling layer followed by a fully connected layer**

  For a (2D) convolutional or subsampling layer l followed by a (1D) fully connected layer $l+1$ the error sensitivity $\delta_n^{(l),k}(c,r)$ is given by

  $$\delta_n^{(l),k}(c,r) = f'^{(l)}\left(o_n^{(l),k}(c,r)\right) \cdot \sum_{i=0}^{N^{(l+1)}-1} \delta^{(l+1),k}(i) \cdot w^{(l+1)}(j,i) \tag{3.13}$$



(a) A convolutional layer followed by a (2×2) subsampling layer (based on Figure 3.7 in [50]).

(b) A convolutional or subsampling layer followed by a (3×3) convolutional layer (based on Figure 3.6 in [50]).

Figure 3.6: Illustration of the error backpropagation for a convolutional layer followed by a subsampling layer (a) and a convolutional or subsampling layer followed by a convolutional layer (b).

where

$$
\begin{aligned}
n &= 0, 1, \ldots, M^{(l)} - 1, \\
c &= 0, 1, \ldots, W^{(l)} - 1, \\
r &= 0, 1, \ldots, H^{(l)} - 1, \quad \text{and} \\
j &= W^{(l)} \cdot \Big(n \cdot H^{(l)} + r \Big) + c.
\end{aligned}
$$

- **A convolutional layer followed by a subsampling layer**

  The error sensitivity $\delta_n^{(l),k}(c, r)$ for the neurons in a convolutional layer l followed by a subsampling layer $l + 1$ can be obtained as

$$
\delta_n^{(l),k}(c, r) = f'^{(l)} \left(o_n^{(l),k}(c, r) \right) \cdot \delta_n^{(l+1),k}(c', r') \cdot w_n^{(l+1)} \tag{3.14}
$$

  where

$$
\begin{aligned}
n &= 0, 1, \ldots, M^{(l)} - 1, \\
c &= 0, 1, \ldots, W^{(l)} - 1, \\
r &= 0, 1, \ldots, H^{(l)} - 1, \\
c' &= \left\lfloor c / s_c^{(l+1)} \right\rfloor, \quad \text{and} \\
r' &= \left\lfloor r / s_r^{(l+1)} \right\rfloor.
\end{aligned}
$$

Figure 3.6(a) illustrates the relation of the respective variables in this equation.

- **A convolutional or subsampling layer followed by a convolutional layer**

  In the case where a convolutional or subsampling layer l is followed by a convolutional layer $l+1$, illustrated in Figure 3.6(b), the error sensitivity $\delta_n^{(l),k}(c,r)$ is given by

$$\delta_n^{(l),k}(c,r) = f'^{(l)}\left(o_n^{(l),k}(c,r)\right) \cdot \sum_{m \in U_n^{(l)}} \sum_{(i,j) \in R^{(l)}(c,r)} \delta_m^{(l+1),k}(i,j) \cdot w_{n,m}^{(l+1)}(c - i \cdot h^{(l+1)}, r - j \cdot v^{(l+1)}) \tag{3.15}$$

  where

$$\begin{aligned}
n &= 0, 1, \ldots, M^{(l)} - 1, \\
c &= 0, 1, \ldots, W^{(l)} - 1, \\
r &= 0, 1, \ldots, H^{(l)} - 1, \\
U_n^{(l)} &= \left\{ m \in \mathbb{N} \mid 0 \leq m < M^{(l+1)} \wedge n \in V_m^{(l+1)} \right\}, \\
R^{(l)}(c,r) &= \left\{ (i,j) \in \mathbb{N}^2 \mid i_1 \leq i < i_2;\ j_1 \leq j < j_2 \right\}, \\
i_1 &= \max\left(\left\lfloor \left(c - \left(k_c^{(l+1)} - h^{(l+1)} \right) \right) / h^{(l+1)} \right\rfloor, 0 \right), \\
i_2 &= \min\left(W^{(l+1)}, \left\lfloor c / h^{(l+1)} \right\rfloor + 1 \right), \\
j_1 &= \max\left(\left\lfloor \left(r - \left(k_r^{(l+1)} - v^{(l+1)} \right) \right) / v^{(l+1)} \right\rfloor, 0 \right), \quad \text{and} \\
j_2 &= \min\left(H^{(l+1)}, \left\lfloor r / v^{(l+1)} \right\rfloor + 1 \right).
\end{aligned}$$

  The symbol $U_n^{(l)}$ denotes the set of feature maps in the convolutional layer $l+1$ connected to the feature map n in layer l while $R^{(l)}(c,r)$ describes the set of single neurons in these maps connected to the neuron (c,r) in feature map n of layer l.

3.3.2 Error Gradient Computation

After determining the error sensitivity of each neuron in all layers, the *error gradient* can be calculated. Using the error sensitivity and the output of the

corresponding neuron in the preceding layer it can be obtained according to the following equations:

- **Output layer ($l = L$) or other fully connected layers ($l < L$)**
  - For the weights $w^{(l)}(i,j)$

$$\frac{\partial E^k}{\partial w^{(l)}(i,j)} = \delta^{(l),k}(j) \cdot y^{(l-1),k}(i) \tag{3.16}$$

    where $i = 0, 1, \ldots, N^{(l-1)} - 1$ and $j = 0, 1, \ldots, N^{(l)} - 1$.
  - For the biases $b^{(l)}(j)$

$$\frac{\partial E^k}{\partial b^{(l)}(j)} = \delta^{(l),k}(j) \tag{3.17}$$

    where $j = 0, 1, \ldots, N^{(l)} - 1$.

- **A fully connected layer after a convolutional or subsampling layer**
  - For the weights $w^{(l)}(i,j)$

$$\frac{\partial E^k}{\partial w^{(l)}(i,j)} = \delta^{(l),k}(j) \cdot y_n^{(l-1),k}(c,r) \tag{3.18}$$

    where

$$\begin{aligned}
&i = 0, 1, \ldots, N^{(l-1)} - 1 = M^{(l-1)} \cdot W^{(l-1)} \cdot H^{(l-1)} - 1,\\
&j = 0, 1, \ldots, N^{(l)} - 1,\\
&n = \left\lfloor i / \left(W^{(l-1)} \cdot H^{(l-1)} \right) \right\rfloor,\\
&r = \left\lfloor \left(i - n \cdot W^{(l-1)} \cdot H^{(l-1)} \right) / W^{(l-1)} \right\rfloor, \quad \text{and}\\
&c = i - W^{(l-1)} \cdot \left(r + n \cdot H^{(l-1)} \right).
\end{aligned}$$

  - For the biases $b^{(l)}(j)$

$$\frac{\partial E^k}{\partial b^{(l)}(j)} = \delta^{(l),k}(j) \tag{3.19}$$

    where $j = 0, 1, \ldots, N^{(l)} - 1$.

- **Subsampling layer (after a convolutional layer)**
  - For the weights $w_n^{(l)}$

$$\frac{\partial E^k}{\partial w_n^{(l)}} = \sum_{(c,r)\in S^{(l)}} \delta_n^{(l),k}(c',r') \cdot y_n^{(l-1),k}(c,r) \tag{3.20}$$

    where

$$\begin{aligned} n &= 0,1,\ldots,M^{(l)}-1, \\ S^{(l)} &= \left\{(c,r)\in\mathbb{N}^2 \mid 0\le c<W^{(l-1)}; 0\le r<H^{(l-1)}\right\}, \\ c' &= \left\lfloor c/s_c^{(l)} \right\rfloor, \quad \text{and} \\ r' &= \left\lfloor r/s_r^{(l)} \right\rfloor. \end{aligned}$$

  - For the biases $b_n^{(l)}$

$$\frac{\partial E^k}{\partial b_n^{(l)}} = \sum_{(c,r)\in S^{(l)}} \delta_n^{(l),k}(c,r) \tag{3.21}$$

    where

$$\begin{aligned} n &= 0,1,\ldots,M^{(l)}-1, \quad \text{and} \\ S^{(l)} &= \left\{(c,r)\in\mathbb{N}^2 \mid 0\le c<W^{(l)}; 0\le r<H^{(l)}\right\}. \end{aligned}$$

- **Convolutional layer (after a subsampling layer or another convolutional layer)**
  - For the weights $w_{m,n}^{(l)}(i,j)$

$$\frac{\partial E^k}{\partial w_{m,n}^{(l)}(i,j)} = \sum_{(c,r)\in R^{(l)}} \delta_n^{(l),k}(c,r)\cdot y_m^{(l-1),k}(c\cdot h^{(l)}+i, r\cdot v^{(l)}+j) \tag{3.22}$$

    where for all $m\in V_n^{(l)}$

$$\begin{aligned} n &= 0,1,\ldots,M^{(l)}-1, \\ i &= 0,1,\ldots,k_c^{(l)}-1, \\ j &= 0,1,\ldots,k_r^{(l)}-1, \quad \text{and} \\ R^{(l)} &= \left\{(c,r)\in\mathbb{N}^2 \mid 0\le c<W^{(l)}; 0\le r<H^{(l)}\right\}. \end{aligned}$$

It has to be noted that $\mathbf{y}_0^{(0),k}$ refers to the input sample $\mathbf{x}^k$ of the training dataset (see also Table 3.2).

– For the biases $b_n^{(l)}$

$$\frac{\partial E^k}{\partial b_n^{(l)}} = \sum_{(c,r)\in R^{(l)}} \delta_n^{(l),k}(c,r) \tag{3.23}$$

where

$$n = 0, 1, \ldots, M^{(l)} - 1, \quad \text{and}$$
$$R^{(l)} = \left\{ (c,r) \in \mathbb{N}^2 \mid 0 \leq c < W^{(l)}; 0 \leq r < H^{(l)} \right\}.$$

The backpropagation formulas for the convolutional and subsampling layer look rather complicated and quite different from those of the fully connected layers. But in fact, they are not. To calculate gradients in all three types of layers the error is backpropagated to neuron j in layer l from the following layer $l+1$ by calculating the dot product of all outgoing weights and the local gradients (δ values) of the neurons in layer $l+1$ connected to them. This value is first multiplied by the weighted sum of neuron's j inputs, passed through the derivation of the activation function and then multiplied by the output of neuron i in layer $l-1$. Note that neuron i is connected to neuron j over the connection that uses weight $w^{(l)}(i,j)$. This procedure is done for every incoming connection in layer l in order to calculate a gradient for every one of them. The biggest difference between the fully connected and convolutional layer is, that in the latter one the calculated values of all connections that use the same shared weight have to be summed together to form one single gradient, because of the weight sharing. The calculation of gradients of different feature maps can be done independently, because weights are not shared over different feature maps, not on the input nor on the output side.

3.4 Types of CNNs

As for neural networks in general to find the best performing configuration of a CNN, which includes the network structure, the distribution of the randomly initialized weights, the learning algorithm, etc. is a difficult task. The peculiarities of a CNN add even more variables to this task, like the kernel and step size of the convolution, the composition of convolutional, subsampling and fully connected layers and the number of feature maps in the single layers. All those parameters have to be adapted to a given problem in general and to the used

training set in specific. This is definitely a long-winded process that requires a lot of testing. For some well known benchmarks, as for example for the MNIST database (Appendix A.1) several well working networks have been published. Figure 3.1 shows a variation of the LeNet5, published by Yann LeCun in [2] for handwritten digit recognition (see also Subsection 7.1.1). It has been optimized for the MNIST database and achieves quite good results on it.

Basically CNNs can be divided in two major groups: CNNs that use subsampling layers and the ones that don't. Networks with subsampling layers have usually three convolutional layers with a subsampling layer in between the first and second as well as between the second and third convolutional layer. After the last convolutional layer follows one or more fully connected layers. The typical step size of the convolution is one and the major input size reduction is done by the subsampling layers. The LeNet5 network is one of this type. Networks without subsampling layers normally contain two or three convolutional layers and one or two following fully connected layers. The typical step size of the convolution in such a network is two. Using this step size a single convolutional layer has a similar effect as a convolutional layer with step size one followed by a 2×2 subsampling layer. Omitting the subsampling layers and increasing the step size of the convolution yields to some speedup due to the lower number of calculations, but doesn't make a huge difference. The memory consumption however is significantly lower. An established example of this kind of network has been published by Simard et al. in [1] as one of the best performing neural networks on the MNIST database (see also Subsection 7.1.2).

Another important differentiator of CNNs is the connection type of the subsampling layer and the following convolutional layer or two subsequent convolutional layers, respectively. They can either be fully connected (every feature map of layer l is connected to every feature map of layer $l-1$), as for example in the network proposed by Simard et al. in [1], or partially connected. If they are partially connected the connectivity can either be predefined, as for example in the LeNet5 [2], or the connections are chosen randomly [51]. Our tests showed, that sparse connections usually lead to better results (see also Subsection 7.1.4). However, using sparse connections implies the use of more feature maps in the network which increases memory requirements and extends execution time.

Chapter 4

Compute Unified Device Architecture

This chapter provides a detailed description of NVIDIA's parallel computing architecture CUDA, used in this thesis to accelerate training and classification of arbitrary CNNs.

Section 4.1 starts with a brief history of GPGPU computing, which is followed by a short introduction to CUDA in Section 4.2. A description of the current CUDA hardware and an outlook on NVIDIA's future hardware are given in Section 4.3 and Section 4.4, respectively. Section 4.5 describes the threading model of CUDA and how the threads are grouped and mapped to the hardware. An introduction to the two CUDA programming APIs, the CUDA runtime and driver API, is given in Section 4.6. Finally, Section 4.7 describes the various memory types on a CUDA device.

4.1 A Brief History of GPGPU Computing

Before the 1980s graphics cards were not much more than simple *signal converters* that translated signals calculated by the CPU into signals readable by a monitor. After 1980 things began to change: hardware developers started to implement two-dimensional (2D) primitives in hardware in order to move computation load away from the CPU to some sort of *fixed function co-processors*. Some of the first companies to build such a solution for the mass-market were Commodore and IBM with their Amiga [52] and 8514 graphics system [53], respectively. They were able to offload common 2D drawing operations (such as line drawing, area filling, and bit-blitting[1][54]) to their fixed function co-processors and to free the CPU for other tasks.

In the 1990s manufacturing capabilities improved and so did the graphics chips. Nearly every computer was equipped with a *2D accelerator*. The first APIs to program such hardware, like Microsoft's WinG graphics library [55] and their later DirectDraw interface [56], emerged. Those were mainly used

[1] Bit-blitting or BitBlit (Bit-Block Image Transfer) for short describes a computer graphics operation in which several bitmaps are combined into one using a raster operator.

to accelerate 2D computer games. In the mid 1990s 3D computer games became popular and the huge amount of calculations for the three-dimensional (3D) images often overloaded the CPUs of these days. This resulted in a high demand for *3D accelerator* cards which now entered the mass-market. These cards shifted some parts of the calculations, like the *Z-buffering*[2][57, 58], *texture filtering* [58], and the *texture mapping* [58] to the graphics accelerator. Because powerful 3D accelerators resulted in big chips the first graphics cards combining 2D and 3D acceleration (for example the Matrox Mystique and ATI Rage) were outperformed by dedicated 3D accelerators without 2D capabilities, like the 3dfx Voodoo Graphics. Together with the 3D accelerators new languages to program them appeared, for example the open standard OpenGL [59, 60] and Microsoft's DirectX [61, 62], which both survived so far. With the demand for computer games with more realistic graphics single pipelined graphics reached their limits quite soon. Therefore in 1998 3dfx introduced the first 3D accelerator, the 3dfx Voodoo[2], with two texture units which were able to work in parallel. With the advancement of chip manufacturing at the end of this decade 3D acceleration with high performance could be included into chips with 2D capabilities and the dedicated 3D accelerators disappeared. During this decade also the first non-graphical applications ranging from robotics and artificial intelligence to computer security using graphics hardware emerged [63–65].

Around the turn of the millennium the development of 3D graphics cards advanced very quickly, even faster than the CPU development. Therefore CPUs became the bottleneck in 3D computer games. To overcome this problem NVIDIA introduced it's GeForce 256 in 1999. This graphics chip was the first that has been called *Graphics Processing Unit* (GPU). It deserved this name because for the first time the *transforming* (mapping the world coordinates of an 3D object to 2D coordinates on the screen) and *lighting* of a scene could be calculated by the graphics accelerator (called hardware T&L). However, this functionality was still implemented in fixed function hardware units, but it worked very well for this particular operations.

In 2001 NVIDIA introduced the first GPU, the GeForce 3 series, with programmable units: the *pixel* and *vertex shader*. For this scope new shading languages, like GLSL (OpenGL Shading Language) [66] for OpenGL, HLSL (High Level Shading Language) [67] for DirectX and NVIDIA's Cg (C for graphics) [68, 69] for OpenGL and DirectX, were introduced. Although the first programmable shader units did support neither branches nor loops, researchers implemented non-graphical applications on this GPUs [70–72]. This introduced

[2]In computer graphics, Z-buffering describes a technique to manage the depth coordinates of images in three-dimensional graphics.

the new term *GPGPU computing* or *General-Purpose computation on Graphics Processing Units*[3].

Another big step toward making GPUs more suitable for general purpose processing was the Shader Model Standard 3.0 [73, 74] in 2004. This standard, first supported by NVIDIA's GeForce 6 series, included *branches* and *loops*. However, the arithmetic units were still organized in a graphics oriented way as pixel and vertex shaders.

This was changed by Shader Model 4.0 [75], introduced in 2006 alongside with NVIDIA's GeForce 8 series. This version requested hardware with *unified shader units*, which are able to perform pixel operations as well as vertex operations. These are the first fully programmable ALUs (Arithmetic Logic Units) deployed in GPUs. For the first time since the introduction of 3D accelerators the graphics card was not restricted to triangles as data primitives. The turn-away from two different types of shaders and the advancements in microchip manufacturing made GPUs *massively parallel homogeneous processors*. Housing hundreds of ALUs they are well suited for a lot of complex computations [76]. Together with those new GPUs new programming languages designed for GPGPU computing, like Brook from the Stanford University [77], NVIDIA's parallel programming language CUDA (Compute Unified Device Architecture) [11], and the later OpenCL (Open Computing Language) [78, 79], appeared. Also Microsoft introduced a new API, the DirectCompute interface [80–82], as part of the DirectX 11 collection to take advantage of the massively parallel processing power of modern GPUs for non-graphical applications. These languages are much easier to use for programmers without graphics programming knowledge than the shading languages.

With the release of CUDA in 2007, NVIDIA simplified the GPGPU programming and many researchers started to accelerate their applications using the GPU[4]. More details about NVIDIA's general purpose parallel computing architecture CUDA will be explained in the following sections.

Recently the first GPUs supporting Shader Model 5.0 [82], the AMD Radeon HD 5000 series, have been released. The new standard includes some features that are useful for GPGPU computing, as for example a cache that can be shared between all ALUs.

Figure 4.1 illustrates the development of GPUs and CPUs over the last few years. In 2003, when the first GPGPU applications using the programmable

[3] A growing community around GPGPU computing with lots of informations and news can be found at: `http://gpgpu.org`

[4] An overview hosting hundreds of applications which have been accelerated using the CUDA parallel computing architecture can be found at: `http://www.nvidia.com/object/cuda_home.html`

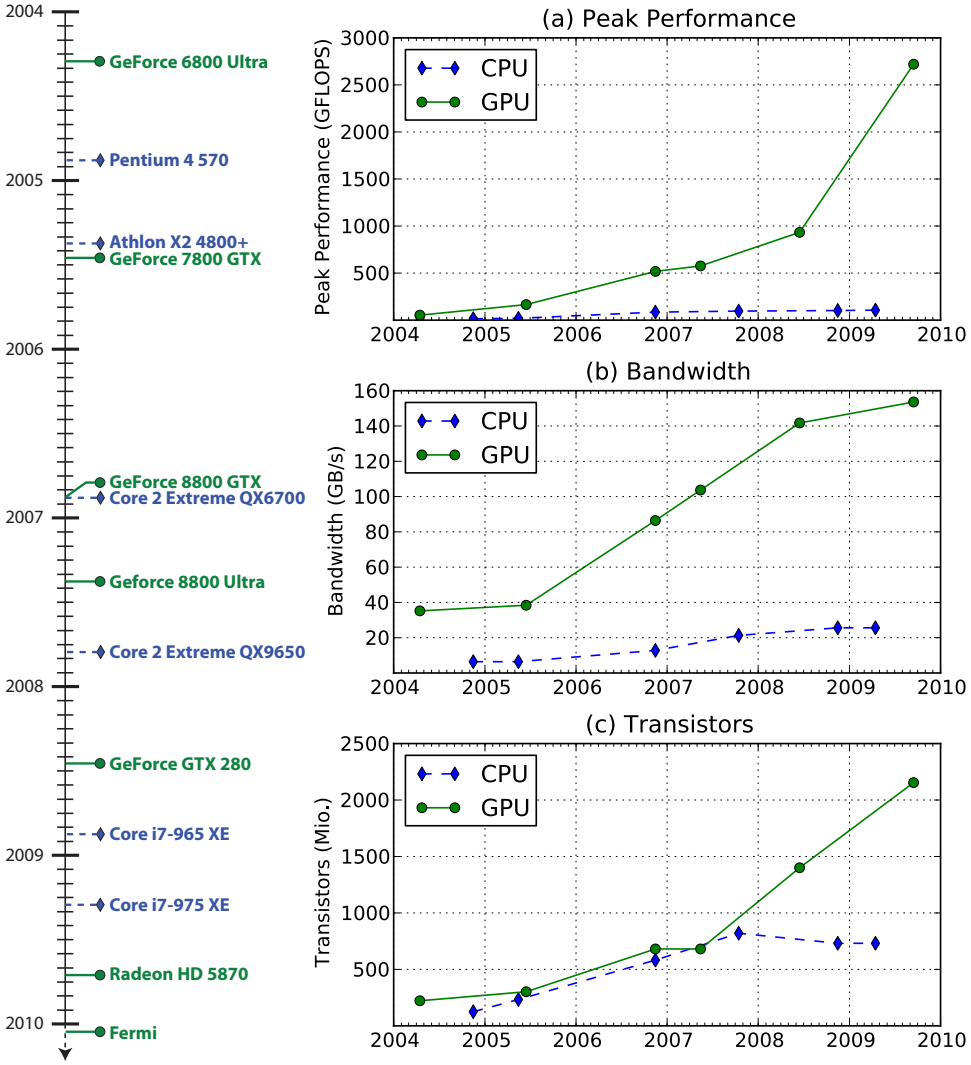
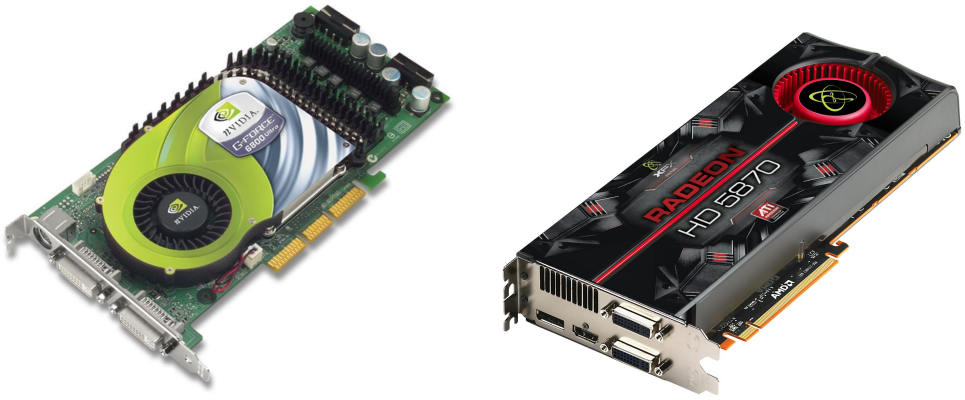


Figure 4.1: The development of GPUs and CPUs over the last few years.

pixel and vertex shaders appeared, GPUs already had a slightly higher *peak performance* than CPUs. Although the number of transistors in a single processor grew with approximately the same speed until 2007, as shown in Figure 4.1(c), the peak performance of GPUs grew much faster (see Figure 4.1(a)). The main reason for that is that new transistors of CPUs were mainly spend on bigger caches and better branch prediction. This can speed up real-world applications, but it has no impact on the peak performance. The additional transistors on GPUs were mainly used to increase the number of ALUs. If we compare the hardware of 2004 and 2007, the maximum number of single precision floating-point calculations in parallel on the CPU raised from 4 (on a



(a) NVIDIA GeForce 6800 Ultra (2004) (b) AMD Radeon HD 5870 (2009)

Figure 4.2: Two graphics cards capable for GPGPU computing.

single Pentium 4) to 16 (on a Core 2 Quad). In this period the calculation units on GPU raised from 16 pixel shaders on a GeForce 6800 (see Figure 4.2(a)) to 320 unified shaders on a Radeon HD 2900. After that date, CPUs did not get bigger up to now and made only minor improvements on peak performance and bandwidth. In contrast to that GPUs got a massive boost in terms of chip size and computational power. The maximum number of shader units found on a 2009 GPU is 1600 on a Radeon HD 5870 (see Figure 4.2(b)). Another important performance index is the *memory bandwidth* between a processor and its main memory. Figure 4.1(b) shows that GPUs always had a bigger bandwidth than CPUs. There are two reasons for that: graphics processing needs a very high memory throughput to be able to draw images fast. Furthermore, GPUs don't have such a technically mature cache structure as CPUs, which could help to hide a low bandwidth.

4.2 Introduction to CUDA

The term *CUDA* (Compute Unified Device Architecture) describes NVIDIA's general purpose parallel computing architecture and was developed to use their GPUs for non-graphical applications [11, 83, 84]. CUDA is supported by all NVIDIA GPUs featuring Shader Model 4.0 or higher (all GPUs since the G80, introduced in 2006 on the GeForce 8800 GTX). Therefore, devices supporting CUDA can be found in a lot of "ordinary" computers.

A CUDA-capable device can be either directly soldered on the mainboard, a PCI-Express card or a server rack. Most on-board and PCI-Express models (the GeForce and Quadro series) can be used as ordinary graphics cards while

some cards as well as the server racks are build for *High Performance Computing* (HPC) only (the Tesla series).

CUDA runs on Windows, Mac OS, and Linux and comes with a software environment (programming language, compiler, profiler, etc.) that simplifies GPGPU programming. The CUDA language itself is based on C, containing some C++ extensions like templates as well as some special instructions and keywords to control multiprocessing on the GPU [11]. This and the big amount of documentation and samples[5] maintain a low learning curve and enjoys therefore a high degree of acceptance by the developers.

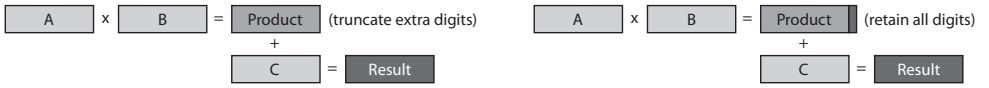
4.3 Current CUDA Hardware

A current CUDA device [83, 84] consists of a GPU as well as some memory (256 MB to 4 GB in 2009), which can either be dedicated or shared with the CPU. The GPU houses several (1 to 30) *streaming multiprocessors* (SMs). The SMs are organized in *clusters* of 2 or 3 SMs (depending on the model), which share the same *texture unit*. However, these SMs are independent of each other and can run different code branches as well as even different programs. Each SM contains eight *scalar processor* (SP) cores (also called *streaming processor* cores or in the graphics programmer jargon *shader units*) and two *special function units* (SFUs) as well as a *load/store unit* that can perform up to 16 parallel memory operations.

The SP cores perform the fundamental operations (like floating-point, integer, comparison, and conversion operations), while the SFUs can calculate either a *transcendental function*[6] [85, page 26] (like sin, cos or log) or four single precision floating-point (FP) multiplications. The SP cores and the SFUs have to share a single *dispatch unit*, but can be used in parallel as follows: the SP cores and the SFUs can issue a new instruction every fourth cycle (they either execute the same code four times in serial on different data or have to stall for some clocks). If the dispatcher is able to issue a MAD (a floating-point multiplication followed by an addition on the result; see also Figure 4.3(a)) every four cycles and a multiplication two clocks after each MAD then (and only then) the SP cores and SFUs are fully occupied. This proceeding is called *dual-issue mode* [84] and is illustrated in Figure 4.4. The dual-issue mode boosts the theoretical peak performance by 50% and the graphics performance about 20%.

[5] Available at: `http://www.nvidia.com/object/cuda_home.html`

[6] A transcendental function is a function that is not algebraic; that is, it cannot be constructed using only algebraic operations (such as addition, subtraction, multiplication, division, and taking roots).



(a) MAD operation (b) FMA operation

Figure 4.3: Illustration of the MAD and FMA operation (based on the figure on page 14 in [86]).

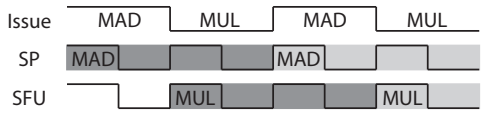


Figure 4.4: Illustration of the dual-issue mode (based on Figure 6 in [84]).

Double precision floating-point calculations are only supported on the newer GPUs (since the GT200 in 2008). In double precision every SM can calculate one FMA (Fused Multiplication Addition) per clock. FMA [87, pages 151–179] is a MAD operation that does not perform any truncation of the intermediate result of the multiplication (see Figure 4.3(b)). Using double precision on current CUDA hardware will drop the theoretical peak performance to one twelfth of single precision peak performance.

Furthermore, *shared memory* and *caches* are placed on the GPU. The shared memory of each SM has a size of 16 KB and has to be controlled by the programmer, which means the data has to be loaded to it manually (*scratch-pad* or *local store memory*). To avoid cache coherence problems the memory types featuring automatically managed caches, the *constant* and *texture memory*, are read only in CUDA. The constants and texture data are written to the GPU memory by the CPU and loaded to the caches of the single SM if needed. The size of the constant and texture cache is 8 KB each one per SM, where the total amount of constant memory in use is limited to 64 KB. The *register file* contains either 8K (G80–G98) or 16K (GT200 and newer) 32-bit registers per SM.

A schematic visualization of a G80 chip is shown in Figure 4.5. This GPU consists of 8 SM clusters each of which houses two SMs. Each of the DRAM memory interface is 64-bit wide to reach a total memory bandwidth of 384-bit. Table 4.1 summaries most of the properties of a G80 (2006) and GT200 (2008) chip, while a detailed description of the GT200 architecture can be found in [84].

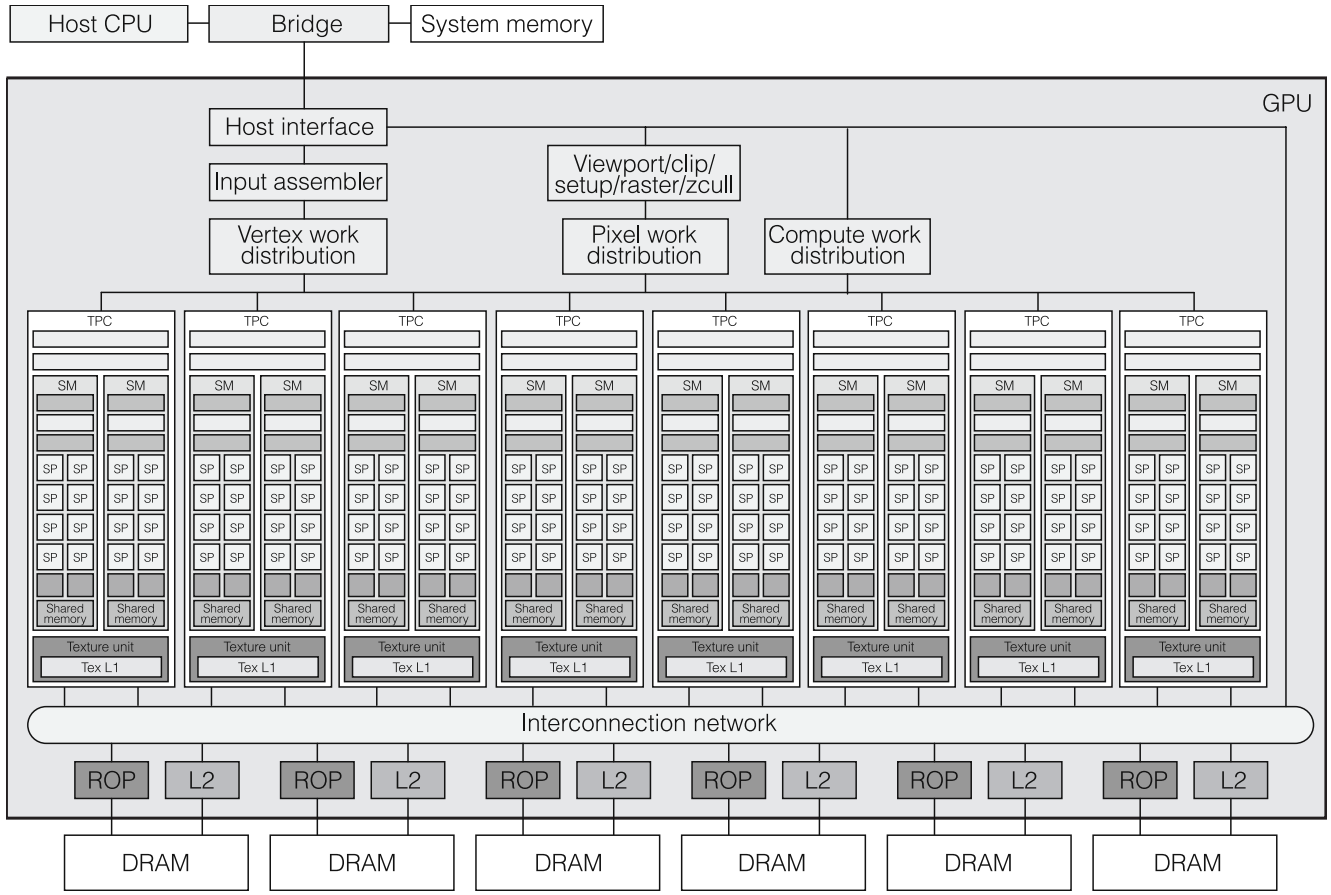


Figure 4.5: Illustration of the NVIDIA G80 architecture (from [83], Figure 1). TPC: Texture/Processor Cluster; SM: Streaming Multiprocessor; SP: Streaming Processor; Tex: Texture; ROP: Raster Operation Processor.

GPU	G80 (2006)	GT200 (2008)	Fermi (2010)
Transistors	681 million	1.4 billion	3.0 billion
SMs	16	30	16
SP (or CUDA) Cores	128	240	512
DP FP Capability (ops/clock)	None	30 FMA	256 FMA
SP FP Capability (ops/clock)	128 MAD + 128 MUL	240 MAD + 240 MUL	512 FMA
SFUs (per SM)	2	2	4
Warp Schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	48 or 16 KB *
L1 Cache (per SM)	None	None	16 or 48 KB *
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Full IEEE 754-2008 Support	No	No	Yes
Load/Store Address Width	32-bit	32-bit	64-bit

* Configurable by the programmer.

Table 4.1: Properties summary of current (G80 and GT200) and future (Fermi) CUDA hardware.

4.4 Future CUDA Hardware

The upcoming hardware of NVIDIA will be somehow different and better suited for HPC. The new chip is called *Fermi*[7] [86, 88–91], supports Shader Model 5.0 and houses 16 SMs. For the first time since the introduction of CUDA in 2007, NVIDIA fundamentally changes the layout of the SM units. The number of SP cores in one SM is raised from 8 to 32. The SP cores are now called *CUDA cores* and organized in two clusters of 16 units each. The number of SFUs per SM increases from two to four, while the number of load/store units stays at 16. Every SP core now can perform one single precision FMA operation (see Section 4.3) per clock cycle. The double precision floating-point performance has been improved: combining the two single precision SP clusters of a SM to one double precision cluster every SM can perform 16 FMA operations per clock cycle, which means that double precision peak performance is half of the single precision processing power. The dual-issue mode (see Section 4.3) has been omitted, but the developers build in two independent *warp schedulers*. Each of them can instruct either one of the two SP clusters, the SFUs or the load/store units. This means that the two SP clusters as well as the SFUs and load/store units can execute different code, but only two of them can get a new instruction in one clock cycle.

There is still an on-chip shared memory on each SM and additionally a *L1 cache* of 16 KB that can be accessed by all of the 32 SP cores. Furthermore, there are additional 32 KB of memory that can be either assigned to the shared

[7] Release of the Fermi chip is planned in the first quarter of 2010 (see also Figure 4.1).

memory or to the L1 cache (decidable by the programmer). The register file has been enlarged to 32K 32-bit registers. To fulfill the requirements of Shader Model 5.0, NVIDIA has added 768 KB of *L2 cache* to the Fermi chip that is shared between all SM. This makes synchronize operations much faster than on previous generations. The new architecture also features a *unified address space*, where the three separate address spaces (thread private local, block shared, and global) now reside in a single, continuous address space. This 40-bit unified address space now supports a terabyte of addressable memory, but the maximum amount of main memory for the Fermi chip will be 6 GB. The Fermi chip is also the first GPU that supports *ECC* (Error Correcting Code) [92–95] memory error correction[8], which will be detachable to increase performance when not needed. Two other new key features of the Fermi chip are the full C++ and IEEE 754-2008 [87, pages 79–103] floating-point support.

Figure 4.6 shows a sketch of the Fermi GPU. To underline that it was designed with GPGPU computing in mind, this sketch lacks all texture processing units. One can see the 16 SMs that are surrounded by the six 64-bit wide memory controllers, the host interface as well as the *GigaThread engine*, which is responsible to distribute the threads to the single SMs. This engine also supports *concurrent kernel execution*, where different kernels of the same application context can be executed simultaneously on the GPU (for more details see [86], GigaThread Thread Scheduler). As noted the SP cores are called CUDA cores and contain separate units for floating-point (FP Unit) and integer (INT Unit) operations (which cannot be used in parallel). There are four execution units (two SP clusters, a load/store unit, and a SFU cluster), which have to share two scheduler/dispatcher units. Fermi introduces a memory hierarchy similar to CPUs: a L1 cache close to the execution units, a shared L2 (uniform) cache, and the main memory.

A comparison between the properties of the Fermi chip and the previous generations (G80 and GT200) are shown in Table 4.1, while more details about the features of the new Fermi chip can be found in [86].

4.5 CUDA Threading Model

The *CUDA threading model* reflects the specific hardware topology of modern NVIDIA GPUs which are well suited for applications with a lot of floating-point operations that can be processed in parallel. NVIDIA calls their new

[8] ECC describes a technique to detect and correct single-bit errors in memory (register files, shared memories, L1 caches, L2 cache, and DRAM memory) before they affect the system.

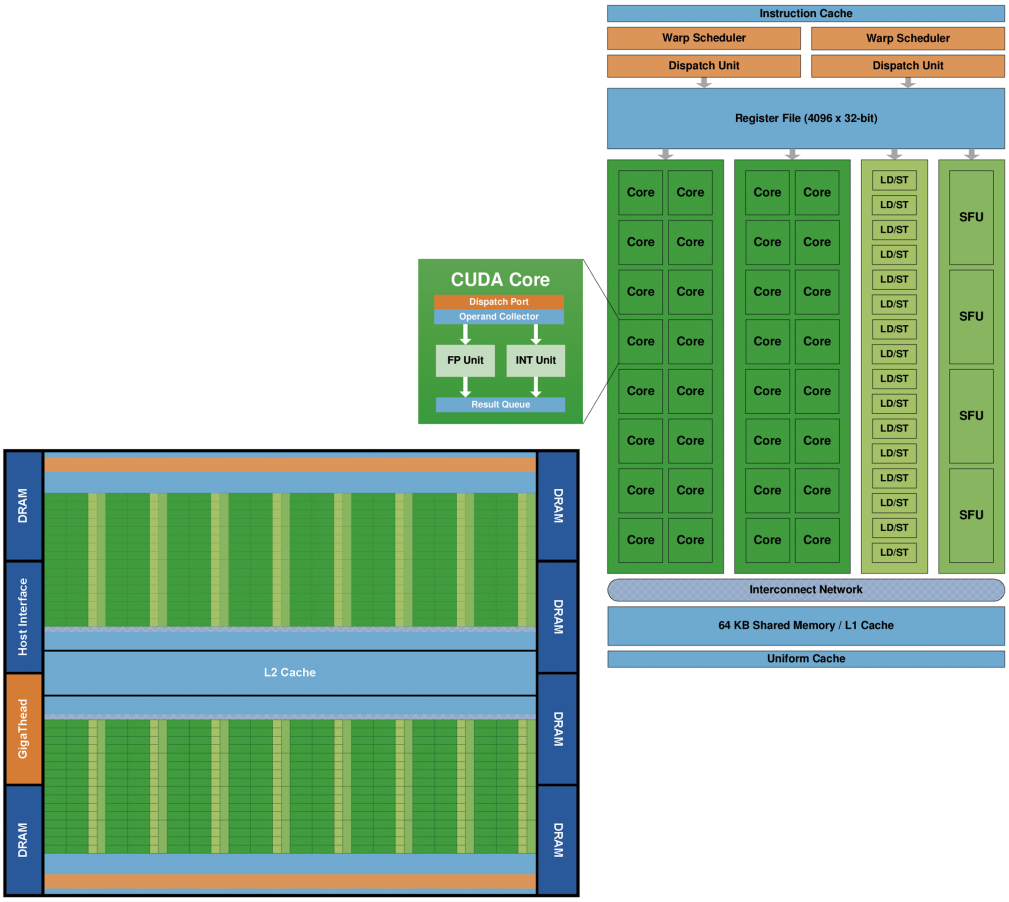


Figure 4.6: Illustration of the future NVIDIA Fermi GPU on the left side and a detailed sketch of a Streaming Multiprocessor (SM) in the top-right corner (from [86], page 7 and 8). The 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

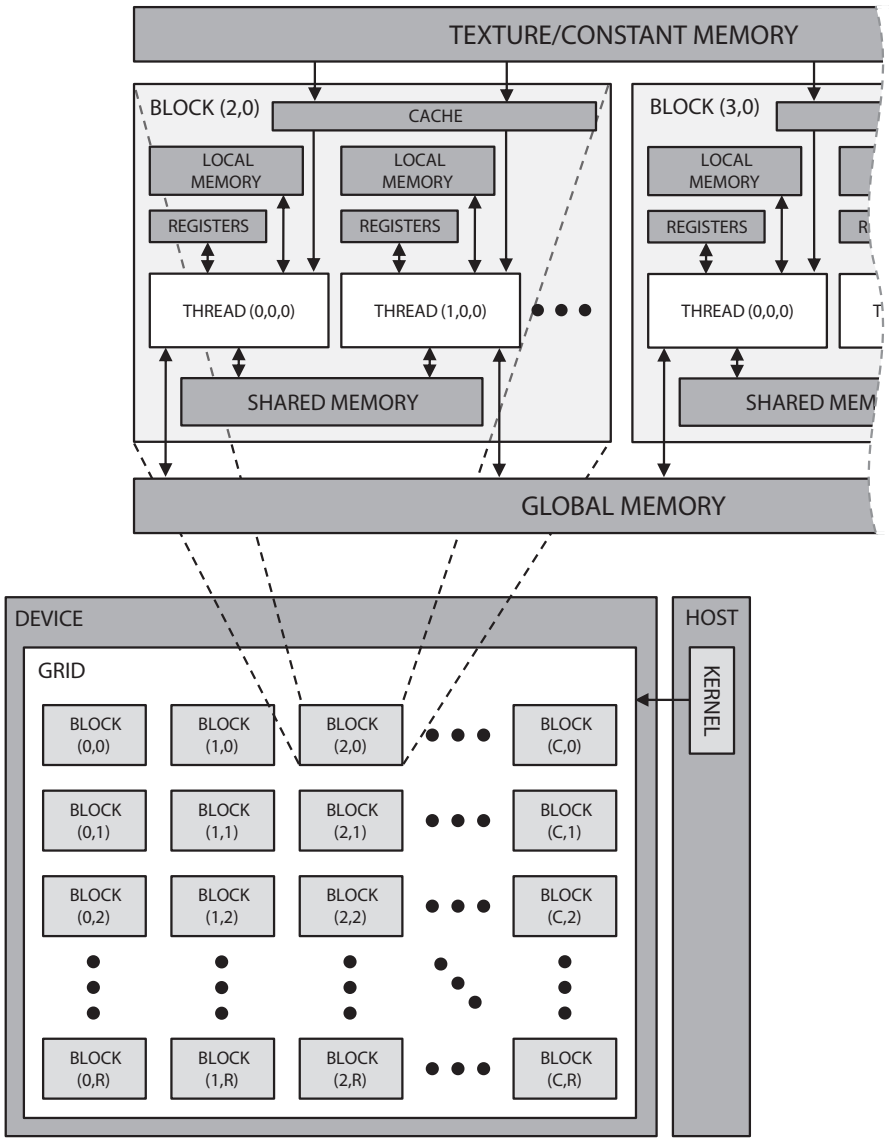


Figure 4.7: CUDA threading and memory topology.

parallel execution model *SIMT* (Single Instruction, Multiple Threads), which is a variant of *SIMD* (Single Instruction, Multiple Data).

At startup of a function on the GPU, a so-called *kernel*, the system creates a certain number of threads defined by the programmer. Those threads are all running the kernel code. The entirety of all those threads is called the *grid* and consists of *thread blocks* that are mapped to it in an one- or two-dimensional manner. The maximum size of the grid is 65,535×65,535 blocks. Every block

consists of a certain number of threads which are arranged in one, two or three dimensions. The size of a thread block is limited to 512×512×64 while the maximum number of threads in a block is limited to 512 (for performance reasons each block should contain at least 32 threads). All threads of a thread block are executed by the same SM. Each thread can be identified by its unique combination of the two global variables blockIdx and threadIdx which define the thread block's position in the grid and the thread's position inside its thread block, respectively.

The smallest execution unit on a NVIDIA CUDA device is one *warp*[9], which consists of 32 threads which execute the same code on different data. The threads of one warp always belong to the same thread block and have consecutive *thread identifiers*. If a thread block consists of less then 32 threads or its threads take different code branches, *stalls* are added to bring the warp to a size of 32. A SM needs more than one serial execution to execute a full warp on current hardware. To fully utilize a CUDA GPU every thread block must contain a multiple of 32 threads and the number of thread blocks must be a multiple of twice the number of SMs on the GPU.

Figure 4.7 shows how threads are grouped and mapped to the hardware in CUDA.

4.6 CUDA APIs

As illustrated in Figure 4.8 there exist two CUDA programming interfaces: The *Driver API* and the *Runtime API*. These two can use the same device code which is usually compiled with NVIDIA's *nvcc compiler* but offer different methods for memory and program flow handling on the GPU. While the Driver API gives a higher level of control, the Runtime API is more comfortable to use. The following sections will introduce the particularities of those APIs and give a short code comparison of a simple vector addition example. A more detailed description of both APIs can be found in [11].

CUDA also provides some libraries for special purposes, such as the CUBLAS library [17] for basic linear algebra routines or the CUFFT library [97] for fast Fourier transformation on the GPU (see also Figure 4.8).

4.6.1 CUDA Runtime API

In newer versions of the CUDA SDK (Software Development Kit) this interface is also called *C for CUDA* because its syntax is based on C with some special extensions. For example it introduces the <<< ··· >>> operator which defines

[9]The term *warp* originates from weaving, the first parallel thread technology [11].

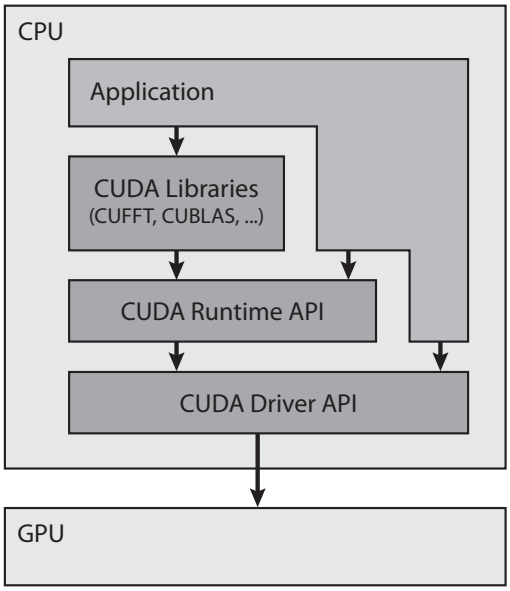


Figure 4.8: Illustration of the CUDA software stack (based on Figure 2-4 in [96]).

the size of the grid and its blocks (see Section 4.5) and can optionally be used to specify the shared memory's size at launch time of a kernel (see Section 4.7). It is placed between a kernel call statement and it's arguments and takes two variables of type dim3 which are separated by a comma. As its name suggests a dim3 variable is a structure containing the three integer members x, y and z (for the first, second, and third dimension). In the <<< ··· >>> operator the first argument indicates the number of blocks in the grid. Since it is restricted to two dimensions the z component must always be 1. The second argument describes the size of each thread block in the grid, where all thread blocks inside one grid are equally sized. The third (optional) argument can be used to define the amount of shared memory which will be allocated at launch time of a kernel. Using this API the arguments can be defined and passed in the same way as in standard C. The Runtime API's host and device code must be compiled with the nvcc compiler. As shown in Figure 4.8 the Runtime API is build on top of the Driver API but it is easier to use. Listing 2 in Appendix D provides an example how to launch a CUDA kernel using the Runtime API.

4.6.2 CUDA Driver API

The host code using the *CUDA Driver API* can be compiled with any C compiler because it is a low-level C interface. It provides functions for all needed functionalities, as for example loading kernels as modules of CUDA binaries or

assembly code, passing arguments to kernels or to define the grid and block size. However, it does not introduce any extensions to the C syntax and handling is therefore long winded. The Driver API's code is usually longer and harder to write but offers a better level of control than in Runtime API implementations. It can handle device code written in any language since it only deals with the binary or assembly code. The code sample in Listing 3 of Appendix D launches the same CUDA kernel as the one in Listing 2 but uses the Driver API instead of the Runtime API.

4.6.3 CUDA Device Code

When using the Runtime API the *device code* (kernel) has to be compiled with the nvcc compiler, when using the Driver API also other compilers and languages, as for example a shading language, can be used. When using the nvcc compiler the device code is equal for both APIs. We will only cover this variant. A CUDA kernel is indicated by the qualifier `__global__` (if it is callable from the host) or `__device__` (if it is callable from the device). A kernel function does not support recursions, static variable declarations inside the body nor a variable number of arguments. Furthermore, `__global__` kernels cannot have a return value and its parameters cannot exceed the size of 256 bytes. The code sample in Listing 4 of Appendix D shows the implementation of a CUDA kernel performing a vector addition as it is launched from the two code samples in Listing 2 (CUDA Runtime API example) and Listing 3 (CUDA Driver API example).

4.7 CUDA Memory Types

In CUDA there exist several different types of memory. Each of them has different properties, is associated with a certain hardware on the device (shown in Figure 4.7) and uses its specific routines for allocation and data transfer between CPU and GPU. The used kind of memory can be controlled by the programmer which can have an enormous impact on the execution speed. Using the most appropriate memory the right way (e.g. *coalesced access* to global memory, avoiding *bank conflicts* in shared memory[10]) is one of the most effective means of improving performance [11, 98, 99].

The different kinds of memory can be divided into two types: The ones that reside on the GPU only, cannot be written or read by the CPU and which's lifetime ends with the termination of the kernel (let's call them Type A) and

[10]Coalesced memory access and the avoidance of bank conflicts in shared memory are achieved by appropriate access patterns, described in [11, 98].

the ones that can be read and written by the CPU and which can be kept alive across different kernel calls (let's call them Type B).

The following enumeration will introduce the different memory types in CUDA along with some of their properties:

- **Registers**. As indicated in the previous sections CUDA devices feature quite a big register file with a size from 16 KB to 64 KB per SM, depending on the model. The register file is used for variables defined inside a kernel without any qualifier. Registers are the fastest memory units on the device and can be accessed with nearly no latency. Variables in registers cannot be shared between different threads. The register file is a Type A memory.

- **Local Memory**. The use of local memory cannot be controlled by the programmer. If a lot of local variables are defined inside a kernel, the compiler can decide to move some of them to the local memory to not exhaust the register file. Unlike as its name suggests this memory is not local from the physical point of view, it resides in the device's main memory. This memory is quite slow and has an access latency of some hundreds of clocks. Just as the register file, local memory is of Type A.

- **Shared Memory**. If variables have to be shared between threads of one thread block the programmer can use the shared memory. To define a variable or array in shared memory the qualifier `__shared__` has to be placed in front of its declaration. The amount of shared memory used by a kernel has to be either defined at compile time or passed as argument during the kernel launch. Its latency is some dozens of clocks and it is of Type A.

- **Texture Memory**. The memory for textures has to be allocated by the CPU either as *linear memory* or as a *CUDA array*[11] [11] and bound to a texture. The data can be read by the GPU using the *texture fetch routines*. Every thread can access the texture which is automatically cached to speed up the reuse of data in texture memory. The latency on a cache hit is some dozens of clocks, on a cache miss it raises to some hundreds clock cycles. Textures mapped on CUDA arrays can take the advantage of the GPU's texture *filtering* and *clamping methods* which are implemented in hardware. The texture indexing can be either *absolute* or *normalized*. Textures in CUDA can be one-, two- or three-dimensional with a maximum side length of 16,384 elements and a maximum size of 512 MB. Textures are a Type B memory.

[11] CUDA arrays are opaque memory layouts optimized for texture fetching.

- **Constant Memory**. As its name suggests the constant memory is read only. It is initialized by the CPU and copied to the main memory of the device. Constant memory is defined with the qualifier __constant__ in front of its declaration. It is cached using the constant cache on each SM which is as fast as shared memory. Constant memory can only be one-dimensional and it is of Type B.

- **Global Memory**. It is either allocated by the CPU in the device's main memory or declared inside a kernel using the __device__ qualifier. However, global memory can be read and modified by the CPU and GPU and it is not cached. This raises the latency to some hundreds clock cycles. To reduce this drawback every SM can perform up to 16 memory operations in parallel under certain circumstances. Global memory is usually used to store results that will be passed to the CPU, therefore it is of Type B.

Further details about the different memory types as well as their optimal access patterns can be found in [11] and [98].

Chapter 5

Implementation

In the context of this thesis we developed a templatized C++ library [19] which's main goal is to enable everyone to build a well performing implementation of arbitrary CNNs without much effort.

This chapter starts with an overview of already existing CNN implementations found in the literature. It moves on with Section 5.2, giving an overview about the implementation we developed, followed by Section 5.3 which comprehends some modifications to the previously described mathematical model of CNNs to keep our implementation simple, fast, and flexible. Section 5.4 describes the performance enhancements of our library gained through parallelization on x86 multicore CPUs and GPUs. The chapter closes with Section 5.5 which covers the software architecture of our library.

5.1 Related Work

During the last years much work has been investigated to improve the performance of neural networks on the GPU. Some of the first GPU implementations of neural networks were presented in [100] and [101]. While the first work only enhanced the performance of the classification part of a neural network using the vertex and pixel shader of a GPU, the latter one also accelerated the training of a neural network.

The first porting of a CNN to the GPU has been released in [102]. It was limited to the network proposed in [1] using the outdated architecture of vertex and pixel shader and implemented through the DirectX API.

In [103] one of the first neural networks for the new unified shader architecture using CUDA was implemented. A GPU implementation of a so-called *Neocognitron* neural network, which is quite similar to a CNN, was presented in [104]. This work only focuses on the improvement of the recognition part on the GPU, training of this network is not considered.

However, as far as we know, no prior effort was made to build a complete framework for accelerating training and classification of arbitrary CNNs on modern GPUs.

5.2 Implementation Procedure

Our work started with a realization of the SimardNet (Subsection 7.1.2) in Matlab[1]. We decided to use Matlab to make our first steps with CNNs because of its easy matrix handling and extended debugging features. The correctness of this implementation was verified using the numerical differentiation method described in Subsection 5.3.3. However, this implementation is not only inflexible and limited to the MNIST database (Appendix A.1) but also very slow. Therefore, we moved on to the development of a library for CNNs in C++. In contrast to the Matlab implementation this library is class-oriented and flexible which means we had to design a quite voluminous class structure as indicated by Section 5.5. This implementation was verified using the procedures described in Subsection 5.3.3 and optimized using Intel's performance libraries (see Subsection 5.4.2).

After completing this tasks the functionalities were ported to CUDA in order to use GPUs for CNNs (see Subsection 5.4.3). The implementation grew to approximately 25,000 lines of code in total. It supports two platforms (x86 CPUs and CUDA-enabled GPUs), CNNs of arbitrary structure, and several types of databases (as described by the DataSource's subclasses in Section 5.5). The modular structure would make it easy to add the support of additional processor types (e.g. OpenCL supporting GPUs or the Cell processor) and databases (e.g. the Weka-format[2]). Finally, this library was used to perform several performance measurements in terms of execution speed (Chapter 6) and classification rate (Chapter 7).

5.3 Used Techniques

The following subsections describe the main concepts that have been applied in our library in order to *simplify* and *accelerate* the implementation. It starts with a method to simplify the backpropagation inside a CNN, followed by a description how to accelerate the convolution for the used hardware. The last part shows two mathematical techniques to verify the implementation of a neural network in order to detect bugs.

[1] `http://www.mathworks.com`

[2] Attribute-Relation File Format (ARFF) used by the Weka machine learning toolkit; `http://www.cs.waikato.ac.nz/~ml/weka/arff.html`

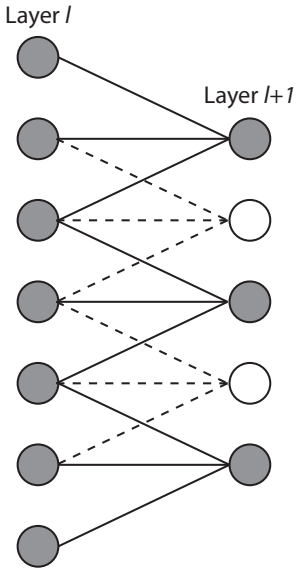


Figure 5.1: Connection structure between two layers of a neural network where layer $l + 1$ performs a one-dimensional convolution using a kernel width of three and a step size of one (based on Figure 4 in [1]).

5.3.1 Making CNNs Simpler

In most implementations of standard MLPs (Section 2.3) each layer relays its activation to the next layer during forward propagation as shown in (2.3) and "pulls" the error back from it during backpropagation as described in (2.10). This has two major drawbacks for CNNs: First, it is very complicated to implement for CNNs because, caused by border effects, the number of outgoing connections is not equal for all neurons in a convolutional layer. Second, every layer has to know the type of its subsequent layer to pull back the errors calculated for every neuron in it (see also Subsection 3.3.1). This is quite cumbersome and difficult to achieve in a flexible library.

Therefore, we used the easier and more flexible approach to "push" the error back to the previous layer as described in [1]. The advantage of this method is that the number of each neuron's incoming connections in a layer, even in a convolutional one, is always constant. This is illustrated in Figure 5.1 which shows a one-dimensional convolution using a kernel width of three and a step size of one. Therefore, it is easier to calculate the sum in equation (2.10) in layer $l + 1$ rather than in layer l. This technique does not require any knowledge about the neighboring layers. In contrast to the many formulas shown in Subsection 3.3.1 each type of layer does only need one method to provide the error to the preceding layer as described in the following parts.

Algorithm 3 Algorithm to compute the $\hat{\delta}$ values inside a convolutional layer.

```
1: Set all δ̂_m^(l)(c, r) to zero.                                   /* Initialize */
2: for n = 0 to M^(l) − 1 do
3:     for each m ∈ V_n^(l) do
4:         for each (x, y) ∈ Q^(l) do
5:             for each (i, j) ∈ R^(l) do
6:                 δ̂_m^(l)(x · h^(l) + i, y · v^(l) + j) += δ_n^(l)(x, y) · w_{m,n}^(l)(i, j)
7:             end for
8:         end for
9:     end for
10: end for
```

Convolutional Layer

Using this technique, a convolutional layer l can calculate its local gradients $\delta_n^{(l)}(c, r)$ as follows

$$\delta_n^{(l)}(c, r) = f'^{(l)}\left(o_n^{(l)}(c, r)\right) \cdot \hat{\delta}_n^{(l+1)}(c, r) \tag{5.1}$$

independent of the subsequent layer $l + 1$, since each layer receives a corresponding $\hat{\delta}^{(l+1)}$ from it. Furthermore, it computes the $\hat{\delta}^{(l)}$ values needed by the preceding layer $l - 1$ according to the Algorithm 3 where

$$Q^{(l)} = \left\{(x, y) \in \mathbb{N}^2 \mid 0 \leq x < W^{(l)}; 0 \leq y < H^{(l)}\right\} \quad \text{and}$$
$$R^{(l)} = \left\{(i, j) \in \mathbb{N}^2 \mid 0 \leq i < k_c^{(l)}; 0 \leq j < k_r^{(l)}\right\}.$$

In this algorithm the variable $M^{(l)}$ defines the number of feature maps in layer l and the set $V_n^{(l)}$ contains the feature maps in the preceding layer $l-1$ that are connected to feature map n in layer l (see also Subsection 3.2.1). The symbols $k_c^{(l)}$ and $k_r^{(l)}$ denote the width and the height of the convolution kernels $\mathbf{w}_{m,n}^{(l)}$ of layer l, while the values $h^{(l)}$ and $v^{(l)}$ describe the horizontal and vertical step size of the convolution in layer l. The remaining two variables $W^{(l)}$ and $H^{(l)}$ define the width and the height of the feature maps in layer l, respectively.

Subsampling Layer

For a subsampling layer l, the local gradients $\delta_n^{(l)}(c, r)$ can be calculated in the same way as for a convolutional one shown in (5.1). Algorithm 4 shows how

Algorithm 4 Algorithm to compute the $\hat{\delta}$ values inside a subsampling layer.

```
1: for n = 0 to M^(l) − 1 do
2:     for each (x, y) ∈ Q^(l) do
3:         for each (i, j) ∈ S^(l) do
4:             δ̂_n^(l)(x · s_c^(l) + i, y · s_r^(l) + j) = δ_n^(l)(x, y) · w_n^(l)
5:         end for
6:     end for
7: end for
```

to compute the $\hat{\delta}^{(l)}$ values for the preceding (convolutional) layer $l-1$ where

$$Q^{(l)} = \left\{(x,y) \in \mathbb{N}^2 \mid 0 \leq x < W^{(l)}; 0 \leq y < H^{(l)}\right\} \quad \text{and}$$
$$S^{(l)} = \left\{(i,j) \in \mathbb{N}^2 \mid 0 \leq i < s_c^{(l)}; 0 \leq j < s_r^{(l)}\right\}.$$

In this algorithm the symbols $s_c^{(l)}$ and $s_r^{(l)}$ denote the width and the height of the subsampling kernel of layer l, respectively, while value $w_n^{(l)}$ is the weight of feature map n in layer l (see also Subsection 3.2.2).

Fully Connected Layer

The local gradients $\delta^{(l)}(j)$ in a fully connected layer l can be obtained by

$$\delta^{(l)}(j) = f'^{(l)}\left(o^{(l)}(j)\right) \cdot \hat{\delta}^{(l+1)}(j), \tag{5.2}$$

whereas the $\hat{\delta}^{(l)}$ values needed by the preceding layer $l-1$ can be computed as shown in Algorithm 5. The symbols $N^{(l)}$ and $N^{(l-1)}$ define the number of neurons in layer l and $l-1$, respectively, while $w^{(l)}(i,j)$ is the weight for the connection from neuron i in layer $l-1$ to neuron j in layer l.

If the preceding layer $l-1$ is a (1D) fully connected layer, it can use the $\hat{\delta}^{(l)}$ values as they are. In the case where layer $l-1$ is a (2D) convolutional or subsampling layer, it has to rearrange the values to match the 2D neuron placement in its feature maps. This can be done reversing the procedure described in Subsection 3.2.3 and illustrated by Figure 3.5.

The resulting regular structure of the backpropagation process can be optimized better. If the weights and $\hat{\delta}$ values are arranged in an adequate order, as for example in our implementation, this operation can be implemented as a matrix-vector multiplication for fully connected layers and as three nested loops for subsampling layers. How to implement this procedure in an efficient way for convolutional layers is described in the following paragraph.

Algorithm 5 Algorithm to compute the $\hat{\delta}$ values inside a fully connected layer.

```
1: Set all δ̂^(l)(i) to zero.                          /* Initialize */
2: for j = 0 to N^(l) − 1 do
3:     for i = 0 to N^(l−1) − 1 do
4:         δ̂^(l)(i) += δ^(l)(j) · w^(l)(i, j)
5:     end for
6: end for
```

5.3.2 Making CNNs Faster

The convolutions inside a CNN are not easy to optimize because of their irregular memory access pattern. The data is not accessed in the same order as it resides in memory and not all values are accessed equally often. To make the access patterns more linear and thus better suited for current processors we used the *unfolding technique* described in [102]. The input is copied to a matrix where the elements of each convolution kernel form one row (in [102]) or one column (in our implementation [19]). Therefore, most of the input elements appear several times in this matrix. Once the unfolding is done the forward- and backpropagation of the convolutional layer can be implemented as a matrix product. This does not only result in an easier but also a much better optimizable implementation.

When implementing the unfolding operation so that every convolution kernel forms one column in the unfolded matrix $\mathbf{U}^{(l)}$, its height is equal to the area of the convolution kernels $k_c^{(l)} \cdot k_r^{(l)}$ in layer l times the number of feature maps $M^{(l-1)}$ in layer $l-1$. The width is defined by the feature map area $W^{(l)} \cdot H^{(l)}$ in layer l which is equal to the number of kernel applications on every feature map in layer $l-1$. The following formula describes the computation of the element on position (c, r) in matrix $\mathbf{U}^{(l)}$, unfolding the output $\mathbf{y}^{(l-1)}$ of the feature maps in layer $l-1$:

$$U^{(l)}(c,r) = y_n^{(l-1)}(x \cdot h^{(l)} + i, y \cdot v^{(l)} + j) \tag{5.3}$$

with

$$\begin{aligned}
n &= \left\lfloor r / \left(k_c^{(l)} \cdot k_r^{(l)}\right) \right\rfloor, \\
j &= \left\lfloor \left(r - n \cdot k_c^{(l)} \cdot k_r^{(l)}\right) / k_c^{(l)} \right\rfloor, \\
i &= r - k_c^{(l)} \cdot \left(j + n \cdot k_r^{(l)}\right), \\
y &= \left\lfloor c / W^{(l)} \right\rfloor, \quad \text{and} \\
x &= c - y \cdot W^{(l)}
\end{aligned}$$

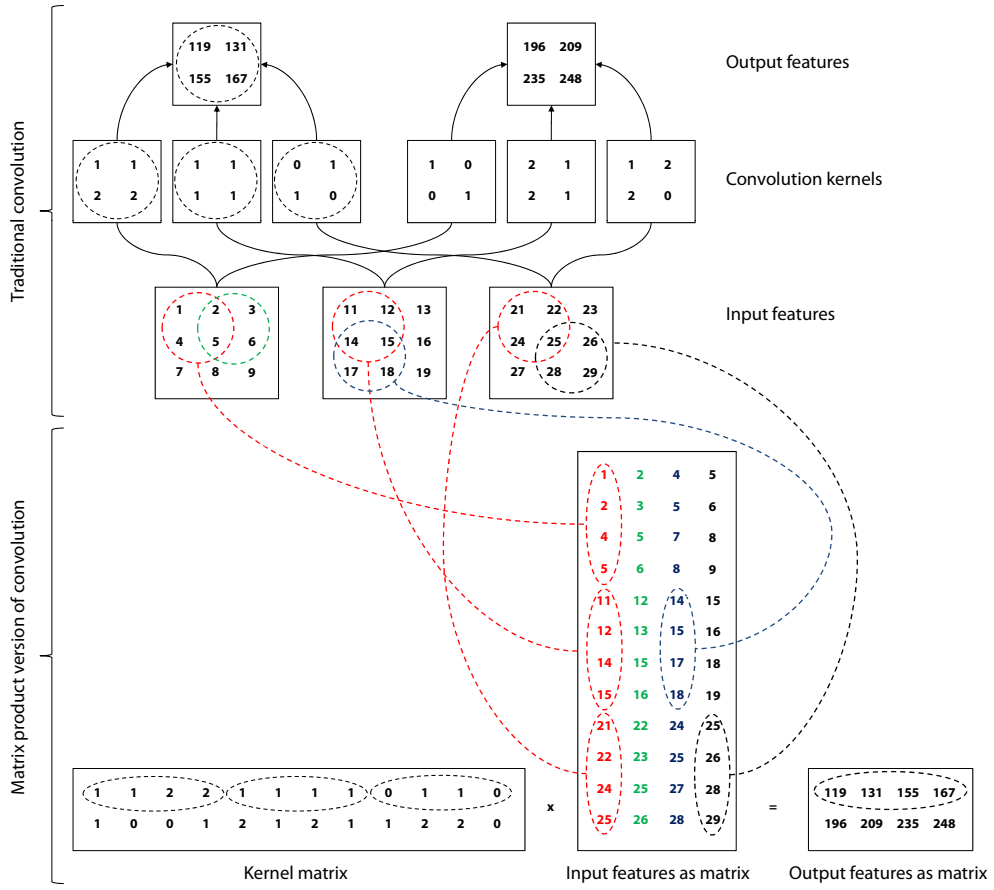


Figure 5.2: Illustration of the convolution operation inside a convolutional layer of a CNN (based on Figure 2 in [102]). The top figure presents the traditional approach, while the bottom figure presents the matrix variant using the unfolding technique.

where $k_c^{(l)}$ and $k_r^{(l)}$ denote the width and the height of the convolution kernels of layer l, while $W^{(l)}$ is the width of the feature maps in layer l. The values $h^{(l)}$ and $v^{(l)}$ describe the horizontal and vertical step size of the convolutions in layer l. In this equation x and y denote the column and row in the feature map n of layer $l-1$, while i and j define the horizontal and vertical position inside the convolution kernel. An illustration of the unfolding process is shown in Figure 5.2 while a pseudocode implementation can be found in Algorithm 6.

Unfolding the input in this way every column of the unfolded matrix $\mathbf{U}^{(l)}$ holds all inputs that are needed to calculate the output of one neuron in layer l. Thus, the convolution can then be implemented using a simple matrix multiplication as shown in Figure 5.2. The forward propagation inside a convolutional layer

Algorithm 6 Unfolding procedure to keep convolution more efficient.

Input:

- The output $\mathbf{y}^{(l-1)}$ of the preceding layer $l-1$.

Output:

- The matrix $\mathbf{U}^{(l)}$ according to the structure illustrated in Figure 5.2.

```
1: /* Loop over number of feature maps in layer l − 1 */
2: for n = 0 to M^(l−1) − 1 do
3:    /* Loop over feature map width of layer l */
4:    for x = 0 to W^(l) − 1 do
5:       /* Loop over feature map height of layer l */
6:       for y = 0 to H^(l) − 1 do
7:          /* Loop over kernel width of layer l */
8:          for i = 0 to k_c^(l) − 1 do
9:             /* Loop over kernel height of layer l */
10:            for j = 0 to k_r^(l) − 1 do
11:               /* Copy values from the output y^(l−1) of the preceding layer
                  l − 1 to the appropriate position in the unfolded matrix U^(l) */
12:               U^(l)(y · W^(l) + x, n · k_c^(l) · k_r^(l) + j · k_c^(l) + i)
                     = y_n^(l−1)(x · h^(l) + i, y · v^(l) + j)
13:            end for
14:         end for
15:      end for
16:   end for
17: end for
```

can then be written as follows:

$$\mathbf{Y}^{(l)} = f^{(l)}\left(\mathbf{K}^{(l)} \times \mathbf{U}^{(l)} + \mathbf{B}^{(l)}\right) \tag{5.4}$$

where $Y^{(l)}(i,j)$ holds the ith neuron of layer l's feature map j (feature maps are aggregated row-wise to get one vector for each feature map). The matrix $\mathbf{K}^{(l)}$ contains all weights of layer l where each row j is composed of the single convolution kernels $\mathbf{w}_{m,j}^{(l)}$ (aggregated row-wise) of feature map j in layer l as illustrated in Figure 5.2. If the feature maps of the previous layer $l-1$ and the ones of the convolutional layer l are only partially connected the corresponding entries in the matrix $\mathbf{K}^{(l)}$ for a nonexisting connection are set to zero. The matrix $\mathbf{B}$ has the same size as the matrix product in (5.4) where each column contains all feature map's biases in layer l in order to add the appropriate bias to every output; the vector containing the single bias values of each feature map

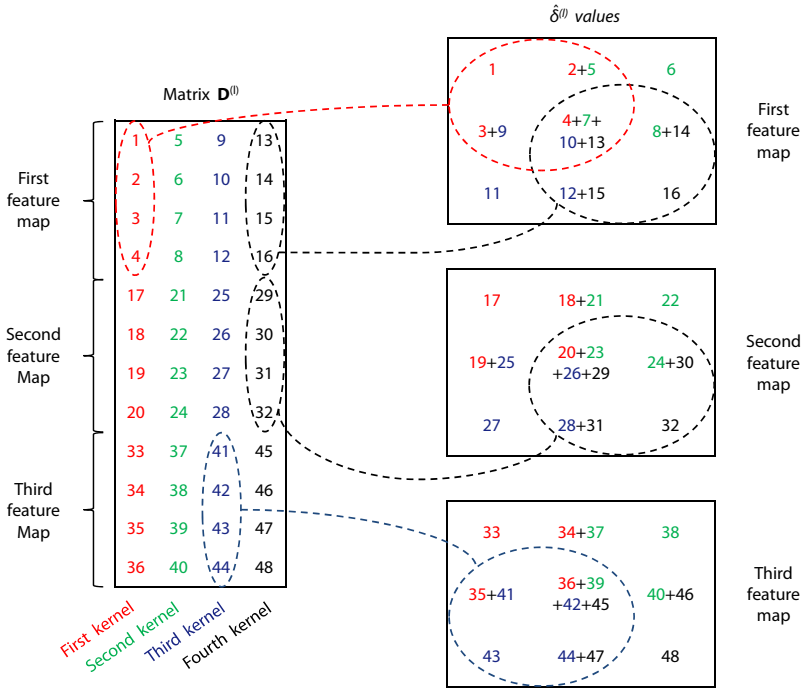


Figure 5.3: Graphical representation of the backfolding procedure to obtain the single $\hat{\delta}^{(l)}$ values needed by the preceding layer in order to propagate the error backwards through the network.

in layer l is therefore replicated to each column in the matrix.

A nice byproduct of this unfolding of the input is that the error gradient computation and the backpropagation step can be written as another matrix product. To compute the gradients for each weight in the convolutional layer one can construct a matrix $\mathbf{\Delta}^{(l)}$ of the same structure as the matrix $\mathbf{Y}^{(l)}$ but holding the local gradients $\delta_n^{(l)}(c, r)$ for each neuron in the layer. This is not affected by the unfolding process. To calculate the matrix $\mathbf{G}^{(l)}$ containing the gradients for each weight only the following matrix multiplication has to be performed:

$$\mathbf{G}^{(l)} = \mathbf{\Delta}^{(l)} \times \left[\mathbf{U}^{(l)}\right]^T \tag{5.5}$$

where the operator $[\mathbf{A}]^T$ delivers the transposed matrix of $\mathbf{A}$. Since the obtained matrix $\mathbf{G}^{(l)}$ has the same structure as the matrix $\mathbf{K}^{(l)}$ the weight update of the backpropagation training algorithm (Subsection 2.5.1) can be simply performed in a pointwise way of the corresponding elements. The error gradient for each

bias of feature map j in layer l can be calculated by summing up the elements of row j in the matrix $\mathbf{\Delta}^{(l)}$. That's all that has to be done in order to calculate the needed gradients for the convolutional layer l.

However, if we want to propagate the error further back through the network to layer $l-1$ as described in Subsection 5.3.1 we have to calculate the $\hat{\delta}^{(l)}$ values needed by the preceding layer. This can be done by multiplying the transposed matrix $\mathbf{K}^{(l)}$ with the matrix $\mathbf{\Delta}^{(l)}$:

$$\mathbf{D}^{(l)} = \left[\mathbf{K}^{(l)}\right]^T \times \mathbf{\Delta}^{(l)} \tag{5.6}$$

where the resulting matrix $\mathbf{D}^{(l)}$ contains the single $\hat{\delta}^{(l)}$ values for every connection from layer $l-1$ to layer l according to the structure shown in Figure 5.3. Since we need such a value for every neuron in layer $l-1$ the values in matrix $\mathbf{D}^{(l)}$ that correspond to a connection starting at the same neuron are accumulated together and arranged in the same order as the neurons in layer $l-1$. Therefore, the elements in matrix $\mathbf{D}^{(l)}$ have to be *backfolded* as illustrated in Figure 5.3 according to the following formula:

$$\hat{\delta}_n^{(l)}(c,r) = \sum_{(x,y)\in Q} D\left(y \cdot W^{(l)} + x, n \cdot k_c^{(l)} \cdot k_r^{(l)} + \left(r - y \cdot v^{(l)}\right) \cdot k_c^{(l)} + \left(c - x \cdot h^{(l)}\right)\right) \tag{5.7}$$

with

$$\begin{aligned}
Q &= \left\{(x,y) \in \mathbb{N}^2 \mid x_1 \le x < x_2; y_1 \le y < y_2\right\}, \\
x_1 &= \left\lfloor \max\left(0, c - \left(k_c^{(l)} - h^{(l)}\right)\right) / h^{(l)} \right\rfloor, \\
x_2 &= \min\left(\left\lfloor c/h^{(l)} \right\rfloor + 1, W^{(l)}\right), \\
y_1 &= \left\lfloor \max\left(0, r - \left(k_r^{(l)} - v^{(l)}\right)\right) / v^{(l)} \right\rfloor, \quad \text{and} \\
y_2 &= \min\left(\left\lfloor r/v^{(l)} \right\rfloor + 1, H^{(l)}\right)
\end{aligned}$$

where $k_c^{(l)}$ and $k_r^{(l)}$ denote the width and the height of the convolution kernels of layer l, while $W^{(l)}$ and $H^{(l)}$ is the width and the height of the feature maps in layer l. The values $h^{(l)}$ and $v^{(l)}$ describe the horizontal and vertical step size of the convolutions in layer l. Algorithm 7 illustrates an pseudocode implementation of this backfolding procedure. The resulting $\hat{\delta}_n^{(l)}$ values can then be used to compute the local gradients $\delta_n^{(l)}(c,r)$ in the preceding layer $l-1$ in order to propagate the error backwards through the network.

Algorithm 7 Backfolding procedure to keep convolution more efficient.

Input:

- The matrix $\mathbf{D}^{(l)}$ according to the structure illustrated in Figure 5.3.

Output:

- The $\hat{\delta}^{(l)}$ values needed by the preceding layer $l-1$.

```
/* Initialize */
Set all δ̂_n^(l)(c, r) to zero.
/* Loop over number of feature maps in layer l − 1 */
for n = 0 to M^(l−1) − 1 do
    /* Loop over feature map width of layer l */
    for x = 0 to W^(l) − 1 do
        /* Loop over feature map height of layer l */
        for y = 0 to H^(l) − 1 do
            /* Loop over kernel width of layer l */
            for i = 0 to k_c^(l) − 1 do
                /* Loop over kernel height of layer l */
                for j = 0 to k_r^(l) − 1 do
                    /* Sum up the corresponding elements in matrix D^(l)
                    to obtain the δ̂^(l) values for the preceding layer l − 1 */
                    δ̂_n^(l)(x · h^(l) + i, y · v^(l) + j)
                        += D^(l)(y · W^(l) + x, n · k_c^(l) · k_r^(l) + j · k_c^(l) + i)
                end for
            end for
        end for
    end for
end for
```

5.3.3 Debugging CNNs

Since incorrect implementations of a neural network sometimes yield reasonable results we proved the correctness of our implementation with two methods. We verified the correctness of the gradients computed by the backpropagation function comparing them with gradients computed via numerical differentiation. Additionally, we calculated the Jacobian matrix using backpropagation and an arbitrarily accurate estimate of it by adding small variations to the input and calling the forward propagation function. These two matrices are then compared in order to verify the implementation. The following paragraphs describe these two methods in detail.

Numerical Differentiation

The gradients computed by the error backpropagation algorithm described in Subsection 2.5.1 can also be calculated using *numerical differentiation* as shown in [22, pages 147–148]:

$$\frac{\partial E}{\partial w^{(l)}(i,j)} = \frac{E_{\left[w^{(l)}(i,j)+\epsilon\right]} - E_{\left[w^{(l)}(i,j)-\epsilon\right]}}{2 \cdot \epsilon} \tag{5.8}$$

The symbol E denotes the network's error calculated by an error function (Section 2.6) and $E_{\left[w^{(l)}(i,j)\pm\epsilon\right]}$ is the error when replacing $w^{(l)}(i,j)$ with $w^{(l)}(i,j) \pm \epsilon$. The variable ϵ should be a (not to) small value, we achieved good results using 10^{-2} for single precision tests and 10^{-5} when running the tests in double precision.

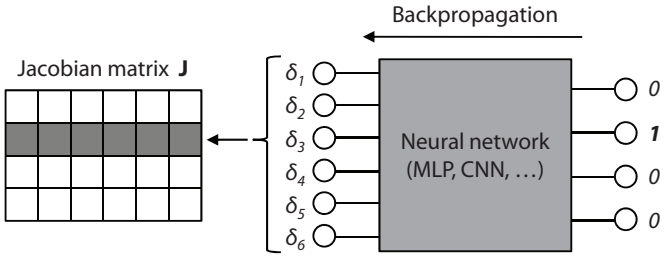
Calculating these approximations of the gradient for every weight (and also bias) of the whole network requires two forward propagations for each trainable parameter in the network. For all those forward propagations one has to use the same input pattern and training signal, which both can be generated randomly in this case. The gathered approximations of the gradients can then be used to verify the correctness of all gradients calculated by backpropagation comparing the corresponding values. In a correct implementation they should be almost equal, in our implementation we tolerated an absolute difference of 10^{-3} for the single precision implementation and 10^{-8} for the double precision one.

Jacobian Matrix

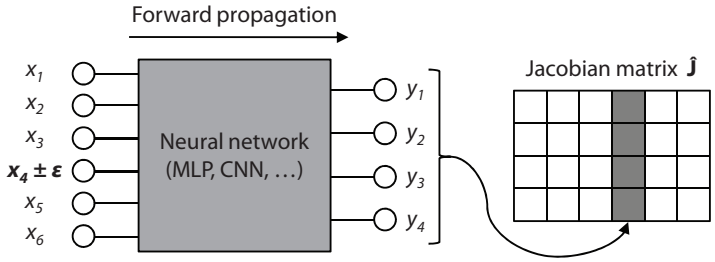
The error backpropagation can also be used to calculate the so-called *Jacobian matrix* $\mathbf{J}$ which's entries are defined by the derivatives of the networks output $\mathbf{y}^{(L)}$ with respect to its input $\mathbf{x}$ as described in [22, pages 148–150]:

$$J(i,j) = \frac{\partial y^{(L)}(j)}{\partial x(i)} \tag{5.9}$$

The height of the Jacobian matrix is equal to the number of output neurons $N^{(L)}$, while its width is defined by the number of inputs $N^{(0)}$ to the network. To calculate this matrix one can generate the input pattern and the trainable parameters randomly. The elements of the matrix $\mathbf{J}$ can be calculated by simply back-propagating the error to the network's input neurons of the 0th layer (input layer), where the local gradient $\delta^{(l)}(i)$ of each neuron i is written in the corresponding column. To obtain the elements of line j in the Jacobian matrix the value of $e(j)$ in the backpropagated error vector is set to 1 and all other elements in it to 0. In each backpropagation the obtained local gradients for the



(a) Constructing the Jacobian matrix $\mathbf{J}$ using the backpopagation function.



(b) Constructing an approximation $\hat{\mathbf{J}}$ of the Jacobian matrix using the forward propagation function.

Figure 5.4: Illustration how to build the Jacobian matrix $\mathbf{J}$ using backpropagation (a) and an arbitrarily accurate estimate $\hat{\mathbf{J}}$ of it by adding small variations to the input and calling the forward propagation function (b).

network input neurons are written to one row in the Jacobian matrix at once as shown in Figure 5.4(a). Therefore, the procedure requires one backpropagation for each neuron in the output layer.

An *numerical approximation* $\hat{\mathbf{J}}$ of the Jacobian matrix can be calculated using the forward propagation function as illustrated in Figure 5.4(b). This procedure is similar to the one described in the previous section, but instead adding small perturbations to the weights in this case some jitter is added to the input. Apart from this jitter the same input pattern and trainable parameters are used for the entire matrix and which should be the same as the one used to calculate the Jacobian matrix by backpropagation in order to compare them. The elements of the matrix $\hat{\mathbf{J}}$ are calculated as follows:

$$\hat{J}(i,j) = \frac{y^{(L)}_{[x(i)+\epsilon]}(j) - y^{(L)}_{[x(i)-\epsilon]}(j)}{2 \cdot \epsilon} \tag{5.10}$$

where $y^{(L)}_{[x(i)\pm\epsilon]}$ is the network's output when replacing input $x(i)$ by $x(i) \pm \epsilon$. In our implementation we used an ϵ of 10^{-1} when running the networks in

single precision and 10^{-4} when running it in double precision. The two matrices $\mathbf{J}$ and $\hat{\mathbf{J}}$ are then compared in order to verify the implementation. To verify the correctness of our implementation we tolerated a maximum difference of 10^{-5} and 10^{-11} for the elements in the matrix for single and double precision, respectively. In contrast to the method using the backpropagation function, two forward propagations are needed for each input neuron of the network in order to construct the whole matrix.

5.4 Accelerate the Implementation

To accelerate an existing implementation there exist two major approaches: one can adapt the code to utilize the used processor in a more efficient way or even exchange the processor with one that is more suited for this kind of problem and therefore faster. Some problems and drawbacks when using a massively parallel (and therefore well suited for neural networks) processors for neural networks are discussed in the following subsection.

Our implementation began with a *trivial implementation* called $\text{CPU}_{\text{triv.}}$ which was written in C++ for the usage on x86 CPUs. Starting with this version we exploited both of the previously mentioned approaches. The $\text{CPU}_{\text{triv.}}$ variant already implements the optimizations described earlier in the Subsections 5.3.1 and 5.3.2. However, it does not deal with any kind of parallelism nor with cache utilization.

To build our *optimized version* called $\text{CPU}_{\text{opt.}}$ we used performance libraries to unveil the whole potential of today's multicore CPUs. Furthermore, we ported our implementation to CUDA in order to use today's GPUs which seem to be more convenient for neural networks than CPUs (as demonstrated in Chapter 6).

5.4.1 Difficulties Using Modern Processors

As mentioned in Chapter 4 today's processors usually gain speed through their *parallelism*. In real-world applications it is often difficult to find parallelizable sections in order to keep all the hardware busy. Neural networks usually are considered as very parallel programs, but there are some limitations. When training a neural network using the online variant of the backpropagation algorithm (as described for MLPs in Section 2.5 and for CNNs in Section 3.3) single training iterations of an epoch cannot be executed concurrently. Online backpropagation performs better than the batched variant on a lot of classification problems [34, pages 13–15], especially if the training dataset is rather large as for example in the MNIST database (Appendix A.1). Therefore, in general every training iteration has to be executed in serial.

The training iteration itself needs some synchronization too, because the forward and backward propagation pass have to be synchronized after each layer to ensure that all needed data is available. This means that the degree of parallelism in a neural network depends on the size of the single layers and is not equal throughout the whole program. When executing a (forward or backward) propagation pass of a rather small layer, like for example the output layer which usually consists of only a few neurons (sometimes of only one neuron), the parallelism is rather poor. This is not such a big issue when executing a neural network on CPU since today's quadcore processors, like the one used for the benchmarks in this thesis (see Table 6.1 on page 86), can execute only up to 16 calculations in parallel. The problem is a bit more serious when using a GPU. The GPU we used in this work can execute 240 MAD operations and 240 multiplications in parallel. Taking into account the CUDA threading model (which is described in detail in Section 4.5) we get an even higher number of parallel calculations that is needed to keep the GPU at full load. It needs 960 MAD operations and the same number of multiplications to keep all the hardware busy. This very high number of parallel executions is not always given in small layers of a neural network.

Another big issue in high performance computing is the availability of data, especially when using massively parallel processors with an enormous peak performance. The CPU can rely on a very mature *cache structure* to hide data latency while in CUDA caches are either read-only or have to be controlled by the programmer. However, the modular and flexible structure of our library does limit the use of caches on the GPU to a small subset of the implemented functions (as for example the matrix-matrix multiplication). Each operation has to read it's data from the device memory and write it back to it at termination. An optimized implementation of one single neural network, where the sizes of all layers and their order are fixed and known at programming time, could make a better use of the GPU's caches, especially if this network is relative small. This would result in an implementation with an even higher performance.

5.4.2 Optimization Using Performance Libraries

When using the techniques described in the previous section most of the operations inside a CNN can be implemented using matrix-matrix and matrix-vector operations. For such operations exist numerous highly optimized implementations for CPUs. Using such a library keeps the implementation effort small and leads to a fast implementation.

To optimize our CPU implementation we decided to use the *performance libraries* from Intel, namely the Intel Performance Primitives (IPP) [15] and the

Math Kernel Library (MKL) [16]. There were many reasons for this decision: The combination of these two libraries cover most of the needed functionalities. Furthermore, the implementation from Intel seems to deliver the highest performance for the used CPU, using the advantage of both, the *multicore architecture* distributing the workload to several threads and the *SIMD capability* of modern CPUs by implementing the newest SSE instructions. Another big advantage of those libraries is that they are provided by Intel, the world's largest computer chip manufacturer. Thus, this libraries will be kept up-to-date and scale with future processor generations.

5.4.3 Optimization Using GPUs

As already mentioned in the introduction in Chapter 1 neural networks are well suited for *GPGPU computing* because of their huge amount of floating-point operations and low data transfer on both, the input and the output side of the network. Therefore, we decided to port our library to GPUs in order to achieve the shortest possible training and classification time.

To realize this we used the proprietary programming language CUDA because it is the most popular and also the most advanced way to program GPUs at the moment. A short introduction into this programming language is given in Chapter 4. Since most operations inside our library are matrix-matrix and matrix-vector operations some of them have been implemented using the CUBLAS library [17] provided by NVIDIA. This bears the advantage that these functions will be kept up to date and also support future graphics chips in a efficient way. However, the quality of the CUBLAS library is not comparable to the two Intel libraries IPP and MKL, not in terms of delivered functionalities, nor in terms of performance (in relation to a trivial implementation for this processor type). Therefore, some functions had to be implemented manually by own kernels because the CUBLAS implementation delivered poor performance or because it did not deliver them at all. To implement the needed functionalities we used the CUDA Runtime API (Subsection 4.6.1). Since this API uses standard C pointers, the interface of the CUDA functions in our library is the same as the one of their CPU counterparts. However, they are not directly interchangeable since one expects a pointer to CPU memory, the other to GPU memory.

Thus, when implementing a CNN using our library, there are two differences between a CPU and a GPU network: First, in the GPU version the input has to be copied to the GPU's memory and the result has to be read from it. Second, the names of the GPU versions of all classes start with the letters `Cu` which stands for CUDA. Apart from that, the implementation is absolutely the same,

which means that a porting of an existing network implementation from CPU to GPU is rather simple using our library.

Although CUDA is quite mature, it seems to be not totally bug free at this point of time. One bug which we discovered during the implementation of the GPU variant of our library is described in detail in Appendix C.

5.5 Software Architecture

Our implementation has been designed as a templatized C++ library, where each class accepts at least one template argument `typename T` to specify the used floating-point precision (32- or 64-bit floating-point arithmetic). The following enumeration describes in short the core classes of the library which are also illustrated in Figure 5.5. For a more detailed description we refer to the source code and the documentation available online at [19].

- **Trainer**: Defines the interface for any supervised trainer (Section 2.5). Each trainer takes a neural network and a datasource as input, where the datasource is used to perform the implemented training algorithm on the given network.
  - ◇ **OnlineTrainer**: Implementation of the online backpropagation algorithm as described in Subsection 2.5.1.
  - ◇ **BatchTrainer**: Implementation of the batch backpropagation algorithm as described in Subsection 2.5.1.
  - ◇ **MomentumTrainer**/**CuMomentumTrainer**: CPU/GPU implementation of the gradient descent algorithm with momentum term as described in Subsection 2.5.2.
  - ◇ **RpropTrainer**/**CuRpropTrainer**: CPU/GPU implementation of the resilient backpropagation (RPROP) algorithm as described in Subsection 2.5.2.
- **DataSource**: Defines the interface for any datasource used as training, validation or test set during the training of a neural network. Each datasource has to deliver a requested pattern and the corresponding label, regardless of where and how the data are stored.
  - ◇ **MnistDataSource**: Designed to work with the MNIST handwritten digit database (Appendix A.1).
    - ▷ **MnistMemSource**: Loads the whole database file immediately to the memory (needs more memory).

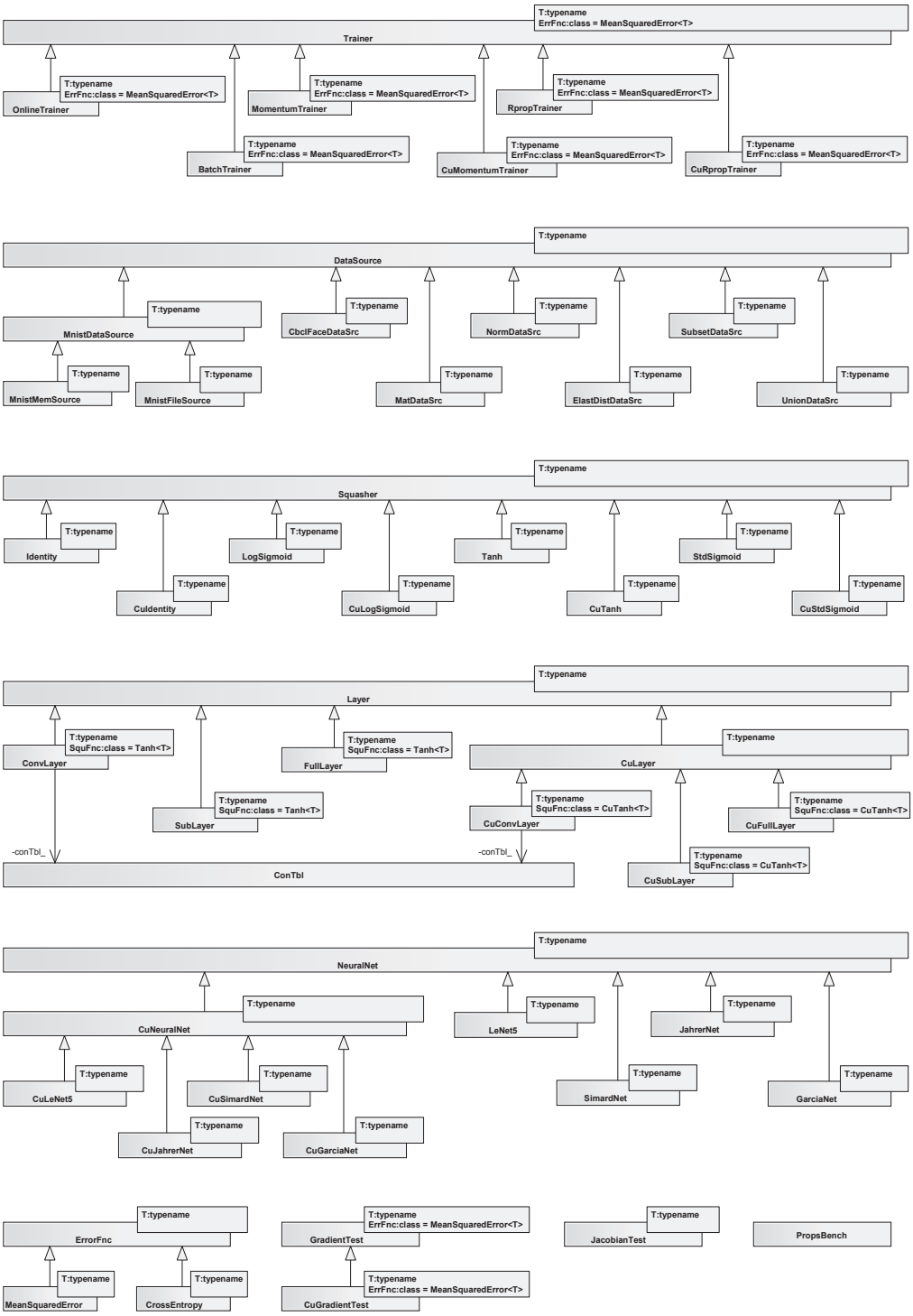


Figure 5.5: Overview of the fundamental classes in our library [19].

        - ▷ **MnistFileSource**: Loads any requested pattern and label dynamically from the database file (needs less memory).
    - ◇ **CbclFaceDataSrc**: Designed to work with the MIT CBCL face database (Appendix A.2).
    - ◇ **MatDataSrc**: Designed to work with datasets stored in a Matlab array inside a MAT-File[3], e.g. the face database from Son Lam Phung (Appendix A.3).
    - ◇ **NormDataSrc**: Takes another datasource as input and normalizes its elements as described in Subsection 2.5.4.
    - ◇ **ElastDistDataSrc**: Applies the elastic distortion technique described in Appendix B to enlarge a given datasource.
    - ◇ **SubsetDataSrc**: Used to pick a subset of another datasource's elements.
    - ◇ **UnionDataSrc**: Used to combine two datasources to a new one.
- **Squasher**: Defines the interface for any squasher/activation function (Section 2.7). Each class has to implement a method which delivers the value of the activation function and a method which delivers the value of the derivative of the activation function.
    - ◇ **Identity**/**CuIdentity**: CPU/GPU implementation of the identity function (Subsection 2.7.1).
    - ◇ **LogSigmoid**/**CuLogSigmoid**: CPU/GPU implementation of the logsigmoid function (Subsection 2.7.2).
    - ◇ **Tanh**/**CuTanh**: CPU/GPU implementation of the tangens hyperbolicus function (Subsection 2.7.3).
    - ◇ **StdSigmoid**/**CuStdSigmoid**: CPU/GPU implementation of the stdsigmoid function (Subsection 2.7.4).
- **Layer**: Defines the interface for any layer in a CNN (Section 3.2). Each layer has to implement the forward propagation as described in Section 3.2, the backward propagation as described in Subsection 5.3.1, and the weight update defined in Subsection 2.5.1.
    - ◇ **ConvLayer**: Implementation of the convolutional layer as described in Subsection 3.2.1.

[3]See also: `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f39876.html`

- ◇ **SubLayer**: Implementation of the subsampling layer as described in Subsection 3.2.2.
- ◇ **FullLayer**: Implementation of the fully connected layer, used for MLPs and CNNs, as described in Section 2.3 and Subsection 3.2.3, respectively.
- ◇ **CuLayer**: Base class for all layers implemented in CUDA to run on the GPU.
  - ▷ **CuConvLayer**: CUDA variant of the ConvLayer.
  - ▷ **CuSubLayer**: CUDA variant of the SubLayer.
  - ▷ **CuFullLayer**: CUDA variant of the FullLayer.

- **NeuralNet**: Defines the interface for any neural network in our library. Each neural network can easily be constructed by composing the single types of layers (convolutional, subsampling, and fully connected layer), described above.
  - ◇ **LeNet5**: Implementation of the LeNet5 network as described in Subsection 7.1.1.
  - ◇ **SimardNet**: Implementation of the SimardNet network as described in Subsection 7.1.2.
  - ◇ **JahrerNet**: Implementation of the JahrerNet network as described in Subsection 7.1.4.
  - ◇ **GarciaNet**: Implementation of the GarciaNet network as described in Subsection 7.2.1.
  - ◇ **CuNeuralNet**: Base class for all neural networks implemented in CUDA to run on the GPU.
    - ▷ **CuLeNet5**: CUDA variant of the LeNet5.
    - ▷ **CuSimardNet**: CUDA variant of the SimardNet.
    - ▷ **CuJahrerNet**: CUDA variant of the JahrerNet.
    - ▷ **CuGarciaNet**: CUDA variant of the GarciaNet.

- **ErrorFnc**: Defines the interface for any error function (Section 2.6). Each error function has to return a global error which considers all output neurons and an error vector containing a separate error for each output neuron which is backpropagated through the network.
  - ◇ **MeanSquaredError**: Implementation of the mean-squared-error (MSE) as described in Subsection 2.6.1.

- ◇ **CrossEntropy**: Implementation of the cross-entropy (CE) as described in Subsection 2.6.2.

- **ConTbl**: Used to define the connectivity of a convolutional layer's feature maps to the ones in its preceding layer in form of a connection matrix, e.g. in a LeNet5.

- **GradientTest**: Used to verify the correctness of a neural network's implementation with the method using numerical calculated gradients as described in Subsection 5.3.3.
  - ◇ **CuGradientTest**: CUDA variant of the GradientTest.

- **JacobianTest**: Used to verify the correctness of a neural network's implementation with the method using the Jacobian matrix as described in Subsection 5.3.3.

- **PropsBench**: Performs a number of forward and backward propagations on a given network using a randomly generated pattern and measures the elapsed time.

5.5.1 Building a CNN

As illustrated in the previous subsection our library already contains implementations of some known CNNs. In the following we will give a short overview how to build such a network using our library [19]. To construct a new network one has to create a class derived from NeuralNet and implement the specified interface. The network itself can then easily be constructed by composing the single types of layers as shown in Figure 5.6 for the LeNet5 (Subsection 7.1.1). The LeNet5 is composed of six layers, using all three types introduced in Chapter 3. The members layer1_, layer3_ and layer5_ represent the convolutional layers C1, C3, and C5, while layer2_ and layer4_ typify the subsampling layers S2 and S4. The member layer6_ stands for the fully connected output layer F6.

The following enumeration describes in short the interface which has to be implemented for any network:

- **forget**: Initializes all weights and biases of the network randomly.

- **reset**: Resets previously calculated gradients to zero.

- **update**: Performs the weight update of the error backpropagation as described in Subsection 2.5.1.

- **fprop**: Performs the forward propagation as described in Section 3.2.

- **bprop**: Performs the backward propagation as described in Section 3.3.
- **load**: Loads previously stored trainable parameters (weights and biases) from the harddisk.
- **save**: Saves the current values of the trainable parameters (weights and biases) to the harddisk.
- **writeOut**: Writes the current excitation of all neurons in the single layers to a file.
- **sizeIn**: Returns the number of inputs to the network.
- **sizeOut**: Returns the number of neurons in the output layer.
- **trainableParam**: Returns a reference to all weights and biases inside the network.
- **toString**: Returns a string describing the network's architecture.
- **numTrainableParam**: Returns the total number of trainable parameters (weights and biases) of the network.
- **numConnections**: Returns the number of connections between the neurons in the network.

All of these functions call the corresponding functions of the single layers inside the network either in ascending or in decreasing order (dependent on the function) while some of them have to implement additional functionalities. For example, the fprop method of a network takes a single pattern as input, passes it through the layers, and uses the output of the last layer as final result of the forward propagation. For more information on how to build a CNN using our library we refer to the sample networks' implementations, e.g. LeNet5, SimardNet, JahrerNet or GarciaNet.

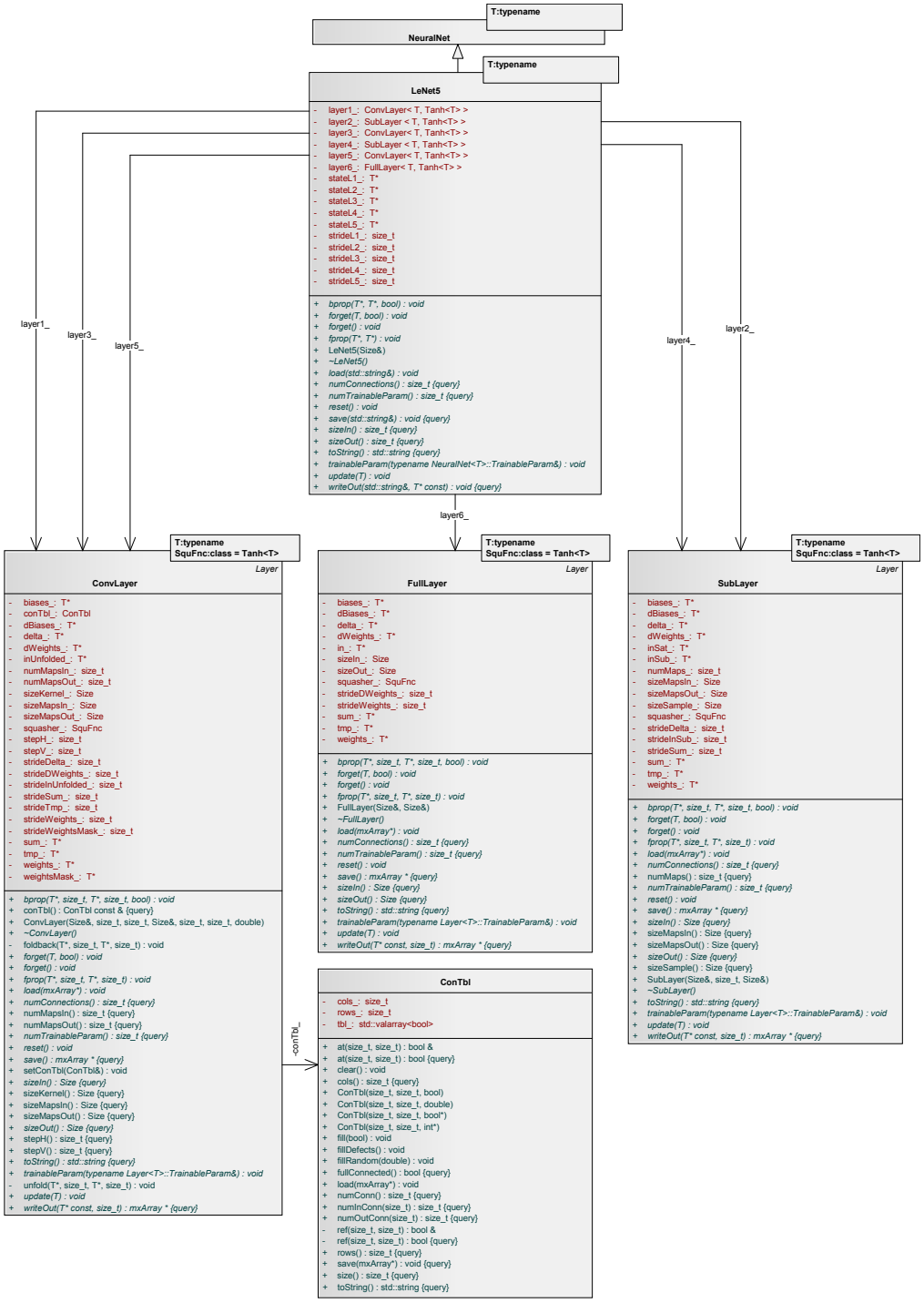


Figure 5.6: LeNet5 class composition.

Chapter 6

Benchmarks and Results

This chapter contains the benchmark results we gained during the work on this thesis (see also [105]). The following Section 6.1 determines how the used numerical precision affects the classification result of a CNN while Section 6.2 compares our implementation with a reference implementation from the New York University. Finally, Section 6.3 covers various benchmarks to show our CNN implementation's performance with a focus on the comparison of scalability between our CPU and GPU implementation.

All experiments and benchmarks in this thesis were performed on an Intel Core i7 860 with a GeForce GTX 275 running Ubuntu 9.04 (system price in fall 2009 approx. € 900)[1]. The technical specifications of these two processors are shown in Table 6.1. As compiler we used the GNU C++ Compiler[2] (g++, version 4.3.3) to generate the CPU executables and NVIDIA's CUDA Compiler (nvcc, version 2.3) to generate the code that runs on the GPU. Compiler optimizations were turned on for the g++ compiler (option -O3). Because we did not observe any performance improvements when turning on the compiler optimization for the nvcc compiler we used the standard settings. For the GPU benchmarks the CUDA driver in version 190.29 was used. The basic input patterns for the benchmarks in this chapter stem from the well known MNIST database [106], which is described in detail in Appendix A.1.

6.1 Numerical Precision

On current x86 processors, like the one used for our benchmarks, calculations in *single precision* (32-bit floating-point numbers) are twice as fast than in *double precision* (64-bit floating-point numbers) when using the *vectorization units*. As mentioned in Chapter 5 we use Intel's performance libraries, which take the advantage of the SSE vectorization units, to speed up our implementation. The execution time of the single and double precision implementation of a LeNet5

[1] The system was provided by our supervisor Dr. Stefan Podlipnig.

[2] `http://gcc.gnu.org`

Processor	Core i7 860	GeForce GTX 275
Processsor Core Clock	2800 MHz	633 MHz
ALU Clock	5600 MHz	1404 MHz
Memory Size	4096 MB	896 MB
Bandwidth Core ↔ Memory	21.3 GB/s	127.0 GB/s
Number of Processor Cores	4	30
Local Cache per Core	64 KB L1	16 KB (shared)
	512 KB L2	8 KB (texture)
	2048 KB L3	8 KB (constant)
SP FLOPS / Core and Clock Cycle	4 MUL or ADD	8 MUL and 8 MAD
Total SP FLOPS Peak Performance	89.6 GFLOPS/s	1010.8 GFLOPS/s
Termal Design Power (TDP)	95 Watt	216 Watt

Table 6.1: Technical specifications of the hardware used for performance measurements in this thesis.

(Subsection 7.1.1), performing one online backpropagation training epoch (Subsection 2.5.1) of the whole MNIST training dataset and the evaluation of the error on the test dataset, is indicated by Figure 6.3. We saw only a minor speedup when switching from double to single precision on our test setup. We suspect that the used activation function tangens hyperbolicus (which's execution takes quite a lot of the total time) is mapped to double precision in any case on our test setup[3]. This anomaly may be not valid for other compiler/CPU combinations, and should be analyzed in a future step. Figure 6.1 shows a numerical comparison of the single and double precision results on the CPU. For this comparison test we ran 500 backpropagation training epochs on a LeNet5 in batch mode (Subsection 2.5.1) and recorded both, classification rate in percent and the MSE (Subsection 2.6.1). As training patterns the first 1,000 images of the MNIST training database were used. There was no difference in terms of classification rates, but slight variations of the MSE. Using the performance libraries ($\text{CPU}_{\text{opt.}}$) led to a different result when using single precision while the results in double precision were equal. Figure 6.1(a) shows the difference of the MSE between single and double precision using the trivial implementation ($\text{CPU}_{\text{triv.}}$), Figure 6.1(b) the variations when using the optimized one ($\text{CPU}_{\text{opt.}}$). In both single precision implementations the difference gets smaller the longer the training takes.

Recognizing that there is nearly no difference between the single and double precision implementation, we performed all further CPU benchmarks in single precision only. Considering that double precision is only poorly supported on today's GPUs we omitted to implement the GPU versions of our networks in double precision, all CUDA based implementations are therefore in single precision only. However, the GPUs we used to test our networks do not support full

[3]Thanks to Dr. Alfred Strey for this suggestion.

IEEE 754-2008[4] (IEEE Standard for Floating-Point Arithmetic) floating-point number standard. They lack the support of *denormalized numbers* [11], which means that every number with an absolute value smaller than 2^{-126} is rounded to 0. As shown in Figure 6.2 the mutilation of the floating-point number format standard has only a minor influence on the MSE of our tested network.

6.2 Comparison to Reference Implementation

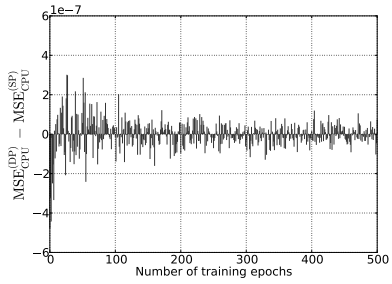
To see how our implementation performs in comparison to others we looked for a reference implementation of CNNs. We found one in the free C++ Machine Learning Library called EBLearn [107, 108], which stands for Energy-Based Learning. It was developed by the New York University's Computational and Biological Learning Laboratory[5], led by Yann LeCun, the inventor of CNNs.

We decided to use the LeNet5 (Subsection 7.1.1) and the MNIST database to build a test environment. A short training time is very important in order to determine the best settings for a given problem. Figure 6.3 illustrates how long one backpropagation training epoch in online mode (Subsection 2.5.1) of the whole MNIST training dataset (60,000 patterns) and the evaluation of the error on the test dataset (10,000 patterns) takes using the different programs and how big the speedup of our implementations is compared to the one in the EBLearn library. The EBLearn implementation of the LeNet5 was tested in *double precision* (DP) only because it is preset when you download it and *single precision* (SP) is not supported by default. It clearly shows the superiority of our CPU implementations in terms of execution speed in comparison to the EBLearn implementation. However, our GPU implementation is even faster.

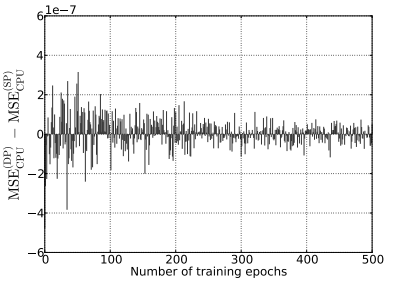
6.3 Performance and Scalability Tests

In this section we want to determine how our CPU and GPU implementations scale with various network sizes. All performance and scalability tests in this section consist of 1,000 online backpropagation training iterations of two different networks, the LeNet5 (Figure 3.1) and the SimardNet (Figure 7.3), described in detail in Subsection 7.1.1 and 7.1.2, respectively. As indicated in Subsection 2.5.1 such a training iteration is composed of one forward propagation (a training pattern is fed into the network and produces some output), one backpropagation (based on the difference between the actual and the desired output, a gradient for every single weight in the network is calculated) and the

[4] `http://grouper.ieee.org/groups/754`
[5] `http://www.cs.nyu.edu/~yann/index.html`

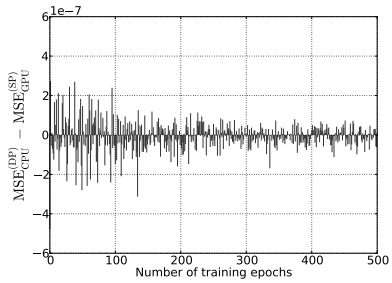


(a) MSE of $\mathrm{CPU}_{\mathrm{triv.}}$ DP vs. SP implementation.

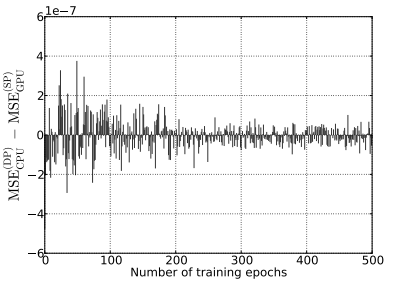


(b) MSE of $\mathrm{CPU}_{\mathrm{opt.}}$ DP vs. SP implementation.

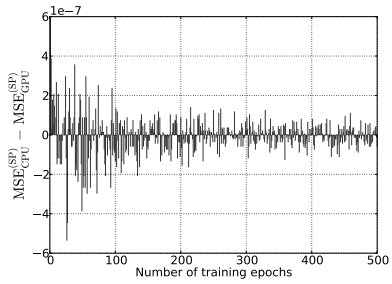
Figure 6.1: Comparison of the MSE between SP and DP implementation on the CPU during the backpropagation training of a LeNet5 in batch mode using the first 1,000 patterns of the MNIST training dataset.



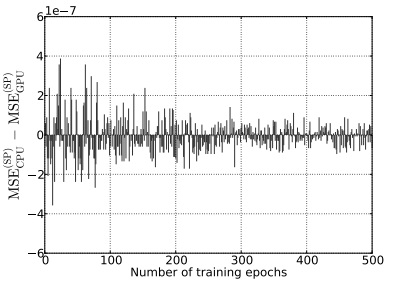
(a) MSE of $\mathrm{CPU}_{\mathrm{triv.}}$ DP vs. GPU SP implementation.



(b) MSE of $\mathrm{CPU}_{\mathrm{opt.}}$ DP vs. GPU SP implementation.

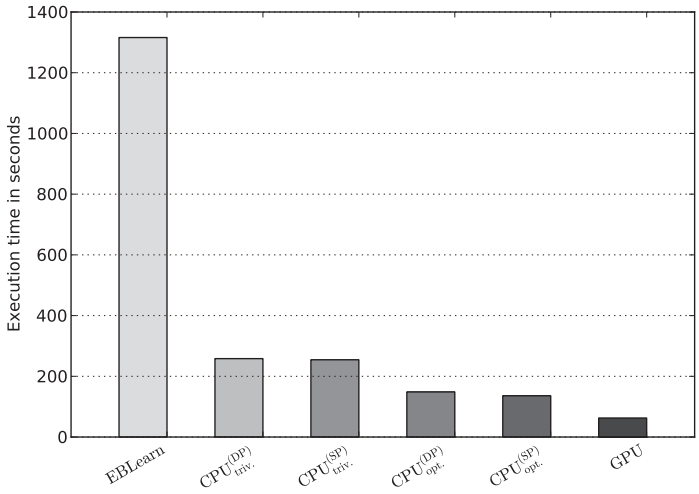


(c) MSE of $\mathrm{CPU}_{\mathrm{triv.}}$ SP vs. GPU SP implementation.

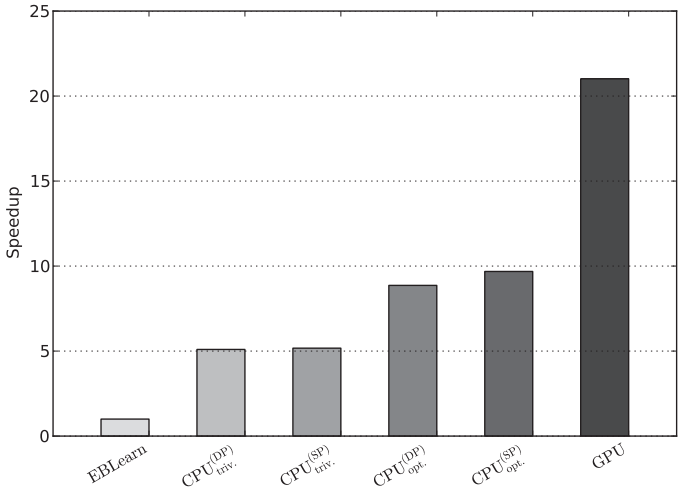


(d) MSE of $\mathrm{CPU}_{\mathrm{opt.}}$ SP vs. GPU SP implementation.

Figure 6.2: Comparison of the MSE between SP/DP CPU and SP GPU implementation during the backpropagation training of a LeNet5 in batch mode using the first 1,000 patterns of the MNIST training dataset.



(a) Execution time of one backpropagation training epoch in online mode of the whole MNIST training dataset and the evaluation of the error on the test dataset.



(b) Speedup of the implementations introduced in this thesis in comparison to the one in the EBLearn library.

Figure 6.3: Comparison of the execution time between the LeNet5 implementation in the EBLearn library [107, 108] and our implementations using the MNIST database.

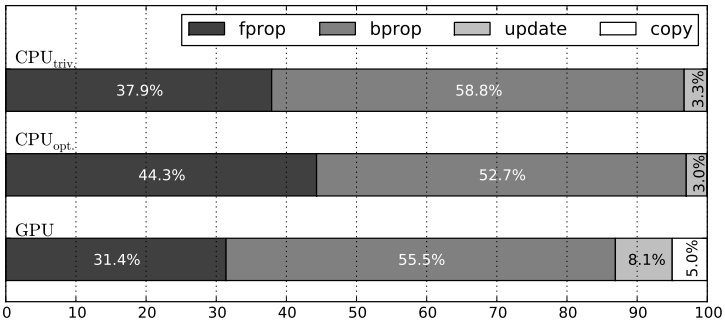


Figure 6.4: Execution time composition of the three different implementations performing 1,000 online backpropagation learning iterations of a LeNet5 using the MNIST database.

weights update (the gradients calculated during backpropagation are multiplied with the learning rate and added to the actual weights). In case of the CUDA version the time to copy the training pattern to the GPU and the result to the main memory is also considered. For each iteration a separate input pattern of the MNIST training database was used. Note that we were interested in the scaling behavior of our implementations. To test this behavior with different settings we decided to restrict our tests to the basic case (one iteration for each pattern) and a small number of input patterns (1,000). In all benchmarks we compared the three different implementations ($CPU_{triv.}$, $CPU_{opt.}$, GPU) explained in Chapter 5.

6.3.1 Composition of Execution Time

In our first experiment we investigated the partition of the LeNet5's training time for the three different CNN versions. The result is shown in Figure 6.4. As one can clearly see, the backpropagation part (bprop) takes most of the overall time in all three versions. The forward propagation (fprop) is quite time-consuming too while the weight update (update) takes only a small part of the total execution time. Although the absolute execution time is quite different the percent distribution of the different parts is rather similar (except the weight update for the GPU version). The overhead caused by copying data (copy) to and from the GPU is quite small in the GPU version (only a few percentages of the overall execution time). The behavior of the SimardNet's training is rather similar and therefore not indicated separately.

6.3.2 Scaling Input Size

In this benchmark we scaled the input size of the training patterns fed into a LeNet5. Increasing the input size automatically increases the number of neurons in the convolutional and subsampling layers and the number of trainable parameters (weights and biases). The input's side length was increased stepwise by eight pixels as shown in Table 6.2 on page 92. Figure 6.5(a) on page 93 shows the execution time of all three implementations with different input sizes. While the CPU version using Intel's performance libraries ($\text{CPU}_{\text{opt.}}$) clearly outperforms the trivial implementation ($\text{CPU}_{\text{triv.}}$), the CUDA version (GPU) is not only the fastest one but it also scales best with the input size. This is underlined by Figure 6.5(b) on page 93 which shows the speedup of the GPU version in comparison to the trivial CPU version ($\text{CPU}_{\text{triv.}}$/GPU) and to the optimized CPU version ($\text{CPU}_{\text{opt.}}$/GPU). The speedup grows with the input size and the GPU version definitely scales better than the CPU versions with large input sizes.

6.3.3 Scaling Feature Maps and Inner Neurons

The last benchmark shows how the implementations scale with the number of neurons inside the network. We tested the SimardNet with all possible combinations resulting from doubling the feature maps/neurons of all inner layers. The input size remained fixed at 29×29, while the output size was set to 10. The properties of the tested network's variants are listed in Table 6.3 on page 92. As shown in Figure 6.6(a) and 6.6(b) on page 94 the optimized CPU variant ($\text{CPU}_{\text{opt.}}$) scales better than the trivial one ($\text{CPU}_{\text{triv.}}$) when doubling the feature maps in one of the first two inner layers or the neurons in the third inner layer. However, the GPU version indisputable scales best with the size of any inner layer.

Input size [a]	Network properties				Execution time (in seconds)			Speedup	
	Input area	Neurons	Trainable parameters	Connections	$CPU_{triv.}$	$CPU_{opt.}$	GPU	$\frac{CPU_{triv.}}{GPU}$	$\frac{CPU_{opt.}}{GPU}$
32×32	1,024	8,010	51,046	331,114	3.6567	2.0317	0.9345	3.9132	2.1742
40×40	1,600	13,770	60,646	956,842	7.2532	3.6693	1.1103	6.5326	3.3048
48×48	2,304	21,130	79,846	2,047,210	13.0610	5.9001	1.3388	9.7554	4.4068
56×56	3,136	30,090	108,646	3,602,218	21.1530	8.7290	1.6000	13.2206	5.4556
64×64	4,096	40,650	147,046	5,621,866	32.5300	12.1500	1.9901	16.3463	6.1054
72×72	5,184	52,810	195,046	8,106,154	44.5400	16.5210	2.3532	18.9274	7.0207
80×80	6,400	66,570	252,646	11,055,082	59.4970	21.3960	2.9072	20.4650	7.3595
88×88	7,744	81,930	319,846	14,468,650	76.3710	26.5900	3.5705	21.3894	7.4471
96×96	9,216	98,890	396,646	18,346,858	95.1780	33.1480	4.1635	22.8601	7.9616
104×104	10,816	117,450	483,046	22,689,706	116.4700	40.8270	4.9055	23.7427	8.3227

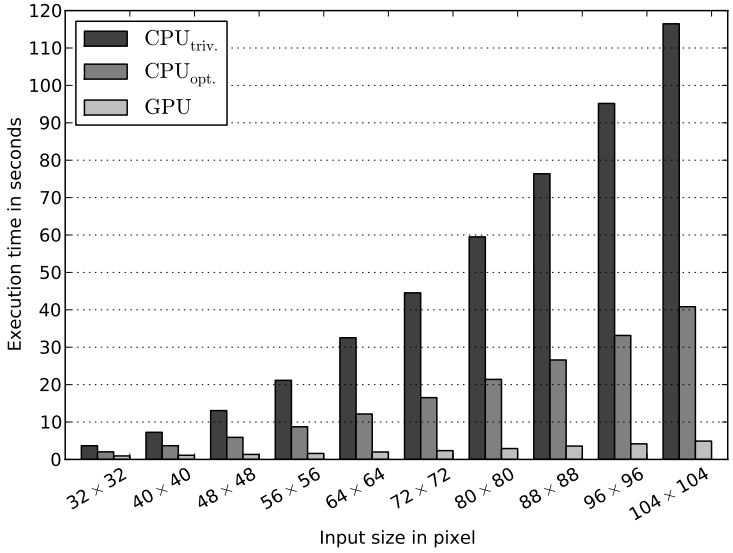
[a] width×height of the input pattern

Table 6.2: Properties of the tested LeNet5 variants.

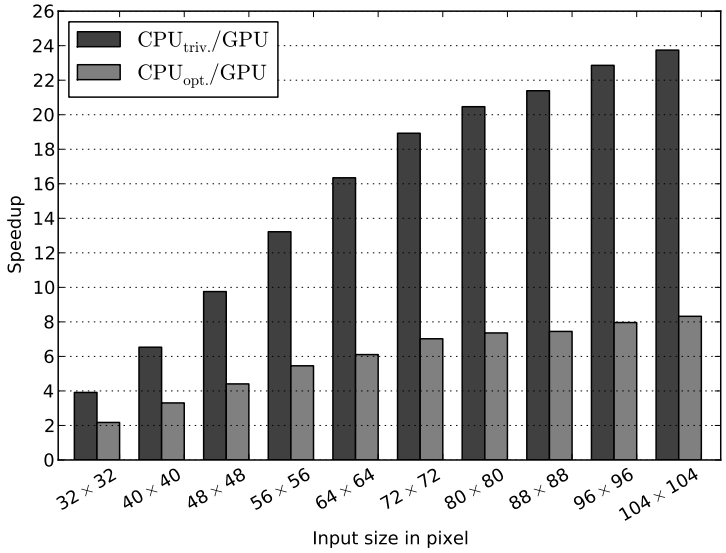
Architecture [a]	Network properties				Execution time (in seconds)			Speedup	
	Input area	Neurons	Trainable parameters	Connections	$CPU_{triv.}$	$CPU_{opt.}$	GPU	$\frac{CPU_{triv.}}{GPU}$	$\frac{CPU_{opt.}}{GPU}$
5,50,100,10	841	2,205	132,540	305,580	1.8364	0.9632	0.5487	3.3470	1.7556
5,50,200,10	841	2,305	258,640	431,680	2.5266	1.3383	0.6361	3.9722	2.1040
5,100,100,10	841	3,455	263,840	588,080	3.3291	1.5949	0.6574	5.0643	2.4262
5,100,200,10	841	3,555	514,940	839,180	4.7236	2.3592	0.7998	5.9062	2.9499
10,50,100,10	841	3,050	138,920	483,800	2.6923	1.1659	0.6235	4.3184	1.8701
10,50,200,10	841	3,150	265,020	609,900	3.3987	1.5546	0.7113	4.7785	2.1857
10,100,100,10	841	4,300	276,470	922,550	4.8025	1.8419	0.7129	6.7369	2.5838
10,100,200,10	841	4,400	527,570	1,173,650	6.1908	2.6085	0.8529	7.2589	3.0586

[a] number of feature maps in layer 1, 2 and number of neurons in layer 3, 4

Table 6.3: Properties of the tested SimardNet variants.

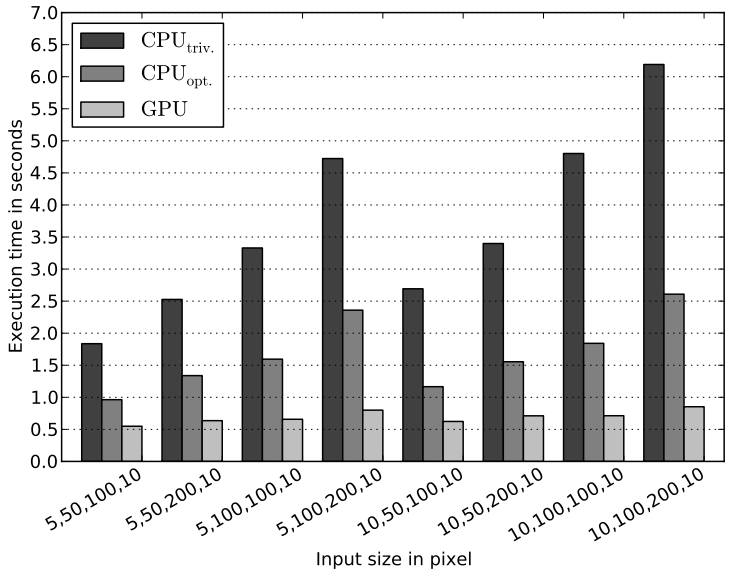


(a) Execution time

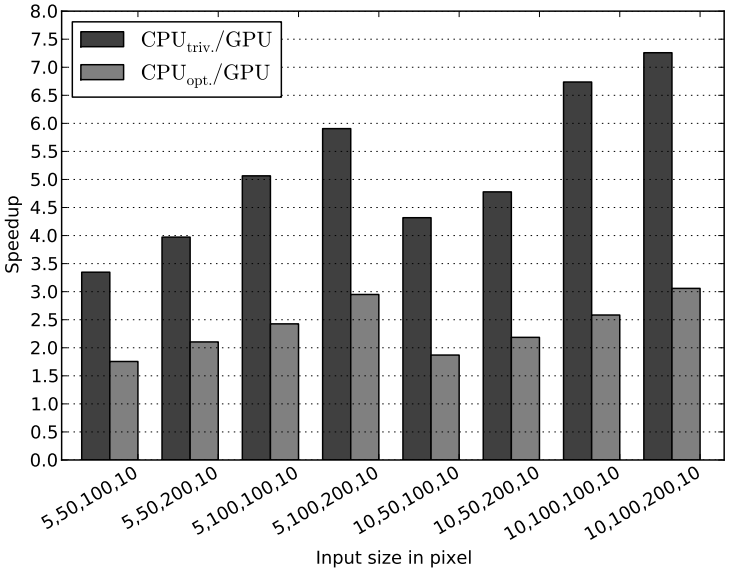


(b) Speedup

Figure 6.5: Execution time (a) and corresponding speedup (b) of the three different implementations performing 1,000 online backpropagation learning iterations of a LeNet5 using the MNIST database.



(a) Execution time



(b) Speedup

Figure 6.6: Execution time (a) and corresponding speedup (b) of the three different implementations performing 1,000 online backpropagation learning iterations of a SimardNet using the MNIST database.

Chapter 7

Applications of CNNs

Neural networks in general perform very well on pattern recognition and classification tasks with a large amount of training data. CNNs in particular are optimized for two-dimensional problems of such kind. The area of applications for CNNs is widespread. They are used for handwriting recognition [1, 2], face, eye and license plate detection [3–9], and in non-vision applications such as semantic analysis [10].

The following two sections will describe how neural networks build with our library (Chapter 5) perform on different tasks. The performance of various CNNs in terms of classification rates on handwritten digits are demonstrated in Section 7.1, while Section 7.2 describes how CNNs can be used to build a face detection system.

7.1 Handwritten Digit Recognition

Optical character recognition (OCR) is a very difficult task in the area of pattern recognition. There exist many different approaches to solve this problem, e.g. principal component analysis (PCA) and Fisher discriminant analysis [28], support vector machines (SVM) [109], and neural networks [110, 111]. Most approaches use feature extractors to reduce the input data and maintain all relevant information. As described in Chapter 3, CNNs make this feature extractor part of the network and therefore also part of the training procedure. This makes the use of CNNs rather easy because no additional feature extraction is needed. As stated in [1] and shown in Table A.1 on page 123 CNNs deliver state-of-the-art performance on OCR tasks.

The following subsections will show the performance of three different CNNs implemented in scope of this thesis on the MNIST handwritten digit database (Appendix A.1). Since this database features a lot of training patterns (60,000) we used online backpropagation training (Subsection 2.5.1) in all of these benchmarks, since it performs better than training in batch mode when using large training sets [34]. The networks were trained until the early stopping procedure,

described in Subsection 2.5.5, reached its break condition. The first 54,000 patterns of the MNIST training dataset were used for training, the last 6,000 (10%) as validation set. The MNIST database also features a separate test dataset of 10,000 patterns which was used to evaluate the classification error of the implemented networks.

7.1.1 LeNet5

The *LeNet5* [2] is one of the networks proposed by Yann LeCun, the inventor of the CNNs[1]. Since our library (Chapter 5) implements only layers featuring sigmoidal neurons we used a slight variation of it omitting the RBF units in the last layer, used by the original concept in [2]. This resulted in a six layer CNN, which's structure is shown in Figure 3.1 on page 28. The input consists of a 32×32 pixel grayscale image. Therefore we had to enlarge the 28×28 images of the MNIST database to this size by adding some background pixels around them.

The first layer C1 of the LeNet5 is a convolutional layer consisting of 6 feature maps. Each of them has a size of 28×28 neurons and uses a convolution kernel of size 5×5 with a step size of one. Then follows a subsampling layer S2 with 6 feature maps. It uses a 2×2 subsampling kernel which leads to a feature map size of 14×14 neurons. It is succeeded by another convolutional layer C3. It uses again a 5×5 convolution kernel with a step size of one and consists of 16 feature maps with a size of 10×10 neurons each. It is followed by a subsampling layer S4 which shrinks the feature maps down to a size of 5×5 neurons using a 2×2 subsampling kernel. The fifth layer F5 is a fully connected layer and consists of 120 neurons while the output layer F6 contains 10 neurons, each of them representing one of the 10 different classes (digits). The LeNet5 uses sparse connections between the second and the third layer, where the connection scheme in form of a *connection matrix* is shown in Table 7.1. As stated in [2] sparse connections break the symmetry in the network and often yields to better results. Having different inputs, the distinct feature maps of layer C3 extract different features. This network consists of 8,010 neurons and its total number of trainable parameters (weights and biases) and connections is 51,046 and 331,114, respectively.

The following paragraph shows the classification performance of the implemented LeNet5 on the MNIST database. In this benchmark MSE (Subsection 2.6.1) was used as error function and the learning rate η (Subsection 2.5.1) was initialized with 0.003 and reduced by 10% every 20th training epoch. We

[1] For a demo of the LeNet5 network applied to handwritten digit recognition see: `http://yann.lecun.com/exdb/lenet/index.html`.

		Feature maps in layer C3															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Feature maps in layer S2	0	×				×	×	×			×	×	×	×		×	×
	1	×	×				×	×	×			×	×	×	×		×
	2	×	×	×				×	×	×			×		×	×	×
	3		×	×	×			×	×	×	×			×		×	×
	4			×	×	×			×	×	×	×		×	×		×
	5				×	×	×			×	×	×	×		×	×	×

Table 7.1: Connection matrix between the feature maps of the second and the third layer in the LeNet5.

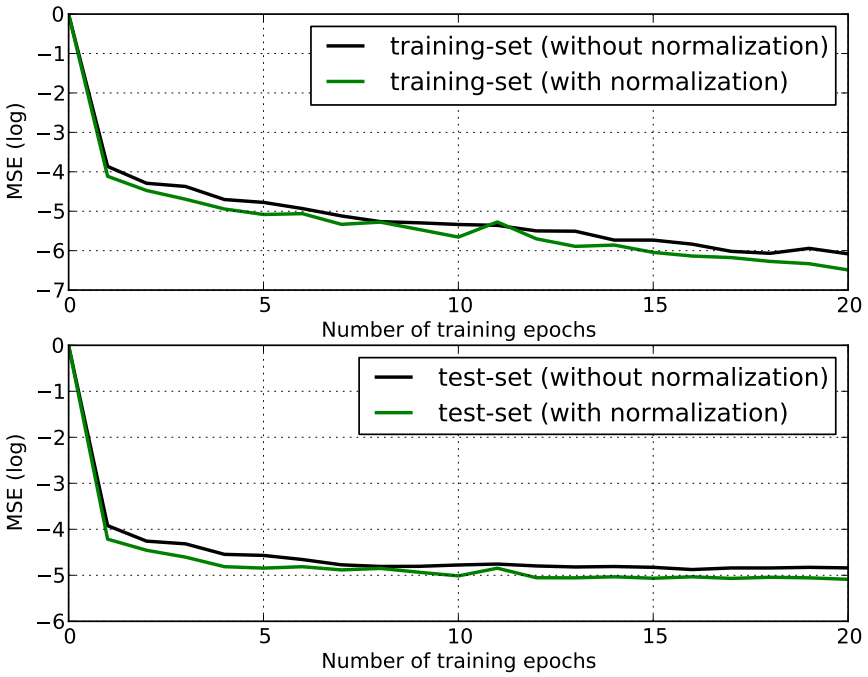


Figure 7.1: Illustration how the input data normalization (Subsection 2.5.4) improves the backpropagation training of the MNIST database by a LeNet5.

Run	Stop.	training dataset		validation dataset		test dataset	
	Iter.	MSE	error %	MSE	error %	MSE	error %
1	33	0.0007	0.1315	0.0071	1.1333	0.0070	1.0000
2	24	0.0008	0.1389	0.0061	0.8000	0.0082	1.1800
3	30	0.0008	0.1611	0.0071	0.9667	0.0080	1.1500
4	55	0.0005	0.1111	0.0061	0.8500	0.0072	1.0500
5	26	0.0008	0.1463	0.0065	0.7667	0.0077	1.0500
avg.		**0.0007**	**0.1378**	**0.0066**	**0.9033**	**0.0076**	**1.0860**

Table 7.2: Result of several training runs of a LeNet5 on the MNIST database.

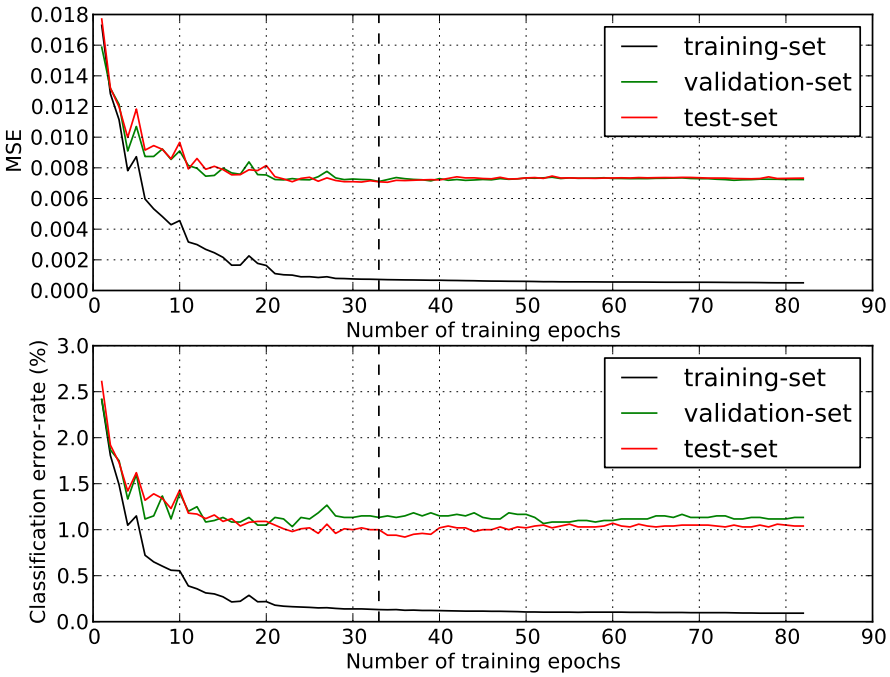


Figure 7.2: Development of the error on all three datasets during the learning of the MNIST database by a LeNet5. The dashed line indicates the epoch with the best performance on the validation set.

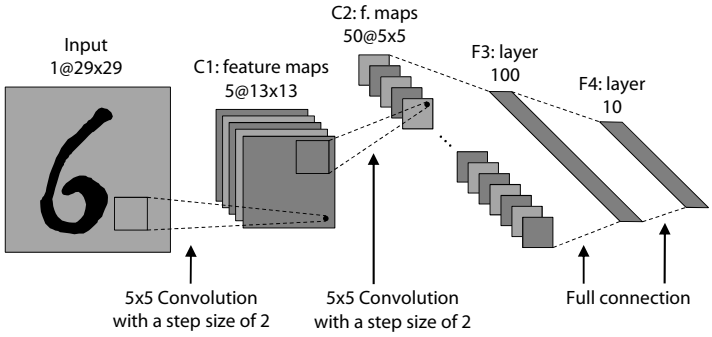


Figure 7.3: Architecture of the network proposed by Simard et al. in [1], one of the best performing neural networks on the MNIST test dataset with a classification error of 0.4%.

used the tangens hyperbolicus (Subsection 2.7.3) as activation function in all layers and the weights were initialized using the method described in Subsection 2.5.3. The training was terminated after 50 epochs without reduction of the MSE on the validation dataset. The achieved results are shown in Table 7.2 while Figure 7.2 shows how the error evolves on the training, validation, and test dataset over the training epochs (taken from the training number 1 in Table 7.2 which resulted in the best classification error rate equal to 99% on the test dataset).

How the input data normalization as described in Subsection 2.5.4 improves the backpropagation training of the MNIST database by a LeNet5 is shown in Figure 7.1. For this comparison test we ran two LeNet5 networks initialized with the same random weight values for 20 (online) backpropagation training epochs and recorded the MSE. While for the first network the input patterns were taken directly from the MNIST database without any transformation, the input patterns for the second network were normalized according to the method described in Subsection 2.5.4.

7.1.2 SimardNet

Simard et al. introduced a CNN in [1] which achieved one of the best classification results (0.4%) on the MNIST test dataset (see also Table A.1 on page 123). The structure of this network is somehow different than the one of the LeNet5. It uses no subsampling layers but instead the convolutional layers use a step size of two in their convolution to reduce the size of consecutive feature maps. Leading to a four layer architecture illustrated in Figure 7.3. This network is called *SimardNet* from now on. It takes as input a 29×29 pixel grayscale im-

age, therefore the MNIST database's patterns have to be padded to this size by adding some background pixels.

The SimardNet's first convolutional layer C1 consists of five feature maps, each housing a plane of 13×13 neurons and using a 5×5 convolution kernel. The consecutive convolutional layer C2 contains 50 feature maps bringing, down their size to 5×5 neurons using a 5×5 convolution kernel with a step size of two. The third layer F3 is a fully connected layer formed by 100 neurons while the output layer F4 has 10 neurons to distinguish between the ten classes in the MNIST database. The SimardNet consists of 2,205 neurons, 132,540 trainable parameters (weights and biases) and 305,580 connections between the neurons.

Below we will show how the SimardNet performs on the MNIST database. The learning rate η starts with a value of 0.005 and is multiplied by 0.3 every 100 training epochs, as noted in [1]. All weights were initialized with random numbers from a uniform distribution between ± 0.05. Early stopping was used to define the training's break condition, where the number of epochs without improvement of the CE on the validation dataset was set to 50. The results in Table 7.3 and Figure 7.4 (which shows the development of the error of training run 4 in Table 7.3 leading to the highest classification rate) were gained using CE (Subsection 2.6.2) as error function and tangens hyperbolicus (Subsection 2.7.3) as activation function in all layers.

7.1.3 SimardNet with Distortions

In [1] the authors suggest to enlarge the training dataset by some distorted patterns as described in Appendix B to enhance the classification rate of a NN. Table 7.5 shows how the SimardNet performs on the MNIST database when using *elastic distortions*. The parameters for the network are the same as described in the previous subsections. For this benchmark we added 9 distorted patterns for each pattern of the training dataset. These patters were regenerated for each new training epoch. The distortion properties σ and α described in Appendix B were set to 4 and 22, respectively.

Figure 7.5 shows how the error curves of training number 4 in Table 7.5 evolve. The error curve of the training dataset always includes the classification error on the distorted images generated in the respective epoch. We clearly recognize, that the error declines faster in the first epoch than without distortions, mainly because the training dataset is now ten times larger. The error curves are not as smooth as in the previous benchmarks without distortions. Especially on the training dataset (which includes the distorted patterns) we see heavy fluctuations which emerge from the randomness introduced by the distortions. These up- and downturns of the error implicated to raise the number of epochs

Run	Stop.	training dataset		validation dataset		test dataset	
	Iter.	CE	error %	CE	error %	CE	error %
1	119	0.7994	0.1741	0.8172	0.8833	0.8230	1.2100
2	76	0.7993	0.1759	0.8198	1.0333	0.8219	1.2500
3	97	0.7993	0.1741	0.8197	0.9000	0.8226	1.2600
4	100	0.7991	0.1574	0.8183	0.8500	0.8224	1.1700
5	50	0.7997	0.1907	0.8173	1.0000	0.8230	1.1800
avg.		**0.7994**	**0.1744**	**0.8185**	**0.9333**	**0.8226**	**1.2140**

Table 7.3: Result of several training runs of a SimardNet on the MNIST database.

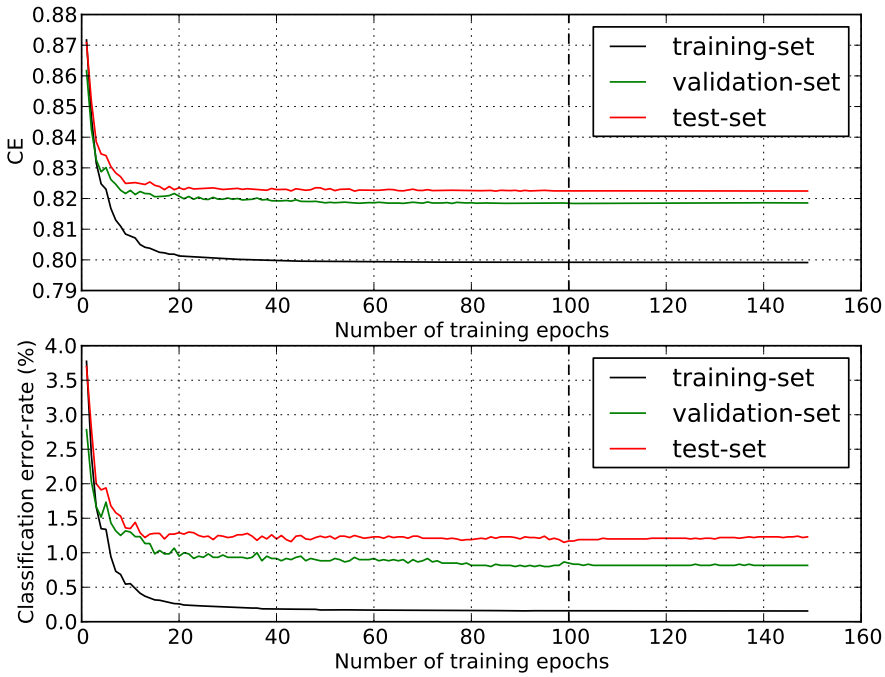


Figure 7.4: Development of the error on all three datasets during the learning of the MNIST database by a SimardNet. The dashed line indicates the epoch with the best performance on the validation set.

Actual	Predicted Class										Classification
Class	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	Rate (%)
0	976	0	1	0	0	1	0	1	0	1	**99.59**
1	0	1129	2	1	0	0	1	1	1	0	**99.47**
2	1	1	1028	0	0	0	0	2	0	0	**99.61**
3	0	0	0	1008	0	1	0	1	0	0	**99.80**
4	0	0	0	0	978	0	0	0	0	4	**99.59**
5	0	0	0	4	0	884	1	1	2	0	**99.10**
6	3	3	0	0	0	3	947	0	2	0	**98.85**
7	0	1	3	0	0	0	0	1022	0	2	**99.42**
8	0	0	1	2	0	1	0	0	969	1	**99.49**
9	0	1	0	2	6	2	0	1	1	996	**98.71**
Overall Classification Rate											**99.37**

Table 7.4: Confusion matrix illustrating the classification performance of a SimardNet trained with distortions on the test dataset of the MNIST database.

to wait for improvement before stopping the training to 100. Overall we see that the error rate on the distorted training dataset is quite high, but the error on the validation and test dataset is definitely lower than without distortions.

Table 7.4 shows the so-called *confusion matrix* of a SimardNet applying the weights gathered in training run number 4 (see Table 7.5) on the test dataset of the MNIST database. This matrix is of size 10×10, where each row represents an actual class (0, 1, ..., 9) in the database while each column stands for the *class-affiliation* predicted by the CNN. The data elements in the matrix show the frequency of the network's class predictions separate for each digit. The third element in the eighth row shows for example how often a pattern representing a 7 has been misclassified as a 2. In addition to these digits we added another column, holding the recognition rate separately for each digit from 0 to 9. We see that class 3 achieved the best performance with a classification rate of 99.80%, while 9 is the digit which is misclassified most frequently (in 1.29% of all cases). We also see that the network has the biggest problems to distinguish between 4 and 9, with six nines recognized as a four and four fours classified as a nine.

7.1.4 CNNs using Random Connections

As mentioned in Subsection 7.1.1 sparse connections between feature maps of consecutive layers should enhance the classification result. Michael Jahrer[2] advanced this approach in his unpublished paper [51] by introducing *random connections* between feature maps of consecutive layers. He presented a five layer CNN that will be called *JahrerNet* from now on. This network is optimized

[2] `http://www.commendo.at`

Run	Stop.	training dataset		validation dataset		test dataset	
	Iter.	CE	error %	CE	error %	CE	error %
1	398	0.8086	1.0937	0.8048	0.4833	0.8076	0.6800
2	231	0.8049	0.8791	0.8041	0.5333	0.8064	0.6400
3	317	0.8117	1.2902	0.8053	0.6333	0.8066	0.5900
4	267	0.8049	0.8794	0.8053	0.6000	0.8069	0.6300
5	317	0.8085	1.0674	0.8052	0.5000	0.8071	0.6900
avg.		**0.8077**	**1.0420**	**0.8050**	**0.5500**	**0.8069**	**0.6460**

Table 7.5: Result of several training runs of a SimardNet on the MNIST database which's training patterns have been enlarged by elastic distortions.

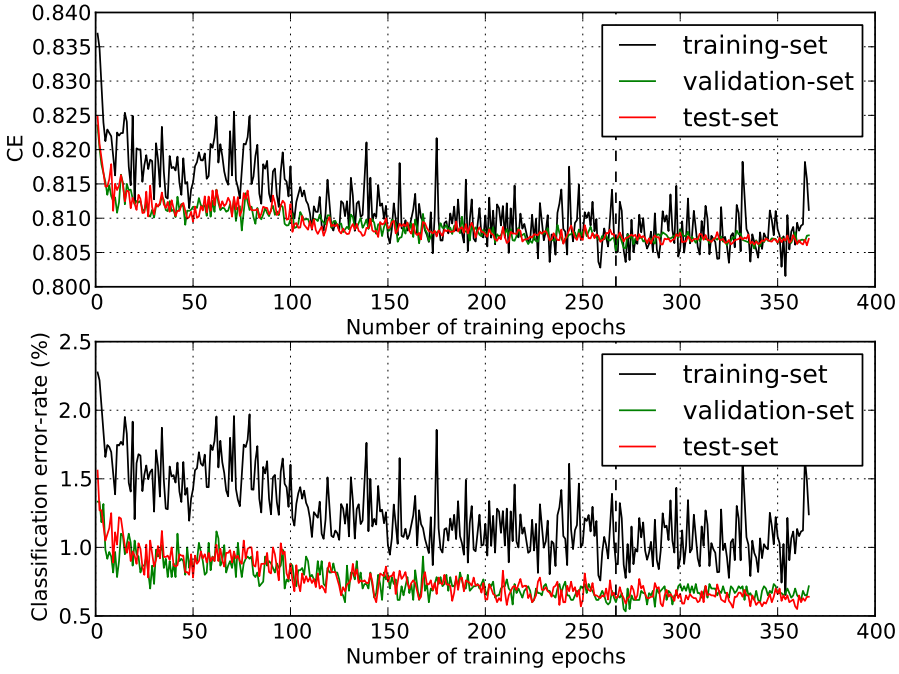


Figure 7.5: Development of the error on all three datasets during the learning of the MNIST database which's training patterns have been enlarged by elastic distortions by a SimardNet. The dashed line indicates the epoch with the best performance on the validation set.

for the MNIST database. It takes a 28×28 pixel grayscale image as input. Therefore, the patterns in the MNIST database can remain unchanged.

The JahrerNet's first layer C1 is a convolutional layer housing 20 feature maps of size 24×24 neurons and a convolution kernel of 5×5 which are mapped on the input using a step size of one. It is followed by a subsampling layer S2 with the same number of feature maps. Due to its subsampling kernel size of 2×2 they consist of 12×12 neurons. The third layer C3 is again a convolutional layer. Its 5×5 convolution kernel with step size one brings the feature maps size down to 8×8 neurons. The number of feature maps in the third and fourth layer is 80. The fourth layer S4 is a subsampling layer and shrinks the feature maps to a size of 4×4 neurons using a 2×2 subsampling kernel. The fifth layer C5 is a convolutional layer with 350 feature maps and a kernel size of 4×4. This means that the kernel is only applied once to each feature map. The output layer F6 contains 10 neurons to implement a 1-of-10 coding for the ten different digits in the MNIST database.

The connectivity between the feature maps in layer S2 and C3 respectively S4 and C5 is defined by randomly generated connection tables. The probability for a connection between two feature maps of layer S2 and C3 is 0.14, the one between layer S4 and C5 is 0.2. The only constraint when generating the connection tables is that all feature maps must have at least one connection. The JahrerNet consists in total of 21,160 neurons. The number of trainable parameters (weights and biases) and connections varies due to the randomness in the connectivity, it has an average of 100,000 trainable parameters and approximately 770,000 connections. These numbers are quite high compared to the two networks introduced earlier in this thesis. This makes training and recognition rather slow. The high number of feature maps is necessary because of the relative low connectivity inside the network.

We performed some tests on the MNIST database using the JahrerNet. For each of these tests we generated new connection tables. The initial learning rate η was set to 0.003 and it was decreased by 10% every 20th training epoch. Table 7.6 shows the result of this training runs, the error curves of training number 3 are shown in Figure 7.6. This test clearly shows that this network using randomized sparse connections between feature maps of consecutive layers delivers superior recognition performance on the MNIST database than networks using fixed (full or sparse) connections. However, it needs a larger network to achieve maximum performance which penalizes the execution speed.

Run	Stop.	training dataset		validation dataset		test dataset	
	Iter.	MSE	error %	MSE	error %	MSE	error %
1	25	0.0007	0.1426	0.0053	0.7000	0.0069	0.9200
2	247	0.0004	0.0852	0.0055	0.6333	0.0065	0.8500
3	87	0.0004	0.0889	0.0057	0.7500	0.0061	0.8000
4	94	0.0004	0.0852	0.0056	0.8000	0.0060	0.8100
5	32	0.0006	0.1185	0.0054	0.7667	0.0067	0.8900
avg.		**0.0005**	**0.1041**	**0.0055**	**0.7300**	**0.0064**	**0.8540**

Table 7.6: Result of several training runs of a JahrerNet on the MNIST database.

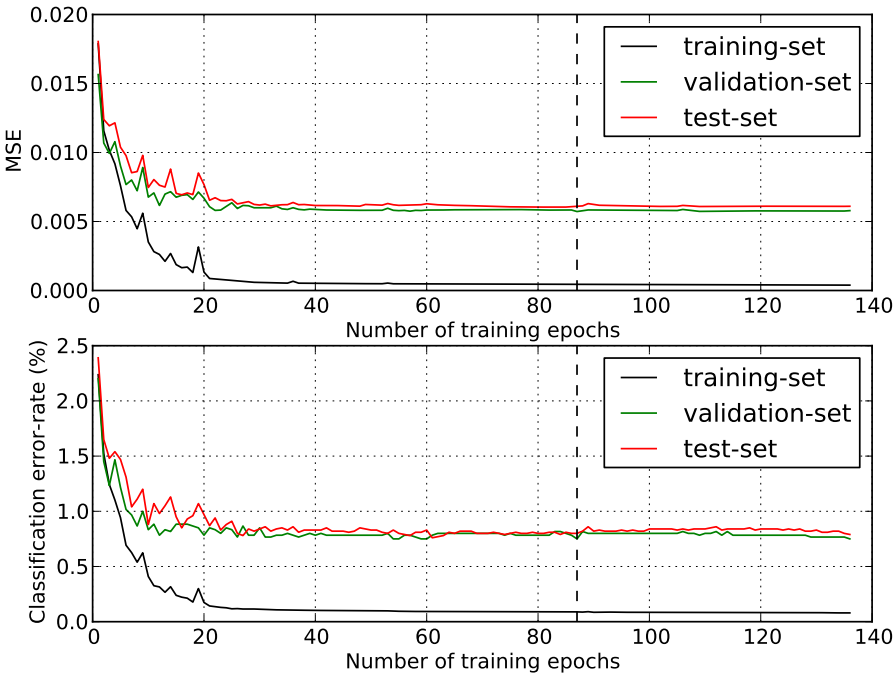


Figure 7.6: Development of the error on all three datasets during the learning of the MNIST database by a JahrerNet. The dashed line indicates the epoch with the best performance on the validation set.

7.2 Convolutional Face Detector

This section shows how a human face detection system can be build using CNNs. The idea behind this system is based on the work of Garcia and Delakis presented in [3–5] and the work of Tivive and Bouzerdoum presented in [6, 7], but in a simplified form. Finding human faces in an image is very difficult due to the large variety of distortions one has to take into account. The faces can be presented in different facial expressions under different environment conditions and camera perspectives. CNNs are therefore very appropriate for this kind of task because they provide partial invariance to translation, rotation, scaling, and other deformations of the input patterns (see also Chapter 3). The face detection is not the only application where CNNs were applied successfully. Also other image-based object detection applications were build using CNNs, e.g. eye and license plate detection [6, 8, 9] or lung nodule detection in the medicine [112].

The presented face detection system is based on a CNN which has been trained to distinguish between face and non-face patterns. This network is then applied to an image in order to localize the human faces. The following subsections describe the architecture of the used network, how it was trained, and how it can be used to detect human faces inside an image.

7.2.1 Network Architecture

The used CNN in this face detection system is rather similar compared to the one in [3–5], but using a different input size of 32×32 pixels only. The limitation to this input size results from the size of the training patterns of the used database described in the following subsection. The architecture of this network, called *GarciaNet* from now on, is quite similar to the one of the LeNet5 (Subsection 7.1.1). The GarciaNet illustrated in Figure 7.7 is a six layer CNN, where the first four layers act as a feature extractor and the last two layers as a classifier.

The first layer C1 of the GarciaNet is a convolutional layer consisting of 4 feature maps. Each of them has a size of 28×28 neurons and uses a convolution kernel of size 5×5 with a step size of one. This layer is followed by a subsampling layer S2 using a 2×2 subsampling kernel which leads to a feature map size of 14×14 neurons. The third layer C3 is again a convolutional layer using a convolution kernel of size 3×3 and a step size of one. It consists of 14 feature maps with a size of 12×12 neurons which are partially connected to the feature maps in the preceding subsampling layer S2 according to the connection matrix in Table 7.7. Then follows a subsampling layer S4 which shrinks the feature maps down to a size of 6×6 neurons using a 2×2 subsampling kernel. The fifth

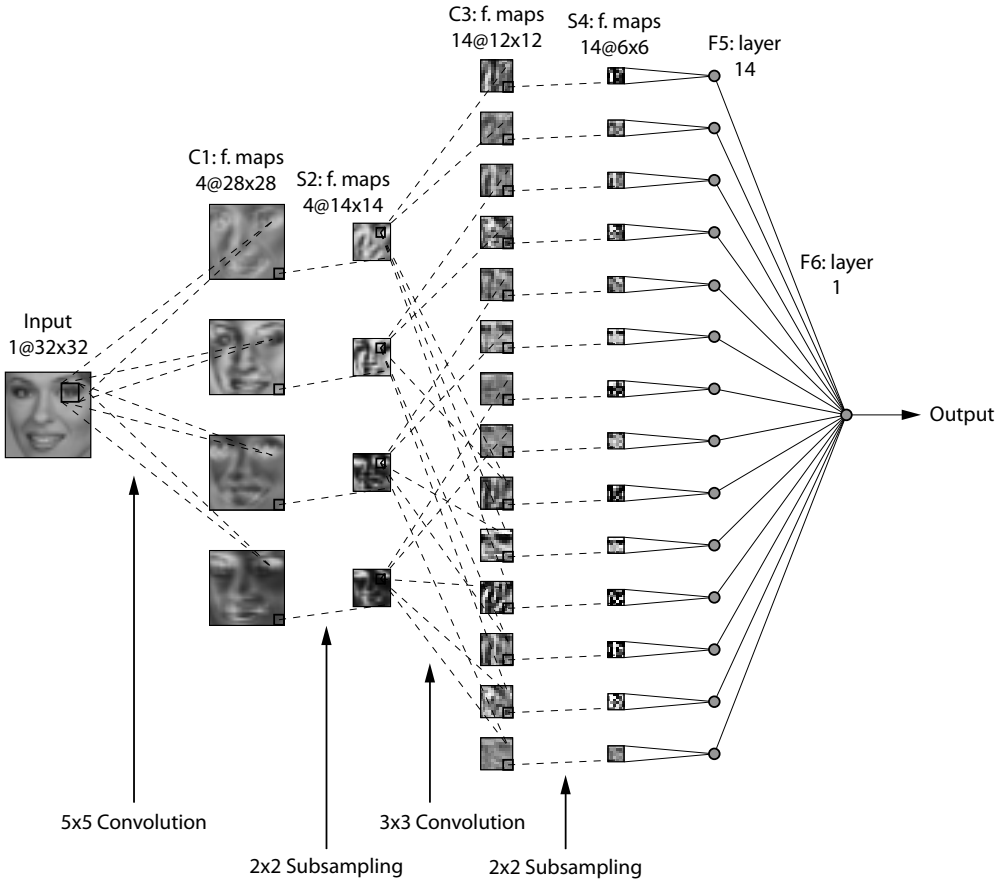


Figure 7.7: Architecture of the CNN used in the face detection system (based on Figure 1(a) in [5]); the feature maps of layer S2 and C3 are partially connected according to the connection matrix in Table 7.7.

		Feature maps in layer C3													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
Feature maps in layer S2	0	×	×							×	×	×			
	1			×	×					×			×	×	
	2					×	×				×		×		×
	3							×	×			×		×	×

Table 7.7: Connection matrix between the feature maps of the second and the third layer in the GarciaNet.

layer F5 is a fully connected layer formed of 14 neurons, where each of the 14 neurons is connected to all neurons of only one corresponding feature map of the previous layer S4. This layer can also be implemented as a convolutional layer of 14 feature maps using a kernel of size 6×6 and a 1-to-1 connection of the corresponding feature maps (as we did in our implementation). The last layer F6 of the GarciaNet consists of one single neuron, which is fully connected to all 14 neurons of the preceding layer F5. The output value of this neuron in the last layer (the network output) is used to classify the presented patterns, where a negative value is produced for a non-face and a positive value for a face. To gain a network output value between ±1 the tangens hyperbolicus was used as activation function (Subsection 2.7.3). The GarciaNet used in our face detection system consists of 6,455 neurons, 867 trainable parameters (weights and biases) and 116,445 connections between the neurons.

7.2.2 Network Training

The GarciaNet was trained using the RPROP algorithm, which is described in Subsection 2.5.2, in order to determine whether an image is a face or non-face pattern. As training database we used the face database from Son Lam Phung (Appendix A.3), which consists in total of 12,000 patterns of size 32×32 pixels. While the face patterns were manually cropped from Web images, the non-face patterns were randomly extracted from different scenery photos. The face database is split into a training set of 1,000 faces and 1,000 non-faces and a test set of 5,000 faces and 5,000 non-faces. To obtain more training patterns we exchanged the training and test set of the database. Additionally, we extracted 10% (1,000 patterns) of the training set for the validation process. As in the previous section, the GarciaNet was trained until the early stopping procedure (Subsection 2.5.5) reached its break condition. The RPROP training was terminated after 50 epochs without reduction of the MSE on the validation dataset. During the training the desired network responses for a face and a non-face pattern were set to +1 and -1, respectively. The smallest error on the validation set was achieved after 270 training epochs and the obtained weights in this epoch were used in the face detection process described in the following subsection.

Figure 7.9 shows how the error evolves on the training, validation, and test dataset over the training epochs. The obtained classification performance on the three different datasets are listed in Table 7.8, while Figure 7.8 presents the *receiver operating characteristic* (ROC) graph [113, 114] of the proposed network. The ROC graph is a way to illustrate the performance of a binary classifier, where the *false positive rate* is plotted on the X-axis and the *true*

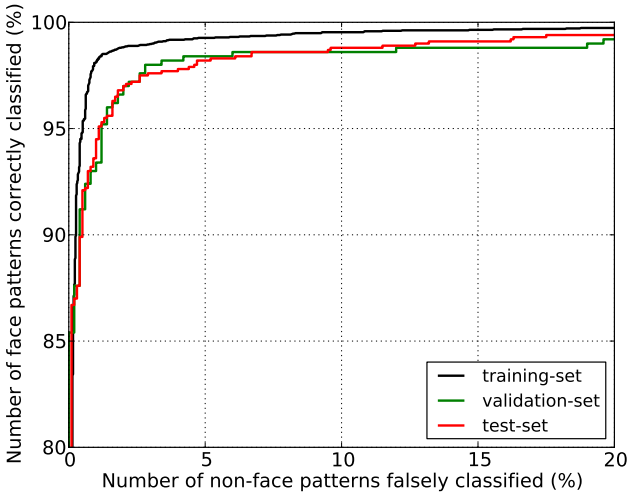


Figure 7.8: ROC curves of the trained GarciaNet applied to the training, validation, and test dataset.

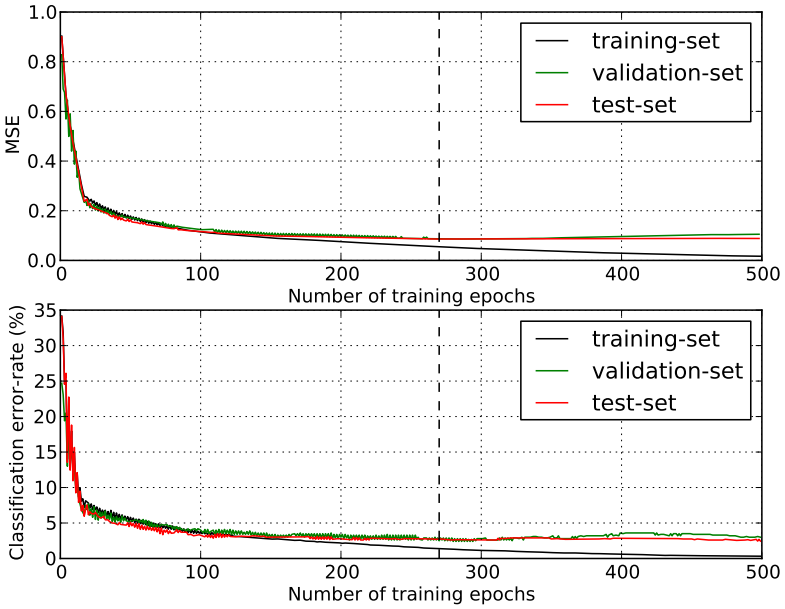


Figure 7.9: Development of the error on all three datasets during the face/non-face training of the GarciaNet using the face database from Son Lam Phung. The dashed line indicates the epoch with the best performance on the validation set.

Datasets	Classification rate					
	overall		faces		non-faces	
training set	**98.52%**	(8867/9000)	98.66%	(4440/4500)	98.38%	(4427/4500)
validation set	**97.50%**	(975/1000)	97.00%	(485/500)	98.00%	(490/500)
test set	**97.30%**	(1946/2000)	97.20%	(972/1000)	97.40%	(974/1000)

Table 7.8: Classification performance of the trained GarciaNet on the three different datasets.

positive rate on the Y-axis. In our case the false and the true positive rate are the number of *misclassified non-face patterns* and the number of *correct classified face patterns* (in %), respectively. The point $(0, 100)$ in the ROC graph denotes the perfect binary classifier that predicts all patterns of a dataset correctly. In this point the false positive rate is 0% (no misclassified non-face pattern) and the true positive rate is 100% (all face patterns are classified correctly). The points of the ROC curves in Figure 7.8 can be calculated according to Algorithm 8. In this algorithm the symbol P and N denote the number of face and non-face patterns in the dataset L (training, validation, or test set), respectively. The probabilistic estimate $f(i)$ that example i is positive can be obtained directly from the network output, because for every pattern that the CNN recognizes as a face it produces a value between 0 and 1. More theoretical details about ROC graphs and how to use them in research can be found in [113, 114].

7.2.3 Face Localization Procedure

Once the GarciaNet has been successfully trained it can be used to detect and locate human faces of arbitrary size in a grayscale image. The network acts like a filter receiving an image of size 32×32 pixels and produces an output value which indicates the presence or absence of a human face. In most image-based approaches [115–117] the neural network is applied to every pixel of the input image, because standard MLPs (Section 2.3) offer little or no invariance in position and scale. However, this is a very time-consuming process. In our approach using a CNN the entire image can be filtered at once by the network as illustrated in Figure 7.10. This results in an output image containing the single network responses which is approximately four times smaller than the input image. Passing the entire image through the network is possible through the successive convolution and subsampling operations inside the CNN and corresponds to the application of the network at every 4th pixel of the input image in both directions. The network responses in the obtained output image are then compared to a threshold T_1, and those responses that are larger than this threshold are considered as *face candidates*. Since the input and output

Algorithm 8 Efficient method for generating ROC points (from [113]).

Input:

- L, the set of test examples; $f(i)$, the probabilistic classifier's estimate that example i is positive; P and N, the number of positive and negative examples.

Output:

- R, a list of ROC points increasing by *fp rate* (false positive rate).

Require:

- $P > 0$ and $N > 0$

1: $L_{sorted} = L$ sorted decreasing by f scores
2: $FP = TP = 0$
3: $R = \langle\rangle$ /* Empty list */
4: $f_{prev} = -\infty$
5: $i = 1$
6: **while** $i \leq |L_{sorted}|$ **do**
7: **if** $f(i) \neq f_{prev}$ **then**
8: push $\left(\frac{FP}{N}, \frac{TP}{P}\right)$ onto R
9: $f_{prev} = f(i)$
10: **end if**
11: **if** $L_{sorted}[i]$ is a positive example **then**
12: TP += 1
13: **else** /* i is a negative example */
14: FP += 1
15: **end if**
16: i += 1
17: **end while**
18: push $\left(\frac{FP}{N}, \frac{TP}{P}\right)$ onto R /* This is (1,1) */
19: **return** R

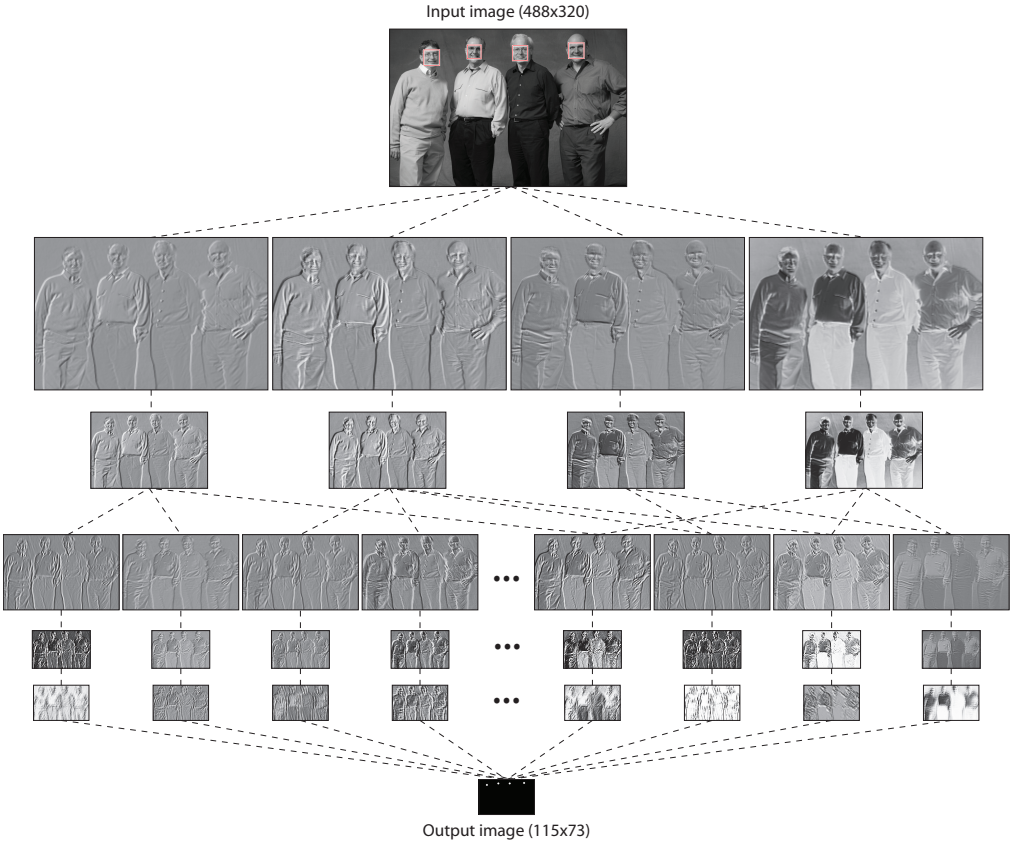


Figure 7.10: The images produced at each layer of the CNN during the processing of an input image (a snapshot of Bill Gates, Craig Mundie, Ray Ozzie, and Steve Ballmer taken from the web[3]) in the face detection system (based on Figure 2 in [5]). The corresponding output image is approximately four times smaller than the input image.

image are of different size the position of the face candidates in the output image must be mapped back to the input image.

The proposed network can only detect human faces which are approximately 32×32 wide. Thus, the input image is repeatedly subsampled by a factor of 1.2 as shown in Figure 7.11 and processed by the network as described above [3–5]. This results in a *pyramid* of input and corresponding output images. The location and size of the found face candidates in the single output images of this pyramid must then be mapped back to the original input image, in order to obtain faces of different sizes. This entire set of face candidates obtained in each scale space of the image pyramid is retained for further processing.

[3] `http://www.microsoft.com/presspass/press/2006/jun06/06-15CorpNewsPR.mspx`

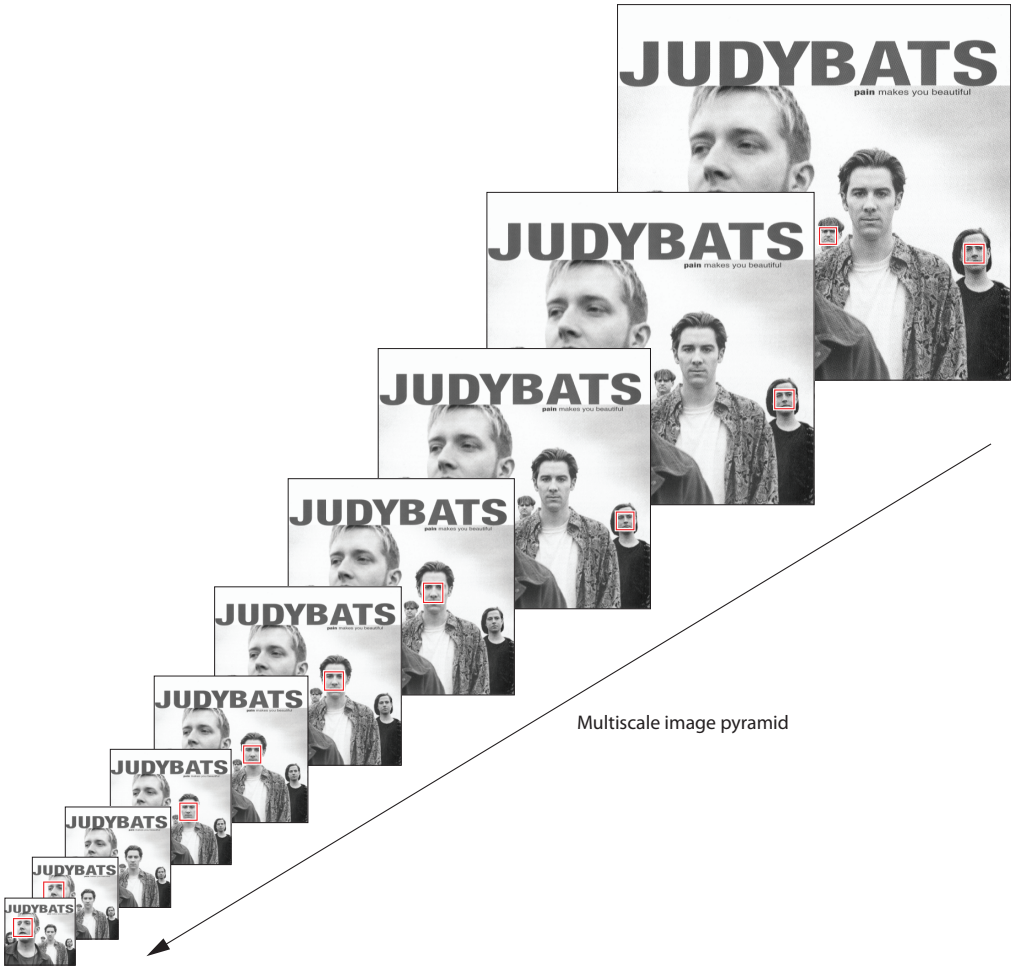


Figure 7.11: The multiscale image pyramid generated by the face detection system in order to detect faces of different sizes; the original input image stems from the test set of the MIT-CMU face database [115][4].

During the described detection process, some regions from the background often produce high network responses, and hence be misclassified as a face. Furthermore, overlapping detections usually occur around each true face in an image caused by the transformation invariance of the CNN. To overcome this problems, a certain number of *post-processing steps* based on the ideas presented in [3–7] are performed on the face candidates.

Human faces are usually rather symmetric. Therefore, each detected face candidate is mirrored along the vertical axis (Y-axis) and passed a second time through the network [6, 7]. The average of both network responses is then

[4] `http://vasc.ri.cmu.edu//idb/html/face/frontal_images/index.html`

Algorithm 9 Clustering algorithm to merge overlapping face detections together (from [3]).

1: Sort the face candidates in decreasing order of their network response o_i.
2: $n = 0$
3: **for** each not yet assigned face candidate i **do**
4: $\quad n = n + 1$
5: $\quad$ Assign face candidate i to a new cluster C_n with $(X_n, Y_n, W_n, H_n) = (x_i, y_i, w_i, h_i, o_i)$.
6: $\quad$ **for** each not yet assigned face candidate j **do**
7: $\quad\quad$ **if** center of face candidate j is within the rectangle which top left and bottom right corners are respectively $(X_n - W_n/4, Y_n - H_n/4)$ and $(X_n + W_n/4, Y_n + H_n/4)$ **then**
8: $\quad\quad\quad$ Assign face candidate j to cluster C_n and update the cluster parameters:
$$X_n = \sum_{k \in C_n}(o_k \cdot x_k) / \sum_{k \in C_n} o_k$$
$$Y_n = \sum_{k \in C_n}(o_k \cdot y_k) / \sum_{k \in C_n} o_k$$
$$W_n = \sum_{k \in C_n}(o_k \cdot w_k) / \sum_{k \in C_n} o_k$$
$$H_n = \sum_{k \in C_n}(o_k \cdot h_k) / \sum_{k \in C_n} o_k$$
$$O_n = \max_{k \in C_n} o_k$$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **end for**
12: $N = n$ /* N found face clusters */

compared to another threshold T_2. If it is less than the threshold the candidate is removed from the list of face candidates, otherwise it will be retained in the list and the average of both network responses is taken as the *final network response* for this face candidate.

In a second step the overlapping detections in the remaining list of face candidates are merged together into a cluster according to Algorithm 9 proposed by Delakis and Garcia in [3]. For each group of overlapping face candidates a *representative face* is computed whose center and size are taken as the centroid of all face candidates in the cluster. In this algorithm each face candidate i is represented by a vector $(x_i, y_i, w_i, h_i, o_i)$ where x_i and y_i are the coordinates of the face center, w_i and h_i are the width and height of the face, and o_i denotes the final network response for this face candidate. The representative faces of the N clusters found by this algorithm are then considered as the new face candidates and serve as a basis for further processing stages.

In the next step the remaining overlapping face candidates which weren't merged together in the previous step are eliminated. Each face candidate that overlaps in any way with another candidate with a higher network response is

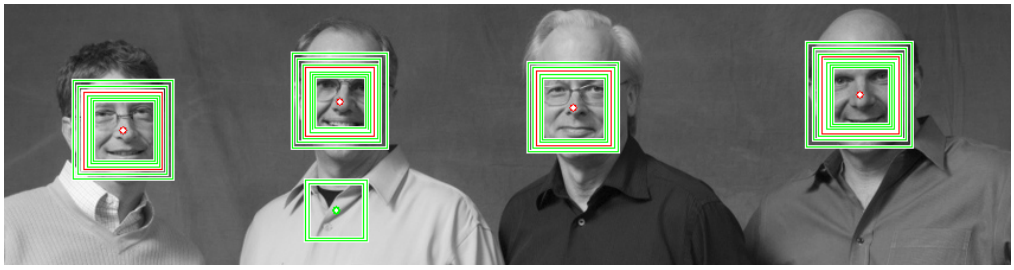
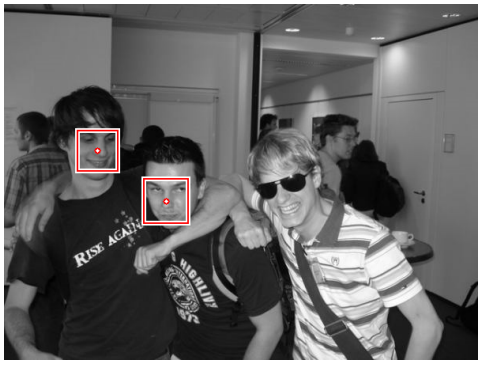


Figure 7.12: Illustration of the verification step in the last stage of the face localization procedure. While true faces will give high network responses in a certain number of consecutive scales, this is usually not the case for non-faces.

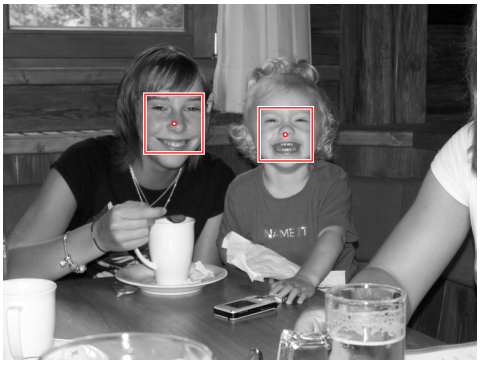
removed from the list.

The last stage of the face localization procedure performs an additional *verification step* on each face candidate in order to reduce the number of false detections. Each face candidate is tested at six scales in the original input image, ranging from 0.7 to 1.3 of the detected size [3–5]. At each scale the presence of a face is evaluated and compared to a third threshold T_3. As noted by Garcia and Delakis in [3–5] and illustrated by Figure 7.12, true faces will give high network responses in a certain number of consecutive scales, while non-faces don't. Therefore, we count the number of network responses which exceed the threshold T_3. If their quantity is less than a fourth threshold T_4, the candidate is removed from the list of face candidates. Furthermore, the corresponding sizes of this positive network responses are averaged to obtain the final size of the face. The remaining elements in the face candidate list are then considered as the final detection result.

The threshold values T_1, T_2, T_3, and T_4 used in our tests which produced good results were set to 0.99, 0.95, 0.90, and 4, respectively. Figure 7.13 shows some face detection examples obtained by our face detection system where each detected face is marked with a red rectangle.



(a) A snapshot of Adrian, Daniel, and Fritz.



(b) A snapshot of Sandra and Joana.

(c) An image from the test set of the MIT-CMU face database [115].

Figure 7.13: Examples of detecting multiple human faces in an image using our face detection system based on a CNN.

Chapter 8

Conclusion and Future Work

The only source of knowledge is experience.
– Albert Einstein

This chapter summarizes the most important experiences we gained during the work on this thesis. Furthermore, it outlines some possible enhancements of our library and gives an outlook on future work that should be completed in order to answer all open questions that arose during our research.

8.1 Conclusion

In the course of our thesis we developed a highly optimized, flexible, and easy-to-use library for CNNs. This library can be used to build arbitrary CNNs and hides their complexity from researchers. We discovered that such an implementation is quite difficult, but the applied techniques as described in Chapter 5 helped to keep the complexity within reasonable boundaries. Additionally, this thesis covers the CNNs' mathematical model, describing both, the forward- and the backpropagation process. It also contains a short overview about CUDA, since this programming language has been used to implement the GPU version of our library. We showed how CNNs scale on modern homogeneous multicore processors. Furthermore, we illustrated the potential of our library on two real-world applications, namely character recognition and face detection (Chapter 7).

Our implementation clearly demonstrates the superiority of GPUs over traditional CPUs for this kind of machine learning algorithms showing serious speedups (Chapter 6). However, we discovered that the superiority of the GPU depends on the network topology (composition of the layers) and its size (number of feature maps and neurons inside the layers). Small networks are not able to fully utilize a today's GPU. Furthermore, the used programming language to realize the GPU implementation, namely CUDA, can be considered as both, fast and easy to use. At this point of time it is definitely the best choice to program NVIDIA GPUs.

When implementing CNNs we saw, that the complex structure of them makes debugging techniques, like the ones described in Subsection 5.3.3, indispensable to avoid bugs. However, we realized that CNNs are worth the immense implementation effort since they deliver good performance on two-dimensional pattern recognition and classification problems. The implementation enhancements introduced in Section 5.3 turned out to work quite well and make the programmers' job much easier.

When trying to improve the accuracy of our networks we made three major experiences: First, the dataset enlargement by distortions (Appendix B) as well as the data preprocessing by normalization (Subsection 2.5.4) did significantly rise the performance in our test scenarios. Second, huge networks using very sparse connections between the feature maps of consecutive layers are considerably slower than smaller ones with a higher connection rate, but seem to achieve higher classification rates (as we have seen so far). Finally, we gained the best results when using the tangens hyperbolicus (tanh) as activation function in combination with a training signal composed of ± 1. This composition seems to be clearly superior to all other activation function/training signal combinations we have tested.

Besides this thesis, we wrote a paper about our work that has been accepted for the *Scalable Data Intensive Applications Special Session* of the PDP 2010[1] in Pisa, Italy. This paper with the title "Performance and Scalability of GPU-based Convolutional Neural Networks" will appear in the *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*.

8.2 Future Work

While working on our thesis we recognized that this area of research contains a lot of interesting questions which could not be addressed in the scope of our work, mainly because of our limited amount of time. Some of those topics as well as a couple of proposals to expand our library will be introduced in the following paragraphs.

As mentioned in Section 6.1 we recognized only a very small difference between our double and single precision floating-point implementation in terms of execution speed on the CPU although its peak performance on the latter one is twice as high. Our statement, that the reason for this is the activation function (e.g. tanh), should be analyzed in detail to verify it. Furthermore, it would be

[1] The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa, Italy, 17.–19. February, 2010; `http://www.pdp2010.org`

interesting to see if this phenomenon does also arise on other compiler/CPU combinations.

Something that is still missing in our library is a double precision implementation for GPUs. Although the used precision has only a minor effect on the network's classification rate, it would be very interesting to see how GPUs perform using 64-bit floating-point numbers. Another improvement of our library could be the implementation of a *manufacturer independent* GPU version, using for example OpenCL [78, 79] or DirectCompute [80–82], to let an even bigger clientele exploit the performance of their GPUs for neural networks.

Since nowadays *green IT* is of big interest a comparison of our CPU and GPU variant in terms of energy consumption could be interesting. As the TDP values of the processors used for our benchmarks (shown in Table 6.1 on page 86) already imply we measured a higher energy use when running a network on the GPU as when running it on the CPU. Our brief measurements showed an increase in power consumption of approximately 15% for the whole system, which should be compensated by the shorter training and execution time of the GPU variant. However, to get accurate and reliable results, more tests, measuring the power consumption over several entire training runs, would be needed.

As already mentioned, the moment when to stop the training of the network is quite difficult to determine. In this thesis only the early stopping method (Subsection 2.5.5) was implemented and tested, because it is simple and often applied in literature. However, as we have seen in our tests, this method is far from optimal. Observing the classification rates on the test set over the entire training we recognized, that it is quite uncommon for the early stopping method to detect the ideal point to terminate the training. Often the best classification rate on the test dataset reached during training is 10% to 15% inferior to the one chosen by the early stopping algorithm. Finding a better stopping criterion holds much potential to enhance the overall results. Therefore, a future research should test alternative algorithms like for example one based on *cross-validation* [27].

In this work we did not put much effort in the human face detection system (Section 7.2), since we were mainly interested in building a library for CNNs. Therefore, this system, primarily the post-processing (fine-search), should be enhanced and tested before applying it to a real-world application. Furthermore, in the actual state, the face detection system is only implemented on the CPU, a GPU variant is still missing and should be implemented in a future step.

During the work on our thesis we focused on two applications (handwritten digit recognition, face detection) and a small number of previously published networks. Therefore, a scope of future research could be the development and

evaluation of other network structures. In the course of it, the capabilities of such networks could also be tested on other problems, such as e.g. data matrix code or license plate detection. Additionally, one could enlarge our library by some other types of layers (e.g. layers using RBF units [22] or shunting inhibitory neurons [118]), additional training algorithms (e.g. SuperSAB [119] or Quickprop [120]) and other error functions. Another core area in future enhancements of our library could be the improvement of the usability, e.g. by deploying a GUI.

Appendix A

Training Databases

There exist many different databases to evaluate methods and applications for pattern recognition and classification. The following sections describe the databases used in this thesis, starting with the MNIST Handwritten Digit Database in Section A.1, followed by the CBCL Face Database #1 in Section A.2, and finally the Face Database from Son Lam Phung in Section A.3.

A.1 MNIST Handwritten Digit Database

The MNIST (Modified NIST) database [106] from Yann LeCun[1] (Computational and Biological Learning Lab, Courant Institute, New York University) and Corinna Cortes[2] (Google Labs, New York) is a subset of a larger database available from the NIST[3] (National Institute of Standards). It is one of the most widely known pattern classifier benchmarks and consists of size-normalized, centered images of handwritten digits, each of size 28×28 pixels with 256 gray levels. The database is divided into two datasets, a training set with 60,000 patterns and a test set with 10,000 patterns.

The procedure how the MNIST database was constructed from the NIST database is described as follows in [2]: "The original black and white (bilevel) images were size normalized to fit in a 20×20 pixel box while preserving their aspect ratio. The resulting images contain gray levels as result of the anti-aliasing (image interpolation) technique used by the normalization algorithm. [...] the images were centered in a 28×28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28×28 field."

The patterns and labels $(0, 1, \ldots, 9)$ of each dataset (training and test set) are located in separate files, stored in a simple file format designed for storing vectors and multidimensional matrices, called IDX. The database and a detailed

[1] `http://yann.lecun.com`
[2] `http://homepage.mac.com/corinnacortes`
[3] `http://www.nist.gov`

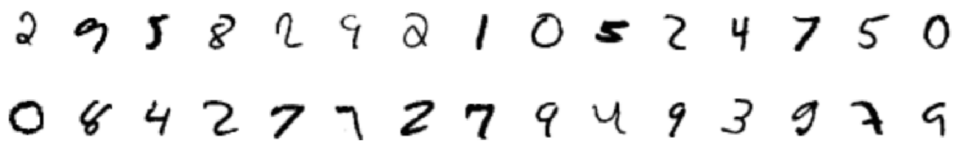


Figure A.1: Some examples from the MNIST Digit Database.

description of the IDX file format are available online[4] and some examples randomly picked from the test set are shown in Figure A.1. Most of the tests and benchmarks in this work were performed with the MNIST database.

As the MNIST database is one of the most widely used benchmarks for pattern classifiers, there are many results based on different classification approaches. Table A.1 gives an overview of some error rates and the methods on which they are based. This overview was provided by Daniel Keysers[5] in a study about the comparison and combination of state-of-the-art techniques for handwritten character recognition [41]. Another, little bit more detailed analysis of errors made by state-of-the-art classifiers for handwritten digits can be found in [121]. As one can see from the overview in Table A.1, CNNs (marked with *) are very suitable for this kind of classification task.

A.2 MIT CBCL Face Database #1

The CBCL Face Database #1 [140] is provided by the Center for Biological and Computational Learning (CBCL)[6] at MIT (Massachusetts Institute of Technology) and consists of face and non-face images, each of size 19×19 pixels with 256 gray levels. The training set includes 2,429 faces and 4,548 non-faces, while the test set consists of 472 faces and 23,573 non-faces.

The training and test set are provided as single images in PGM (Portable Gray Map) file format as well as in a single human-readable file where each line of the file represents a histogram equalized and normalized version of the images (so that all pixel values are between 0 and 1), followed by a 1 for a face or a -1 for a non-face. In our work we used the latter representation of the datasets, because it was much easier to handle than the thousand of single files.

Figure A.2 shows some face and non-face examples from the CBCL Face Database #1.

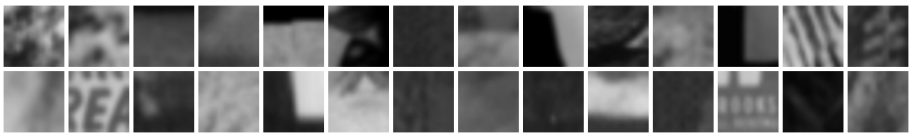
[4] `http://yann.lecun.com/exdb/mnist`
[5] `http://www.keysers.net/daniel`
[6] `http://cbcl.mit.edu`

Method	Error rate (%)	Reference
Human Performance	0.20	[122]
Euclidean Nearest Neighbor	3.50	[41]
Decision Trees and Sub-Windows	2.63	[123]
Euclidean 3-NN (deslant)	2.40	[2]
Elastic Matching	2.10	[124]
One-sided Tangent Distance	1.90	[125]
CNN LeNet1	*1.70	[126]
Product of Experts	1.70	[127]
Hyperplanes and Support Vector Machine	1.50	[128]
Support Vector Machine	1.40	[129]
Our SimardNet[a]	***1.21**	—
CNN LeNet4	*1.10	[126]
Tangent Distance	1.10	[122]
Our LeNet5[b]	***1.09**	—
Two-sided Tangent Distance (virtual data)	1.00	[125]
Local Learning	0.99	[130]
Our JahrerNet[c]	***0.85**	—
CNN LeNet5 (distortions)	*0.82	[2]
Virtual Support Vector Machine	0.80	[131]
Bio-inspired Features and SVM	0.72	[132]
Boosted LeNet4 (distortions)	*0.70	[2]
Virtual Support Vector Machine (jitter)	0.68	[133]
Our SimardNet (with distortions)[d]	***0.65**	—
Shape Context Matching	0.63	[134]
Support Vector Machine	0.60	[135]
Bio-inspired Features (deslant)	0.59	[136]
Cascaded Shape Context	0.58	[137]
Virtual SVM (deslant, jitter, shift)	0.56	[133]
Shape Context Matching	0.54	[137]
Deformation Model (IDM)	0.54	[138]
Support Vector Machine (preprocessing)	0.42	[139]
CNN from Simard et al. (virtual data)	*0.40	[1]

[a] see Subsection 7.1.2
[b] see Subsection 7.1.1
[c] see Subsection 7.1.4
[d] see Subsection 7.1.3

Table A.1: Error rates (in %) of different classifiers for the MNIST Database (based on Table 1 in [41]).

(a) face patterns



(b) non-face patterns

Figure A.2: Some examples from the CBCL Face Database #1.

(a) face patterns

(b) non-face patterns

Figure A.3: Some examples from Son Lam Phungs Face Database (from [49]).

A.3 Face Database from Son Lam Phung

The face database from Son Lam Phung[7] comes with [49] and is taken from the face and skin detection database used in [141]. It is split into a training set of 1,000 faces and 1,000 non-faces and a test set of 5,000 faces and 5,000 non-faces.

The images and labels of both datasets are stored as a three- and an one-dimensional Matlab array inside a MAT-file[8]. Those images are of size 32×32 pixels with normalized grayscale values between 0 and 1. The labels are single scalar values indicating whether an image contains a face or not (1 for a face and -1 for a non-face).

Some examples of the face database provided by Son Lam Phung are shown in Figure A.3. In comparison to the images in the CBCL Face Database #1 (A.2), more details of the face and non-face patterns are visible.

[7] `http://www.elec.uow.edu.au/staff/sphung`

[8] Detailed information about the MAT-file format can be found in the "MAT-File Format" documentation coming with the Matlab software (`matfile_format.pdf`).

Appendix B

Enlargement of the Training Dataset

One of the major drawbacks of neural networks is that they need a huge training dataset to learn a specific problem. As stated by LeCun et al. in [2], more training data usually improve the accuracy of a neural network. Gathering such training samples is very costly since the patterns have to be selected and classified manually, which limits the size of a training dataset. Therefore, a better generalization and classification performance of a neural network can be obtained by expanding the training dataset by a new set of *virtual training images*. This virtual images are constructed from a given dataset by applying a two-dimensional *displacement field* to each of the images in the given dataset. Since the images are normally generated on-the-fly in each training epoch no data have to be stored, which therefore simplifies data handling.

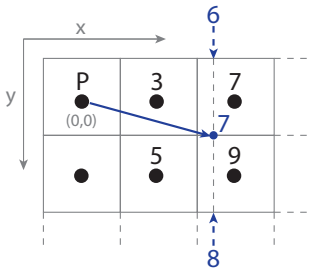
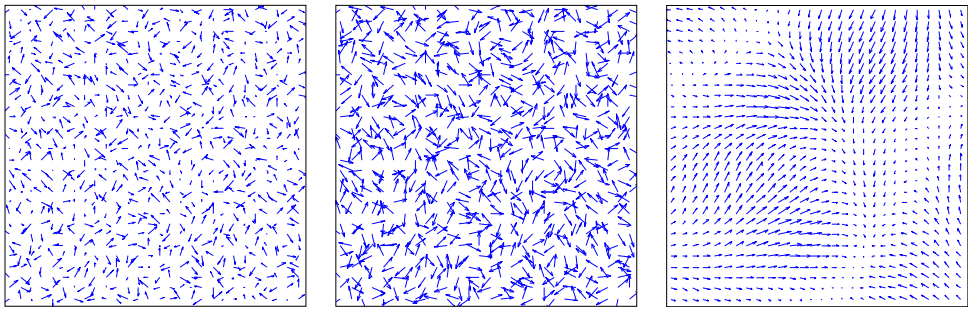


Figure B.1: Illustration of the bilinear interpolation, used to compute the new value of pixel P at location $(0, 0)$ and a given displacement $\Delta x = 1.75$ and $\Delta y = 0.5$ (based on Figure 1 in [1]).

The displacement field is of the same size as the images in the given dataset, where each cell describes a vector pointing to the new source value. For example a vector with $\Delta\mathbf{x} = -1$ and $\Delta\mathbf{y} = 0$ in each cell of the displacement field means that the hole image is shifted one position to the right, because every pixel in the virtual image is initialized with the value of its left neighbor in the original image. Since the vectors in the displacement field may not point exactly to a pixel in the original image (see Figure B.1), *interpolation* is necessary. The

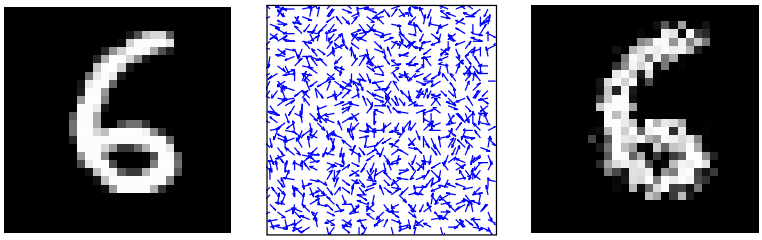


(a) after initialization (b) after normalization (c) after Gaussian ($\sigma = 4$)

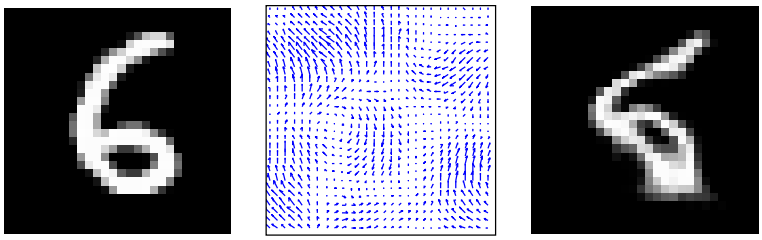
Figure B.2: Illustration of the single steps to generate a displacement field for elastic deformations.

interpolation used in our implementation was the *bilinear* one [142, pages 412–414], although any other method can be used (e.g. nearest-neighbor, bicubic, and spline interpolation). The bilinear interpolation is rather simple to implement and works very well to generate distorted images of the handwritten digits in the MNIST database (Appendix A.1). Figure B.1 illustrates how to compute the new value of pixel P at the location $(0, 0)$ with a displacement vector given by $\Delta x = 1.75$ and $\Delta y = 0.5$. This vector does not point exactly to a pixel in the original image, so the new value has to be computed by a horizontal interpolation (x-direction) which is followed by a vertical interpolation (y-direction). The two values of the horizontal interpolation are: $3 + 0.75 \cdot (7 - 3) = 6$ and $5 + 0.75 \cdot (9 - 5) = 8$. To obtain the final value at the position $(1.75, 0.5)$ these two values are interpolated again in the vertical direction: $6 + 0.5 \cdot (8 - 6) = 7$. This yields the new value for the pixel P in the warped image.

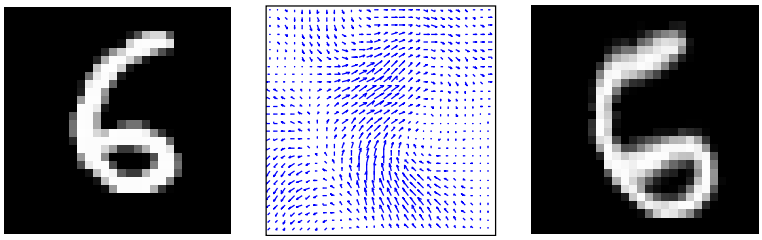
Applying an *affine* displacement field to the images will generate some simple distortions such as translated, rotated or scaled images. Much more powerful in the context of optical character recognition (OCR) are *elastic deformations* simulating *uncontrolled oscillations* of the hand muscles, first presented by Simard et al. in [1]. The displacement fields performing an elastic deformation were created by first initializing $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ with (uniform distributed) random numbers between -1 and 1 (Figure B.2(a)). The fields $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ are then normalized to a norm of 1, which deliver displacement vectors all of length 1 but with random directions (Figure B.2(b)). In the next step this fields are convolved with a *Gaussian low-pass filter* (Figure B.2(c)), where the standard deviation σ of the Gaussian filter defines the *elasticity* of the deformation. Small values of the standard deviation σ will generate completely random fields while larger values will generate fields that look like elastic deformations (see also Figure B.3). Fi-



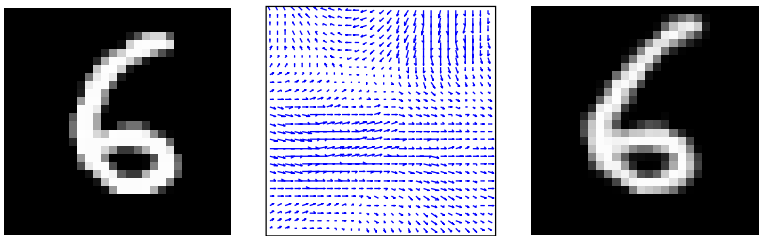
(a) $\sigma = 0.1$ and $\alpha = 1$



(b) $\sigma = 3$ and $\alpha = 30$



(c) $\sigma = 4$ and $\alpha = 34$



(d) $\sigma = 6$ and $\alpha = 22$

Figure B.3: Some examples of a distorted digit of the MNIST database (Appendix A.1) using different values for the elasticity factor σ and the intensity factor α. The applied displacement fields are displayed between the original and warped image.

nally, the fields $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ are multiplied by a scaling factor α which defines the *intensity* of the deformation. Some examples of a distorted digit together with their corresponding displacement fields are shown in Figure B.3.

Appendix C

CUDA Bug #569295

One of the biggest problems we noticed during the work on this thesis was a problem in the CUDA environment, which later exposed to be a bug in CUDA. It took some time to locate the problem and extract a simple sample where the bug could be reproduced, so that it could be reported to the NVIDIA development team[1].

The problem occurred in the CUDA kernel named gemv_kernel provided in Listing 1. This kernel was responsible to compute the matrix-vector product of a matrix src1 and a vector src2 accumulated by a vector src3 using the shared memory.

When invoking this kernel inside a loop, using the following statement, the kernel-call failed with a "unspecified launch failure" after a random number of iterations.

```
gemv_kernel<128><<<rowsSrc1, 128>>>
    (src1, strideSrc1, rowsSrc1, colsSrc1, src2, src3, dst);
```

The irregular failures occurred in CUDA 2.2 as well as in CUDA 2.3. An interesting behavior of this problem was that it only occurred on the low-end and midrange models of NVIDIA's GPUs (on our Quadro FX360M and Geforce 8600M GT). Other models, like the GeForce 8800 Ultra, GeForce 9800 GTX+, and GeForce GTX 275, seem not to be affected by this bug.

We also found a workaround to redress this problem on our affected (mobile) devices: adding the following statement on top of the kernel function (see also Listing 1, line 7) seems to suppress the bug.

```
if (blockIdx.x >= rows) return;
```

Although this statement makes no sense, because the thread identifier blockIdx.x should never be greater or equal to the number of rows (since we launch the kernel with a block size equal to the number of rows), it changes the behavior.

[1] A nice article how to report bugs (effectively) can be found at: `http://www.chiark.greenend.org.uk/~sgtatham/bugs.html`

Listing 1 CUDA Bug #569295.

```
template<size_t blockSize>
__global__
void gemv_kernel(float const * src1, size_t const strideSrc1,
                 size_t const rows, size_t const cols,
                 float const * src2, float const * src3, float * dst)
{
    //if (blockIdx.x >= rows) return;

    __shared__ float sdata[blockSize];

    float  const * const row = src1 + __umul24(blockIdx.x, strideSrc1);
    size_t const   const tid = threadIdx.x;

    // Reduce multiple elements per thread
    sdata[tid] = 0;
    for (size_t i = tid; i < cols; i += blockSize)
        sdata[tid] += row[i] * src2[i];
    __syncthreads();

    // Do reduction in shared memory
    if (blockSize >= 512) {
        if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
    }
    if (blockSize >= 256) {
        if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
    }
    if (blockSize >= 128) {
        if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads();
    }

    if (tid < 32) {
        if (blockSize >=  64) { sdata[tid] += sdata[tid + 32]; }
        if (blockSize >=  32) { sdata[tid] += sdata[tid + 16]; }
        if (blockSize >=  16) { sdata[tid] += sdata[tid +  8]; }
        if (blockSize >=   8) { sdata[tid] += sdata[tid +  4]; }
        if (blockSize >=   4) { sdata[tid] += sdata[tid +  2]; }
        if (blockSize >=   2) { sdata[tid] += sdata[tid +  1]; }
    }

    // Write result to global memory
    if (tid == 0) {
        dst[blockIdx.x] = sdata[0] + src3[blockIdx.x];
    }
}
```

We reported this bug to the NVIDIA development team on the 25 June 2009 and Kurt Wall from NVIDIA was able to reproduce it on a Quadro FX370 (using CUDA 2.3 Beta). More details about this bug and the current status can be found on the NVIDIA Developer Zone[2] under the Current Bugs section and the Bug ID #569295. Furthermore, some information can be found in the NVIDIA Forums under `http://forums.nvidia.com/index.php?showtopic=99845`.

[2] `https://nvdeveloper.nvidia.com` (restricted for registered developers)

Appendix D

Listings

Listing 2 Vector addition CUDA Runtime API host code.

```
void addv(float const * h_A, float const * h_B, float * h_C, int const len)
{
    int const size = len * sizeof(float);

    // Allocate memory on the device
    float * d_A, * d_B, * d_C;
    cudaMalloc((void**) &d_A, size);
    cudaMalloc((void**) &d_B, size);
    cudaMalloc((void**) &d_C, size);

    // Copy data from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Execute device function
    dim3 const dimBlock(128);
    dim3 const dimGrid((len + dimBlock.x - 1) / dimBlock.x);
    addv_kernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, len);

    // Copy data from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Listing 3 Vector addition CUDA Driver API host code.

```
void addv(float const * h_A, float const * h_B, float * h_C, int const len)
{
    int const size = len * sizeof(float);

    // Initialize
    cuInit(0);

    // Get handle for device 0
    CUdevice cuDevice = 0;
    cuDeviceGet(&cuDevice, 0);

    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, 0, cuDevice);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, "addv_kernel.cubin");

    // Get function handle from module
    CUfunction cuAddvFnc;
    cuModuleGetFunction(&cuAddvFnc, cuModule, "addv_kernel");

    // Allocate memory on the device
    CUdeviceptr d_A, d_B, d_C;
    cuMemAlloc(&d_A, size);
    cuMemAlloc(&d_B, size);
    cuMemAlloc(&d_C, size);

    // Copy data from host to device
    cuMemcpyHtoD(d_A, h_A, size);
    cuMemcpyHtoD(d_B, h_B, size);

    // Setup execution parameters
    void * ptr = (void *)(size_t) d_A;
    cuParamSetv(cuAddvFnc, 0, &ptr, sizeof(ptr));
    ptr = (void *)(size_t) d_B;
    cuParamSetv(cuAddvFnc, 4, &ptr, sizeof(ptr));
    ptr = (void *)(size_t) d_C;
    cuParamSetv(cuAddvFnc, 8, &ptr, sizeof(ptr));
    cuParamSeti(cuAddvFnc, 12, len);
    cuParamSetSize(cuAddvFnc, 16);

    // Execute device function
    int const threadsPerBlock = 128;
    int const blocksPerGrid = (len + threadsPerBlock - 1) / threadsPerBlock;
    cuFuncSetBlockShape(cuAddvFnc, threadsPerBlock, 1, 1);
    cuLaunchGrid(cuAddvFnc, blocksPerGrid, 1);

    // Copy data from device to host
    cuMemcpyDtoH(h_C, d_C, size);

    // Free device memory
    cuMemFree(d_A);
    cuMemFree(d_B);
    cuMemFree(d_C);
}
```

Listing 4 Vector addition CUDA device code.

```
__global__
void addv_kernel(float const * A, float const * B, float * C, int const len)
{
    // Calculate element index
    int const i = blockIdx.x * blockDim.x + threadIdx.x;

    // Check vector boundaries
    if (i < len)
    {
        // Perform vector addition
        C[i] = A[i] + B[i];
    }
}
```

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

[1] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *Proceedings of the 7th International Conference on Document Analysis and Recognition (ICDAR 2003)*, pages 958–962, Edinburgh, Scotland, August 2003.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[3] M. Delakis and C. Garcia. Robust Face Detection Based on Convolutional Neural Networks. In *Proceedings of the 2nd Hellenic Conference on Artificial Intelligence (SETN 2002)*, pages 367–378, Thessaloniki, Greece, April 2002.

[4] C. Garcia and M. Delakis. A Neural Architecture for Fast and Robust Face Detection. In *Proceedings of the 16th International Conference on Pattern Recognition (ICPR 2002)*, volume 2, pages 44–47, Quebec City, Canada, August 2002.

[5] C. Garcia and M. Delakis. Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(11):1408–1423, November 2004.

[6] F. H. C. Tivive and A. Bouzerdoum. A Fast Neural-Based Eye Detection System. In *Proceedings of the International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS 2005)*, pages 641–644, Hong Kong, December 2005.

[7] F. H. C. Tivive and A. Bouzerdoum. A hierarchical learning network for face detection with in-plane rotation. *Neurocomputing*, 71(16–18):3253–3263, June 2008.

[8] J. C. L. Lam and M. Eizenman. Convolutional Neural Networks for Eye Detection in Remote Gaze Estimation Systems. In *Proceedings of the*

International MultiConference of Engineers and Computer Scientists 2008 (IMECS 2008), volume 1, pages 601–606, Hong Kong, March 2008.

[9] Z. Zhao, S. Yang, and X. Ma. Chinese License Plate Recognition Using a Convolutional Neural Network. In *Proceedings of the Pacific-Asia Workshop on Computational Intelligence and Industrial Application (PACIIA 2008)*, volume 1, pages 27–30, Wuhan, China, December 2008.

[10] R. Collobert and J. Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, volume 307, pages 160–167, Helsinki, Finland, July 2008.

[11] NVIDIA Corporation. NVIDIA CUDA – Programming Guide (ver. 2.3.1). `http://www.nvidia.com/object/cuda_home.html`, August 2009.

[12] D. Luebke and G. Humphreys. How GPUs Work. *IEEE Computer*, 40(2):96–100, February 2007.

[13] I. Durdanovic, E. Cosatto, and H. P. Graf. *Large-Scale Kernel Machines*, chapter *Large-Scale Parallel SVM Implementation*, pages 105–138. Neural Information Processing Series. MIT Press, Cambridge, MA., September 2007.

[14] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, volume 307, pages 104–111, Helsinki, Finland, July 2008.

[15] Intel Corporation. Intel Integrated Performance Primitives (IPP). `http://software.intel.com/en-us/intel-ipp`, August 2009.

[16] Intel Corporation. Intel Math Kernel Library (MKL). `http://software.intel.com/en-us/intel-mkl`, August 2009.

[17] NVIDIA Corporation. NVIDIA CUBLAS Library. `http://www.nvidia.com/object/cuda_home.html`, August 2009.

[18] S. Sonnenburg, M. L. Braun, C. S. Ong, S. Bengio, L. Bottou, G. Holmes, Y. LeCun, K.-R. Müller, F. Pereira, C. E. Rasmussen, G. Rätsch, B. Schölkopf, A. Smola, P. Vincent, J. Weston, and R. Williamson. The Need for Open Source Software in Machine Learning. *The Journal of Machine Learning Research*, 8:1532–4435, December 2007. Available at: `http://jmlr.csail.mit.edu/papers/volume8/sonnenburg07a/sonnenburg07a.pdf`.

[19] D. Strigl and K. Kofler. CNNLIB: A library for fast convolutional neural network training and classification. http://cnnlib.sourceforge.net, October 2009.

[20] The Editors of Scientific American. *Scientific American Book of the Brain.* The Lyons Press, November 1999.

[21] A. Abraham. *Handbook for Measurement Systems Design*, chapter *Artificial Neural Networks*, pages 901–908. John Wiley and Sons Ltd., London, 2005. ISBN 0-470-02143-8, Available at: http://www.softcomputing.net/ann_chapter.pdf.

[22] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, New York, 1995.

[23] R. Rojas. *Neural Networks – A Systematic Introduction.* Springer, July 1996. Available at: http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/1996/NeuralNetworks/neuron.pdf.

[24] F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65:386–408, November 1958.

[25] D. Mandic and J. Chambers. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability.* Wiley, August 2001.

[26] A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks.* PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, 2008. http://www6.in.tum.de/pub/Main/Publications/graves_phd_2008.pdf.

[27] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd Edition).* Springer Series in Statistics. Springer, December 2008.

[28] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification, 2nd Edition.* Wiley, November 2000.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, volume 1, pages 318–362. MIT Press, 1986.

[30] Z. Tang, X. G. Wang, H. Tamura, and M. Ishii. An Algorithm of Supervised Learning for Multilayer Neural Networks. *Neural Computation*, 15(5):1125–1142, May 2003.

[31] X. G. Wang, Z. Tang, H. Tamura, M. Ishii, and W. D. Sun. An improved backpropagation algorithm to avoid the local minima problem. *Neurocomputing*, 56(1):455–460, 2004.

[32] S. Chai and Y. Zhou. A Study on How to Help Back-propagation Escape Local Minimum. In *Proceedings of the Third International Conference on Natural Computation (ICNC 2007)*, pages 64–68, Haikou, Hainan, China, August 2007.

[33] A. Atakulreka and D. Sutivong. Avoiding Local Minima in Feedforward Neural Networks by Simultaneous Learning. In *Proceedings of the 20th Australian Joint Conference on Artificial Intelligence*, pages 100–109, Gold Coast, Australia, December 2007.

[34] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient BackProp. In G. B. Orr and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 9–50. Springer, 1998.

[35] D. C. Plaut, S. J. Nowlan, and G. E. Hinton. Experiments on Learning by Back Propagation. Technical report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, June 1986.

[36] M. Riedmiller and H. Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. In *Proceedings of the International Conference on Neural Networks*, pages 586–591, San Francisco, California, 1993.

[37] C. Igel and M. Hüsken. Improving the Rprop Learning Algorithm. In *Proceedings of the Second International Symposium on Neural Computation (NC 2000)*, pages 115–121, Berlin, Germany, May 2000.

[38] K. L. Priddy and P. E. Keller. *Artificial Neural Networks: An Introduction*. SPIE Press Books, August 2005.

[39] S. Raudys. *Statistical and Neural Classifiers: An Integrated Approach to Design*. Advances in Pattern Recognition. Springer, January 2001.

[40] N. M. McCord and W .T. Illingworth. *A Practical Guide to Neural Nets*. Addison Wesley, New York, 1991.

[41] D. Keysers. Comparison and Combination of State-of-the-art Techniques for Handwritten Character Recognition: Topping the MNIST Benchmark. Technical report, IUPR Research Group, DFKI and Technical University of Kaiserslautern, May 2006.

[42] J. S. Bridle. On the Probabilistic Interpretation of Neural Network Classifiers and Discriminative Training Criteria. In F. Souli and J. Hrault, editors, *Neurocomputing: Algorithms, Architectures and Applications*, pages 227–236. Springer, 1990.

[43] D. M. Kline and V. L. Berardi. Revisiting squared-error and cross-entropy functions for training neural network classifiers. *Neural Computing and Applications*, 14(4):310–318, July 2005.

[44] N. H. F. Beebe. Accurate Hyperbolic Tangent Computation. Technical report, Center for Scientific Computing, Department of Mathematics, University of Utah, Salt Lake City, Utah, USA, April 1993. Available at: `http://www.math.utah.edu/~beebe/software/ieee/tanh.pdf`.

[45] Y. LeCun. Generalization and Network Design Strategies. In R. Pfeifer, Z. Schreter, F. Fogelman-Soulie, and L. Steels, editors, *Connectionism in Perspective*, North-Holland, Amsterdam, 1989. Elsevier Science Inc.

[46] Y. LeCun, 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 D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS 1989)*, pages 396–404, Denver, Colorado, USA, 1990. Morgan Kaufman.

[47] D. H. Hubel and T. N. Wiesel. Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex. *Journal of Physiology (London)*, 160:106–154, January 1962.

[48] Y. LeCun and Y. Bengio. *The Handbook of Brain Theory and Neural Networks*, chapter *Convolutional Networks for Images, Speech, and Time-Series*. MIT Press, June 1995.

[49] S. L. Phung and A. Bouzerdoum. MATLAB Library for Convolutional Neural Network. Technical report, ICT Research Institute, Visual and Audio Processing Laboratory, University of Wollongong, Wollongong, New South Wales, Australia, 2009. Available at: `http://www.elec.uow.edu.au/staff/sphung`.

[50] S. Duffner. *Face Image Analysis With Convolutional Neural Networks*. PhD thesis, Fakultät für Angewandte Wissenschaften, Albert-Ludwigs-Universität Freiburg, Germany, 2007. http://www.freidok.uni-freiburg.de/volltexte/4835/.

[51] M. Jahrer. Handwritten digit recognition using a Convolutional Neural Network. University of Technology, Graz, Unpublished, September 2008.

[52] N. Hampshire. *Commodore 64 Graphics*. Macmillan Computer Publishing, December 1984.

[53] J. Richter and B. Smith. *Graphics Programming for the 8514/A: The New PC Graphics Standard*. M&T Books, April 1990.

[54] J. Sanchez and M. P. Canton. *Software Solutions for Engineers and Scientists*, chapter *Displaying Bit-Mapped Images*, pages 687–719. CRC Press, October 2007.

[55] C. Hecker. A Whirlwind Tour of WinG. *Game Developer*, pages 14–18, September 1994. Available at: http://chrishecker.com/images/6/66/Gdmwing.pdf.

[56] B. Timmins. *DirectDraw Programming*. M&T Books, March 1996.

[57] W. K. Giloi, J. L. Encarnação, and W. Straßer. The Giloi's School of Computer Graphics. *ACM SIGGRAPH Computer Graphics*, 35(4):12–16, November 2001.

[58] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C, 2nd Edition*. Addison-Wesley Professional, August 1995.

[59] The Khronos Group. OpenGL – The Industry's Foundation for High Performance Graphics. http://www.khronos.org/opengl, August 2009.

[60] D. Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1, 7th Edition*. Addison-Wesley Professional, July 2009.

[61] Microsoft Corporation. Microsoft DirectX 11. http://www.microsoft.com/games/en-US/aboutgfw/Pages/directx10-a.aspx, August 2009.

[62] F. D. Luna. *Introduction to 3D Game Programming with DirectX 10*. Wordware Publishing, October 2008.

[63] J. Lengyel, M. Reichert, B. R. Donaldy, and D. P. Greenbergz. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1990)*, pages 327–335, Dallas, Texas, USA, August 1990.

[64] C.-A. Bohn. Kohonen Feature Mapping Through Graphics Hardware. In *Proceedings of the 3rd International Conference on Computational Intelligence and Neurosciences (ICCIN 1998)*, pages 64–67, Research Triangle Park, Durham, 1998.

[65] G. Kedem and Y. Ishihara. Brute Force Attack on UNIX Passwords with SIMD Computer. In *Proceedings of the 8th USENIX Security Symposium*, Washington, D.C., USA, August 1999.

[66] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL Shading Language, 3rd Edition.* Addison-Wesley Professional, July 2009.

[67] S. St-Laurent. *The COMPLETE Effect and HLSL Guide.* Paradoxal Press, July 2005.

[68] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2003)*, pages 896–907, San Diego, California, USA, July 2003.

[69] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Professional, March 2003.

[70] M. Macedonia. The GPU Enters Computing's Mainstream. *IEEE Computer*, 36(10):106–108, October 2003.

[71] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[72] J. Krüger and R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Transactions on Graphics*, 22(3):908–916, July 2003.

[73] A. Rege (NVIDIA Developer Technology Group). Shader Model 3.0. ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/Shader_Model_3.pdf, 2004.

[74] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[75] S. Patidar, S. Bhattacharjee, J. M. Singh, and P. J. Narayanan. Exploiting the Shader Model 4.0 Architecture. Unpublished, http://www.iiit.ac.in/techreports/2007_145.pdf, October 2007.

[76] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, August 2007.

[77] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[78] The Khronos Group. OpenCL Overview. http://www.khronos.org/opencl, August 2009.

[79] NVIDIA Corporation. NVIDIA OpenCL – Programming Guide. http://www.nvidia.de/object/cuda_opencl.html, November 2009.

[80] C. Boyd (Microsoft). DirectX 11 Compute Shader. http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf, August 2008. The 35th International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH 2008), Los Angeles, California, USA.

[81] C. Boyd (Microsoft) and M. Schmit (Advanced Micro Devices). The Direct3D11 Compute Shader. http://download.microsoft.com/download/5/E/6/5E66B27B-988B-4F50-AF3A-C2FF1E62180F/GRA-T517_WH08.pptx, November 2008. Windows Hardware Engineering Conference (WinHEC 2008), Los Angeles, California, USA.

[82] N. Thibieroz (AMD). Shader Model 5.0 and Compute Shader. http://developer.amd.com/gpu_assets/Shader%20Model%205-0%20and%20Compute%20Shader.pps, March 2009. Game Developers Conference 2009, San Francisco, California, USA.

[83] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March–April 2008.

[84] D. Kanter. NVIDIA's GT200: Inside a Parallel Processor. Real World Technologies – In-Depth Technical Analyses And Help For Professionals, http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242, September 2008.

[85] J. Stewart. *Calculus, 5th Edition.* Brooks Cole, December 2002.

[86] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, November 2009.

[87] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser, December 2009.

[88] N. Brookwood. NVIDIA Solves the GPU Computing Puzzle. Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/N.Brookwood_NVIDIA_Solves_the_GPU_Computing_Puzzle1.pdf, September 2009.

[89] P. N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf, September 2009.

[90] T. R. Halfhill. Looking Beyond Graphics. Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_Looking_Beyond_Graphics.pdf, September 2009.

[91] D. Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf, September 2009.

[92] W. W. Peterson and E. J. Weldon. *Error-Correcting Codes, 2nd Edition.* MIT Press, March 1972.

[93] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16. North-Holland Mathematical Library, 1977.

[94] S. Lin and D. J. Costello. *Error Control Coding, 2nd Edition.* Prentice Hall, June 2004.

[95] R. Morelos-Zaragoza. *The Art of Error Correcting Coding*. Wiley, September 2006.

[96] NVIDIA Corporation. NVIDIA CUDA – Programming Guide (ver. 2.0). `http://www.nvidia.com/object/cuda_home.html`, June 2008.

[97] NVIDIA Corporation. NVIDIA CUFFT Library. `http://www.nvidia.com/object/cuda_home.html`, August 2009.

[98] NVIDIA Corporation. NVIDIA CUDA C Programming – Best Practices Guide (CUDA Toolkit 2.3). `http://www.nvidia.com/object/cuda_home.html`, July 2009.

[99] P. H. Ha, P. Tsigas, and O. J. Anshus. The Synchronization Power of Coalesced Memory Accesses. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC 2008)*, pages 320–334, Arcachon, France, September 2008.

[100] K.-S. Oh and K. Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, June 2004.

[101] D. Steinkraus, I. Buck, and P. Y. Simard. Using GPUs for Machine Learning Algorithms. In *Proceedings of the 8th International Conference on Document Analysis and Recognition (ICDAR 2005)*, volume 2, pages 1115–1120, Seoul, South Korea, August 2005.

[102] K. Chellapilla, S. Puri, and P. Y. Simard. High Performance Convolutional Neural Networks for Document Processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition (IWFHR 2006)*, La Baule, France, October 2006.

[103] S. Lahabar, P. Agrawal, and P. J. Narayanan. High Performance Pattern Recognition on GPU. In *Proceedings of the National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG 2008)*, pages 154–159, Gandhinagar, India, January 2008.

[104] G. Poli, J. H. Saito, J. F. Mari, and M. R. Zorzan. Processing Neocognitron of Face Recognition on High Performance Environment Based on GPU with CUDA Architecture. In *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2008)*, pages 81–88, Campo Grande, MS, Brazil, October 2008.

[105] D. Strigl, K. Kofler, and S. Podlipnig. Performance and Scalability of GPU-based Convolutional Neural Networks. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010)*, Pisa, Italy, February 2010.

[106] Y. LeCun and C. Cortes. MNIST Handwritten Digit Database. `http://yann.lecun.com/exdb/mnist`, August 2009.

[107] Computational and Biological Learning Laboratory, Courant Institute of Mathematical Sciences, New York University. EBLearn: Energy-Based Learning, a C++ Machine Learning Library. `http://eblearn.sourceforge.net`, November 2009.

[108] P. Sermanet, K. Kavukcuoglu, and Y. LeCun. EBLearn: Open-Source Energy-Based Learning in C++. In *Proceedings of the 21st IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2009)*, pages 693–697, Newark, New Jersey, USA, November 2009. IEEE Computer Society. Available at: `http://yann.lecun.com/exdb/publis/pdf/sermanet-ictai-09.pdf`.

[109] V. N. Vapnik. *The Nature of Statistical Learning Theory, 2nd Edition.* Springer, 2000.

[110] A. Sinha. An Improved Recognition Module for the Identification of Handwritten Digits. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (MIT), May 1999. Available at: `http://web.mit.edu/profit/PDFS/SinhaA.pdf`.

[111] Y. H. Tay, P.-M. Lallican, M. Khalid, C. Viard-Gaudin, and S. Knerr. An Offline Cursive Handwritten Word Recognition System. In *Proceedings of the IEEE Region 10 International Conference on Electrical and Electronic Technology (TENCON 2001)*, volume 2, pages 519–524, 2001.

[112] S.-C. B. Lo, S.-L. A. Lou, J.-S. Lin, M. T. Freedman, and M. V. Chien. Artificial Convolution Neural Network Techniques and Applications for Lung Nodule Detection. *IEEE Transaction on Medical Imaging*, 14(4):711–718, December 1995.

[113] T. Fawcett. ROC Graphs: Notes and Practical Considerations for Researchers. Technical report, HP Laboratories, MS 1143, 1501 Page Mill Road, Palo Alto, CA 94304, March 2004. Available at: `http://home.comcast.net/~tom.fawcett/public_html/papers/ROC101.pdf`.

[114] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006. Available at: `http://home.comcast.net/~tom.fawcett/public_html/papers/ROC101-PRL.pdf`.

[115] H. A. Rowley, S. Baluja, and T. Kanade. Neural Network-Based Face Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, January 1998.

[116] K. Sung and T. Poggi. Example-Based Learning for View-Based Human Face Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):39–51, January 1998.

[117] R. Féraud, O. Bernier, J. Viallet, and M. Collobert. A Fast and Accurate Face Detector Based on Neural Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(1):42–53, January 2001.

[118] F. H. C Tivive. *A new class of convolutional neural networks based on shunting inhibition with applications to visual pattern recognition.* PhD thesis, School of Electrical, Computer and Telecommunications Engineering, University of Wollongong, Australia, 2006. `http://ro.uow.edu.au/theses/540/`.

[119] T. Tollenaere. SuperSAB: Fast Adaptive Back Propagation with Good Scaling Properties. *Neural Networks*, 3(5):561–573, 1990.

[120] S. E. Fahlman. An Empirical Study of Learning Speed in Back-Propagation Networks. Technical Report CMU-CS-88-162, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1988.

[121] C. Y. Suen and J. Tan. Analysis of errors of handwritten digits made by a multitude of classifiers. *Pattern Recognition Letters*, 26(3):369–379, February 2005.

[122] P. Y. Simard, Y. LeCun, and J. Denker. Efficient Pattern Recognition Using a New Transformation Distance. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems (NIPS 1992)*, volume 5, pages 50–58, San Mateo, California, USA, 1993.

[123] R. Mare, P. Geurts, J. Piater, and L. Wehenkel. A Generic Aproach for Image Classification Based on Decision Tree Ensembles and Local Sub-Windows. In *Proceedings of the 6th Asian Conference on Computer Vision (ACCV 2004)*, volume 2, pages 860–865, Jeju Island, Korea, January 2004.

[124] N. Matsumoto, S. Uchida, and H. Sakoe. Prototype Setting for Elastic Matching-Based Image Pattern Recognition. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004)*, volume 1, pages 224–227, Cambridge, United Kingdom, August 2004.

[125] D. Keysers, J. Dahmen, T. Theiner, and H. Ney. Experiments with an Extended Tangent Distance. In *Proceedings of the 15th International Conference on Pattern Recognition (ICPR 2000)*, volume 2, pages 38–42, Barcelona, Spain, September 2000.

[126] L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, L. D. Jackel, Y. LeCun, U. A. Müller, E. Säckinger, P. Y. Simard, and V. Vapnik. Comparison of Classifier Methods: A Case Study in Handwritten Digit Recognition. In *Proceedings of the 12th International Conference on Pattern Recognition (ICPR 1994)*, pages 77–82, Jerusalem, Israel, October 1994.

[127] G. Mayraz and G. Hinton. Recognizing Handwritten Digits Using Hierarchical Products of Experts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(2):189–197, February 2002.

[128] J. Milgram, R. Sabourin, and M. Cheriet. Combining Model-based and Discriminative Approaches in a Modular Two-stage Classification System: Application to Isolated Handwritten Digit Recognition. *Electronic Letters on Computer Vision and Image Analysis*, 5(2):1–15, 2005.

[129] B. Schölkopf. *Support Vector Learning*. Oldenbourg Verlag, Munich, 1997.

[130] J.-X. Dong, A. Krzyzak, and C. Y. Suen. Local learning framework for handwritten character recognition. *Engineering Applications of Artificial Intelligence*, 15(2):151–159, April 2002.

[131] B. Schölkopf, P. Y. Simard, A. Smola, and V. Vapnik. Prior Knowledge in Support Vector Kernels. In M. I. Jordan, M. J. Kearns, and S. A. Solla, editors, *Advances in Neural Information Processing Systems (NIPS 1997)*, volume 10, pages 640–646, Denver, Colorado, USA, June 1998.

[132] L.-N. Teow and K.-F. Loe. Handwritten Digit Recognition with a Novel Vision Model that Extracts Linearly Separable Features. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR 2000)*, volume 2, pages 76–81, Hilton Head Island, South Carolina, USA, June 2000.

[133] D. DeCoste and B. Schölkopf. Training Invariant Support Vector Machines. *Machine Learning*, 46(1-3):161–190, 2002.

[134] S. Belongie, J. Malik, and J. Puzicha. Shape Matching and Object Recognition Using Shape Contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, April 2002.

[135] J.-X. Dong, A. Krzyzak, and C. Y. Suen. Fast SVM Training Algorithm with Decomposition on Very Large Data Sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(4):603–618, April 2005. Additional results available at: `http://www.cenparmi.concordia.ca/~people/jdong/HeroSvm.html`.

[136] L.-N. Teow and K.-F. Loe. Robust Vision-Based Features and Classification Schemes for Off-Line Handwritten Digit Recognition. *Pattern Recognition*, 35(11):2355–2364, November 2002.

[137] V. Athitsos, J. Alon, and S. Sclaroff. Efficient Nearest Neighbor Classification Using a Cascade of Approximate Similarity Measures. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, volume 1, pages 486–493, San Diego, California, USA, June 2005.

[138] D. Keysers, C. Gollan, and H. Ney. Local Context in Non-linear Deformation Models for Handwritten Character Recognition. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004)*, volume 4, pages 511–514, Cambridge, United Kingdom, August 2004.

[139] C.-L. Liu, K. Nakashima, H. Sako, and H. Fujisawa. Handwritten Digit Recognition: Benchmarking of State-of-the-Art Techniques. *Pattern Recognition*, 36(10):2271–2285, October 2003.

[140] MIT Center for Biological and Computational Learning. CBCL Face Database #1. `http://cbcl.mit.edu/software-datasets/FaceData2.html`, August 2000.

[141] S. L. Phung, A. Bouzerdoum, and D. Chai. Skin Segmentation Using Color Pixel Classification: Analysis and Comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(1):148–154, 2005.

[142] B. Cyganek and J. P. Siebert. *An Introduction to 3D Computer Vision Techniques and Algorithms.* John Wiley & Sons, January 2009.