TECHNISCHE UNIVERSITÄT MÜNCHEN
DEPARTMENT OF MATHEMATICS

MASTER'S THESIS

A Comparison of Second Order Type Methods for Deep Learning

Rebecca König

Supervisor:	Prof. Dr. Michael Ulbrich
Advisors:	Dr. Andre Milzarek Sebastian Garreis, M.Sc. (hons) Dr. David Sattlegger (MVTec) Patrick Follmann, M.Sc. (MVTec)
Submission date:	May 12, 2017

Declaration of Authorship

I, Rebecca König, hereby declare that this thesis is my own work and that no other sources have been used except those clearly indicated and referenced.

Munich, May 12, 2017

Rebecca König

Technische Universität München

Abstract

Chair of Mathematical Optimization
Department of Mathematics

Master's Thesis

A Comparison of Second Order Type Methods for Deep Learning

by Rebecca König

In computer vision image classification is a crucial task, which is best tackled by machine learning with convolutional neural networks. Training such deep networks is a difficult challenge in mathematical optimization since the minimization problem is nonconvex and large-scale and the training dataset contains a large number of images. The standard optimization methods that are used are stochastic gradient descent and some modifications of it. However, these first order methods need careful tuning of hyperparameters and many iterations are necessary until convergence. Thus, second order type methods, which compute the search direction and step size adaptively by taking curvature information into account, can be very useful in deep learning We implemented promising second order optimization methods and evaluate their performance compared to first order methods on different datasets and net architectures. Because of the higher per-iteration-complexity the second order methods take up more time and memory, but can be superior considering classification accuracy.

Acknowledgements

I would like to thank my first advisor Andre Milzarek who accompanied me for the most part of this work, but unfortunately moved abroad before I could finish the thesis.

Furthermore, I want to thank Sebastian Garreis who took over the supervision with great enthusiasm and gave me thoughtful advice and guidance.

Most of all, I would like to express my thanks to the whole company MVTec and in particular its research team for the excellent cooperation which made the successful outcome of this thesis possible. I am especially grateful to my advisors, David Sattlegger and Patrick Follmann, who always had an open ear for my problems and questions, and supported me with great commitment in all respects. They taught me tips and tricks which helped me to survive stepping into the world of software engineers.

Last but not least, a special thank goes to the other students at MVTec, with whom I shared many great experiences and who made the last few months so enjoyable.

Contents

List of Abbreviations

AI	**A**rtificial **I**ntelligence
BFGS	**B**royden-**F**letcher-**G**oldfarb-**S**hanno
conv	**conv**olution
CG	**C**onjugate **G**radient
CNN	**C**onvolutional **N**eural **N**etwork
fc	**f**ully-**c**onnected
G	**G**auss-Newton
GPU	**G**raphics **P**rocessing **U**nit
H	**H**essian
L-BFGS	**L**imited memory **B**royden-**F**letcher-**G**oldfarb-**S**hanno
MLP	**M**ulti**L**ayer **P**erceptron
OCR	**O**ptical **C**haracter **R**ecognition
PCA	**P**rinciple **C**omponent **A**nalysis
PCG	**P**reconditioned **C**onjugate **G**radient
ReLU	**Re**ctified **L**inear **U**nit
RMS	**R**oot **M**ean **S**quared
SCG	**S**caled **C**onjugate **G**radient
SGD	**S**tochastic **G**radient **D**escent
SVHN	**S**treet **V**iew **H**ouse **N**umbers

List of Symbols

$\mathbb{1}_{(n,m)}$	matrix of ones with dimension $n \times m$
$a^{(\ell)}$	activation of layer ℓ
$b^{(\ell)}$	bias of layer ℓ
c	output label of the network
C	loss function of the network including weight decay regularization
C_0	loss function of the network on the whole dataset
C_r	function mapping logits of last network layer to loss
C_s	loss function of the network on one sample
d	dimension of input data to the network
E	expected risk function
E_N	empirical risk function
f	function representing a fully connected neural network
f_C	function representing a convolutional neural network
f_r	function mapping input and parameters to logits of last layer
$\mathbf{G}$	Gauss-Newton matrix
h	classification function
$\mathbf{H}$	Hessian matrix
$\mathbf{I}$	identity matrix
$\mathbf{J}_f$	Jacobian of a function f
ℓ	layer in the network
l	logistic function
L	number of layers in the network
L_C	number of convolution layers in the CNN
L_F	number of fully connected layers in the CNN
m	number of classes
n	dimension of θ
$n^{(\ell)}$	number of units in layer ℓ
N	number of samples in the training set
N_B	number of samples in one minibatch
t	function mapping ground truth to one-hot vector
$W^{(\ell)}$	weight matrix of layer ℓ
$w^{(\ell)}$	filter weights of layer ℓ in a CNN
$\boldsymbol{x}$	input data to the network
$\boldsymbol{X}$	set of all training samples
y	output probability for each class of the network
$\hat{y}$	ground truth class label
$\hat{Y}$	set of all ground truth class labels
$z^{(\ell)}$	logit of layer ℓ
∇g	gradient of a function g
$\nabla^2 g$	Hessian of a function g
ϕ	activation function
σ	softmax function
θ	parameter vector containing all weights and biases of the network

Chapter 1

Introduction

Creating self-thinking machines has been on the mind of mankind for a long time. In the Greek mythology Hephaestus, the Greek god of craftsmen, blacksmiths and artisans, built metal men, so-called automatons, who worked for him [Nel16]. Later, in the 19th century, Mary Shelley wrote about Frankenstein who created a human-like intelligent being. People wished to have God's ability to endow creatures with intelligence.

Nowadays, artificial intelligence (AI) is a thriving field in computer science. It is easy to hard-code formal knowledge for applications such as chess computers – for example IBM's Deep Blue [CHH02] – which can outperform humans. By logical inference rules the best action can be deduced. While these inferences can be complex to the human brain, computers can draw correct conclusions easily given all rules.

Sometimes, though, the rules provided to the machine by the scientist are not enough. It is necessary for the computer to extract patterns from the data on its own. This approach is called *machine learning* [GBC16]. An example is the decision whether to recommend a cesarean delivery, based on logistic regression [MYSM$^+$90]. The doctor provides data, such as the presentation of the fetus and the presence of a uterine scar, which is processed to make a decision. Each piece of information in the data representation is called a feature. It is not straightforward to apply formal inference rules to decide based on the input features. However, complex correlations of different features are learned by defining them on simpler concepts. This hierarchical structure of concepts can have many stages, inducing the term *deep learning*.

The outcome of this procedure is strongly dependent on the representation of the data. It is not immediately clear in every scenario which representation is a good choice for a specific problem. There are tasks that cannot be described by a formal knowledge base. Among these tasks are for example speech recognition and image understanding. For humans it is natural to recognize any known object in an image. But how can for example a cat be described formally so that a computer can recognize it? It is not the image pixels' values that describe it. A cat can have different fur colors and patterns, it can lie on the ground or jump. The objects on an image can appear in different shapes, intensities or can be occluded by some other object. Thus, a different data representation is necessary that the machine can understand and process correctly. Since it would take a nearly infinite amount of time to conceive and hard-code good representations for all objects, the approach to tackle this problem is to let the computer learn the representation on its own by extracting patterns from raw data with deep learning.

For deep learning a lot of labeled data is necessary to produce good results. Therefore, only in the last years, when the era of *big data* started, deep learning has become more and more important as in the course of digitization a continuously increasing amount of data is generated, recorded and distributed through the world wide web [GBC16]. Due to the high availability of data, research and application of deep learning has experienced a strong growth. Furthermore, the advance of deep learning benefits from the rapid progress in equipping computers with more and more computational power and memory. Especially the development of graphical processing units (GPUs) contributes to the success of deep learning, since convolutional neural networks, the main algorithm in image classification, are highly parallelizable [Sch15].

The focus of this work lies on the task of image classification by training convolutional neural networks (CNN). A CNN is a powerful tool for representation learning and subsequent feature classification. In computer vision image classification is an important field, with applications for example in autonomous driving, where objects have to be categorized in real time and with a high accuracy.

Training deep convolutional neural networks is realized by optimizing a complex objective function. The research concentrates on more and more advanced network architectures, whereas as optimization method still comparably simple first order methods are used. However, the mathematical research contributes much more advanced optimization methods to the range of tools. They are more complex to implement and usually consume more memory. Albeit, these second order methods have better theoretical convergence properties than first order methods. In this thesis second order optimization methods applied to deep learning are investigated. Their performance is compared to each other and the most common used method stochastic gradient descent.

The outline is as follows. In Chapter 2 we will introduce basic neural networks and the components that are needed for training. Chapter 3 then describes the special instance of convolutional neural networks. We give details on how second order information can be efficiently computed in neural networks in Chapter 4. In Chapter 5 the concept of training a neural network is explained and some widely used first order optimization methods are briefly described. The theory of the investigated second order optimization methods is derived in Chapter 6 before we perform numerical experiments on real data in Chapter 7.

This thesis was written in cooperation with MVTec Software GmbH[1], which provided software and hardware as well as the image data used in the experiments.

[1] Arnulfstr. 205, 80634 München, Germany. www.mvtec.com

Chapter 2

Neural networks

Neural networks are a popular tool in deep learning. In this chapter we will describe the layout of a neural network, its components and functionality.

2.1 Biological inspiration

In computer science, neural networks are algorithms that try to mimic the biological functionality of the human brain. Their main component is constituted by neurons. Neurons can be either active and fire an electrical impulse or inactive. Each neuron has several dendrites (input wires) through which it receives the activation from connected neurons. In the neuron's nucleus these signals are processed and eventually an impulse is passed on through the axon (output wire) to other connected neurons (see [Kri07] for detailed information). An important property of the brain is its ability to learn things. Different parts of the brain are responsible for different tasks, such as speech, vision or hearing. In the case of an early brain damage, these tasks can be relocated to other parts so that the brain is still functional [RM99]. That means that each neuron in the brain can learn anything, its purpose is not necessarily predetermined. This is exactly the function that a neural network should fulfill. It is supposed to learn features of the input without somebody telling it in advance what it should look for.

2.2 Structure of a neural network

In general, a neural network that is applied to a classification problem is a function $F : \mathbb{R}^d \to \mathbb{R}$. It can be represented by a composition of several functions. These functions can be viewed as layers of multiple processing units, called neurons. Each network has an input layer which represents the data fed into the network and an output layer which is the output determined by the network for the corresponding input. The layers between input and output layer are the so called hidden layers as the calculations done inside these layers are usually not visible to the user. The user is only interested in the input and output of the algorithm. A neural network can thus be seen as a representation of a blackbox function which takes as input some data vector $\boldsymbol{x} \in \mathbb{R}^d$, e.g. an image, and outputs a label c that corresponds to the input (Figure 2.1). Depending on the problem, c can be a value in $\mathbb{R}$, for example in logistic regression, or – as it is the case in this work – $c \in \mathbb{N}$ when considering a classification task. $d \in \mathbb{N}$ is the dimension of the input data. The function F that maps $\boldsymbol{x} \mapsto c$ typically is not known, otherwise an exact algorithm for computing $F(\boldsymbol{x})$ could be implemented. The neural network, however, is trained to learn how the input features correlate to produce the desired outcome.

The goal is to replace the blackbox by a neural net that learns from the data how the input is mapped to the output. In a simple example, the neural net has only one hidden layer. In Figure 2.2 the schematic structure of such a neural net is shown. Here the input $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$ consists of two scalar values and the hidden layer has three neurons. As

Figure 2.1: Scheme of blackbox function to get label c for input $\boldsymbol{x}$.

all input units are connected to all units in the hidden layer, this type of layer is called *fully connected layer*. Furthermore, in this special case, we want to decide between two classes, so that the output layer has one unit which takes either the value 0 or 1. In general, the number of possible class labels is m and the output layer consists of m units if $m > 2$.

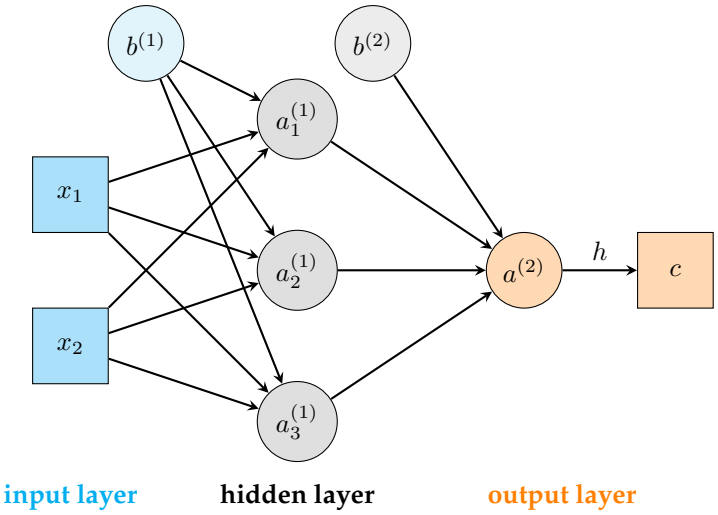


Figure 2.2: Example of a neural network with one hidden layer.

We enumerate the layers by $\ell = 0, \ldots, L$ where the input layer corresponds to $\ell = 0$ and the output layer to $\ell = L$. In our example we have $L = 2$. By $a_i^{(\ell)}$ we denote the activation of neuron i in layer ℓ. In this type of neural network the activation of a neuron is computed by a linear combination of all the values in the previous layer, followed by the application of an activation function $\phi^{(\ell)} : \mathbb{R} \to \mathbb{R}$:

$$a_i^{(\ell)} = \phi^{(\ell)}(z_i^{(\ell)}), \text{ with} \tag{2.1}$$

$$z_i^{(\ell)} = W_{i,0}^{(\ell)} a_0^{(\ell-1)} + W_{i,1}^{(\ell)} a_1^{(\ell-1)} + \cdots + W_{i,n^{(\ell-1)}}^{(\ell)} a_{n^{(\ell-1)}}^{(\ell-1)} + b_i^{(\ell)} \tag{2.2}$$

where $n^{(\ell)}$ denotes the number of units in layer ℓ and $b_i^{(\ell)}$ is an additional bias term. The bias helps to fit a function to the data by shifting the activation function. We will refer to the variables $z^{(\ell)}$ as logits. Denoting the input $\boldsymbol{x}$ by $a^{(0)}$ (i.e. we have $n^{(0)} = d$) we can write these computations in vectorized form:

$$z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}, \tag{2.3}$$

$$a^{(\ell)} = \phi^{(\ell)}(z^{(\ell)}), \tag{2.4}$$

for $\ell = 1, \ldots, L$. Here, $a^{(\ell)}$, $b^{(\ell)}$, $z^{(\ell)} \in \mathbb{R}^{n^{(\ell)}}$ and the weight matrix $W^{(\ell)}$ is an element of $\mathbb{R}^{n^{(\ell+1)} \times n^{(\ell)}}$, where $n^{(L)} = m$. The activation function $\phi^{(\ell)}$ is applied elementwise here. The final label c is computed by the classification function $h : \mathbb{R}^m \to \mathbb{N}$ evaluated at the activation

of the last layer $a^{(L)}$. We summarize the weight matrices and biases in

$$W = (W^{(1)}, W^{(2)}, \dots, W^{(L)}), \tag{2.5}$$

$$b = (b^{(1)}, b^{(2)}, \dots, b^{(L)}). \tag{2.6}$$

We define the function $f : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ that maps an input $\boldsymbol{x}$ and the parameters W and b to the activation of the last layer: $f(\boldsymbol{x}; W, b) = a^{(L)}$. More precisely, f can be written as

$$f(\boldsymbol{x}; W, b) = \phi^{(L)}(W^{(L)}\phi^{(L-1)}(W^{(L-1)}\phi^{(L-2)}(\cdots) + b^{(L-1)}) + b^{(L)}). \tag{2.7}$$

Altogether, the neural network produces the output $c = h(f(\boldsymbol{x}; W, b))$ for an input $\boldsymbol{x}$. The composition $h \circ f$ should be as close as possible to the 'real' blackbox function F.

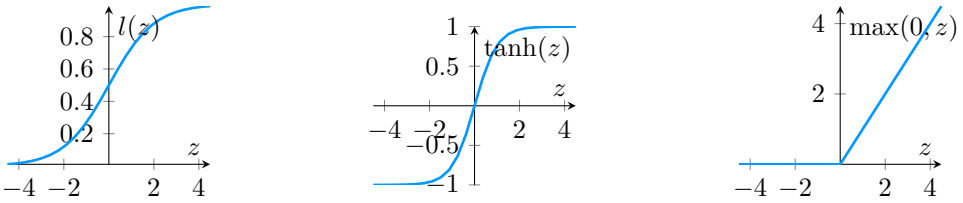
An input $\boldsymbol{x}$ is fed into the input layer and then propagated forward layer by layer until finally an output is computed. Therefore, this type of neural net is called *feed-forward*.

All components of a neural network were already defined above, in the following, we will explain activation functions ϕ and the classification function h in more detail. For convenience we summarize the weight matrices and biases in one parameter vector which we will denote in the following by $\theta \in \mathbb{R}^n$:

$$\theta = \left(W^{(1)}_{1,1}, W^{(1)}_{1,2}, \dots, W^{(1)}_{n^{(2)},n^{(1)}}, b^{(1)}_1, \dots, b^{(1)}_{n^{(2)}}, W^{(2)}_{1,1}, \dots, \dots, b^{(L)}_m\right)^\top. \tag{2.8}$$

2.3 Activation functions

The activation function models what happens in the nucleus of a neuron. It plays a crucial role in neural networks as it introduces nonlinearity. Especially in deep neural networks (i.e. those with more than one hidden layer), these nonlinearities are needed after every layer. Otherwise the network could be represented by a single linear transformation as compositions of linear transformations are still linear. Only by introducing nonlinear activation functions between the linear transformations, more complex functions can be represented by the neural network.



(a) Logistic function (b) Hyperbolic tangent (c) ReLU

Figure 2.3: The most common types of activation functions.

In the early literature on neural networks the most common activation functions were the logistic function (also called sigmoid function) and the hyperbolic tangent [Kri07], defined by

$$l : \mathbb{R} \to \mathbb{R}, \quad z \mapsto \frac{1}{1 + e^{-z}}, \tag{2.9}$$

and

$$\tanh : \mathbb{R} \to \mathbb{R}, \quad z \mapsto 1 - \frac{2}{e^{2z} + 1} \tag{2.10}$$

respectively. Both of these functions saturate if the absolute value of z approaches infinity (see Figure 2.3). Thus, their derivative almost vanishes for large absolute values of z. This is a nondesirable property as in this case not much information remains that can be used to

learn.
Therefore, another type of activation function has proven to be more efficient, a rectifier defined componentwise by the max-function:

$$\phi : \mathbb{R} \to \mathbb{R}, \qquad z \mapsto \max(0, z). \tag{2.11}$$

A neuron whose output is computed by a rectifier is called a *Rectified Linear Unit* (ReLU). This type of activation was first proposed in [GBB11] where the motivation was to find a better model for biological neurons. In particular, real neurons in a brain cannot have a negative activation [Kri07]. Douglas et al. [DKM+95] suggest that neurons' activations are rarely at a saturated level. Other advantages of a rectifier over different activation functions include its easy and cheap computation. There is no need to evaluate the exponential function. Besides, it induces sparse activation of neurons which seems to be suitable for natural images [GBB11]. Furthermore, Nair and Hinton [NH10] point out that with using ReLUs technically an exponential number of linear models that share parameters are created. This helps in classification, as will also be discussed later in Section 5.1. There is, however, a potential problem inhering the ReLU. Because of the hard nonlinearity and nondifferentiability of the maximum function at 0 it might be hard to optimize. Therefore, Glorot et al. [GBB11] also experimented with the softplus-function

$$\text{softplus} : \mathbb{R} \to \mathbb{R}, \qquad z \mapsto \log(1 + e^z) \tag{2.12}$$

which smoothly approximates the maximum function. It emerges from the approximation

$$\max(0, z) = \lim_{k \to \infty} \frac{1}{k} \log(1 + e^{kz}) \tag{2.13}$$

and the fact that the parameter k can be included in the previous and following layer of the network. Thus, k can be chosen to be 1 in (2.13) resulting in the softplus-function.

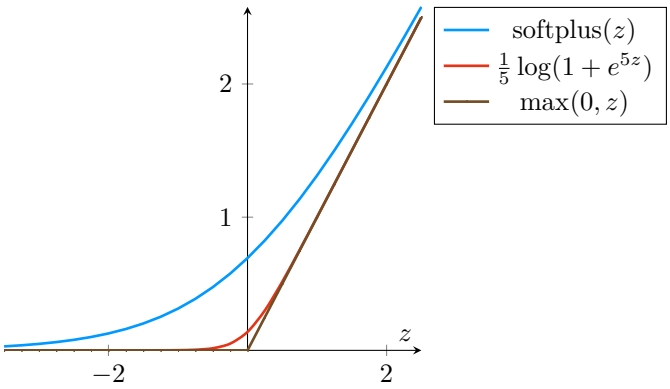


Figure 2.4: Comparison of the maximum function and two smooth approximations.

In Figure 2.4 the maximum function is compared to the softplus-function and an approximation $\frac{1}{k}\log(1 + e^{kz})$ with $k = 5$ which is already very close to the maximum function. However, Glorot et al. [GBB11] found that using the maximum function helps in supervised training and improves the obtained results. Krizhevsky, Sutskever and Hinton [KSH12] argue that ReLUs accelerate the decrease of the training error in their experiments by a factor of six compared to using the hyperbolic tangent. A modification of the ReLUs is to allow a small gradient, when the neuron is inactive. It is defined as

$$\phi(z) = \begin{cases} z & \text{if } z \geq 0 \\ cz & \text{if } z < 0 \end{cases} \tag{2.14}$$

for a small value of $c > 0$, such as 0.01. This version is called *leaky ReLU* and can be a solution to the problem of 'dying neurons' which occurs when all neurons are inactive, and the ReLU lets no gradient flow farther back and thus the network is stuck in this state.

In the following, we will use the derivative of the rectified linear unit. Since the maximum function is not differentiable, we will thus refer to the derivative of $\frac{1}{k}\log(1+e^{kz})$ for some large $k > 0$ instead:

$$\phi'(z) = \frac{1}{1+e^{-kz}}. \tag{2.15}$$

For the output layer we need a certain type of activation function as we interpret the activation of the output neuron as a probability. Therefore, in the case of one single output neuron (i.e., only two possible classes in classification), the logistic function (also called sigmoid function) is used as activation function for the output layer. Thus, $l(z)$ takes values in the interval $(0, 1)$ and we can interpret the value of $\sigma(z^{(L)})$ as the probability of being class 1, i.e. $\sigma(z) = P(c = 1|z)$. The classification function $h(z^L)$ outputs the predicted class label c:

$$c = h(z^{(L)}) = \begin{cases} 1 & \text{if } \sigma(z^{(L)}) \geq 0.5 \\ 0 & \text{if } \sigma(z^{(L)}) < 0.5 \end{cases}. \tag{2.16}$$

More generally, for $m > 2$, if we have m classes the last layer has m units and the activation function used in the last layer is the *softmax function*

$$\sigma : \mathbb{R}^m \to \mathbb{R}^m, \qquad z \mapsto \left(\frac{\exp(z_1)}{\sum_j \exp(z_j)}, \dots \frac{\exp(z_m)}{\sum_j \exp(z_j)} \right)^\top. \tag{2.17}$$

As all logits z_i are involved in the activation of each neuron, the softmax function is called a nonlocal nonlinearity. Note that the entries of $\sigma(z)$ sum to 1. The softmax function therefore represents a probability distribution. Each neuron in the last layer has the activation $y_i = a_i^{(L)} = \sigma(z)_i$, which expresses the probability of the input being classified as class i. The classification function h then simply takes the index of the maximum value of $a^{(L)}$ as the predicted class label c:

$$\begin{aligned} c &= h(a^{(L)}) && (2.18)\\ &= \arg\max_i \ a_i^{(L)} && (2.19)\\ &= \arg\max_i \ \sigma(z^{(L)})_i. && (2.20) \end{aligned}$$

2.4 Error function

In order to analyze the performance of the neural network one needs an error function – often called loss function – that can be computed when having a ground truth for the dataset. This error function is going to be the objective function in the optimization problem. Let us consider a set of N pairs $(\boldsymbol{X}, \hat{Y}) = (\boldsymbol{x}^i, \hat{y}^i)_{i\in[N]}$ where $\hat{y}^i \in \mathbb{N}$ is the correct class label for image $\boldsymbol{x}^i \in \mathbb{R}^d$. The neural network is represented by parameters $\theta \in \mathbb{R}^n$ and the architecture implicitly given by f outputs the predicted class $c^i = h(f(\boldsymbol{x}^i; \theta))$ for image $\boldsymbol{x}^i$.

In logistic regression, where the output $c = f(\boldsymbol{x}^i; \theta)$ is not a class label but a real number, one can compute the error $L(\boldsymbol{X}, \hat{Y}; \theta)$ by the summed squared difference of all actual and predicted labels:

$$L(\boldsymbol{X}, \hat{Y}; \theta) = \frac{1}{2N} \sum_{i=1}^{N} \left(\hat{y}^i - h(f(\boldsymbol{x}^i; \theta))\right)^2. \tag{2.21}$$

In classification, however, a different measure is used since the activations y_i of the last layer represent a probability distribution. Let us recall that in multi-class classification we

use the softmax activation function (2.17), and we have $y = \sigma(z^{(L)})$. To derive the loss function for this problem we use the maximum likelihood principle [GBC16]. Let $p_{data}(\boldsymbol{x})$ be the true probability distribution, from which the set $\boldsymbol{X}$ is drawn independently. Furthermore, let $p_{model}(\boldsymbol{x};\theta)$ be a parametric family of probability distributions, indexed by θ, that estimates the true probability $p_{data}(\boldsymbol{x})$. The maximum likelihood estimator for the parameters θ is then defined as

$$\theta^* = \arg\max_\theta \; p_{model}(\boldsymbol{X};\theta) \tag{2.22}$$

$$= \arg\max_\theta \prod_{i=1}^{N} p_{model}(\boldsymbol{x}^i;\theta). \tag{2.23}$$

Since it is more convenient and numerically stable to compute a sum rather than a product over many small probabilities, one can take the logarithm of (2.23) since it is strictly monotonic increasing. We then get the problem

$$\theta^* = \arg\max_\theta \sum_{i=1}^{N} \log p_{model}(\boldsymbol{x}^i;\theta). \tag{2.24}$$

The computation of the maximum likelihood estimator can be generalized to conditional probabilities [GBC16], which is the present situation here. Since we do not have access to the true probability distribution of the data and labels, our goal is to predict the true label $\hat{y}^i$ given $\boldsymbol{x}^i$ for the training data $(\boldsymbol{X}, \hat{Y})$. We introduce the function $t : \mathbb{R} \to \mathbb{R}^m$ that maps the ground truth $\hat{y}^i$ of sample i to the one-hot vector $t(\hat{y}^i) = (t_1(\hat{y}^i), \dots, t_m(\hat{y}^i))^\top$. It has only one nonzero entry, namely $t_j(\hat{y}^i) = 1$ for $j = \hat{y}^i$. Thus, we want to maximize the probability $p_{model}(t(\hat{y}^i)|\boldsymbol{x}^i;\theta)$ of predicting $t(\hat{y}^i)$ given $\boldsymbol{x}^i$ and θ for all samples $i = 1, \dots, N$. This probability approximates the true probability $p_{data}(t|\boldsymbol{x})$. By denoting $T = \{t(\hat{y}^1), \dots, t(\hat{y}^N)\}$ we have for the conditional maximum likelihood estimator

$$\theta^* = \arg\max_\theta \; p_{model}(T|\boldsymbol{X};\theta) \tag{2.25}$$

$$= \arg\max_\theta \prod_{i=1}^{N} p_{model}(t(\hat{y}^i)|\boldsymbol{x}^i;\theta) \tag{2.26}$$

$$= \arg\max_\theta \sum_{i=1}^{N} \log p_{model}(t(\hat{y}^i)|\boldsymbol{x}^i;\theta). \tag{2.27}$$

Let us recall that in multi-class classification we use the softmax activation function (2.17), and we have $y = f(\boldsymbol{x};\theta) = \sigma(f_r(\boldsymbol{x};\theta))$, where $f_r : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ is the part of the network function without the last softmax activation function. The probability that the network outputs a class $c \in \{1, \dots, m\}$ for the input $\boldsymbol{x}$ is then equal to the c-th activation of the last layer:

$$P(c|\boldsymbol{x},\theta) = (\sigma(f_r(\boldsymbol{x};\theta)))_c = (f(\boldsymbol{x};\theta))_c = y_c. \tag{2.28}$$

Thus, the probability of obtaining the target vector $t(\hat{y}^i)$ for sample i is exactly the probability of obtaining the class $\hat{y}^i$, for which $t^i_{\hat{y}^i} = 1$. Since $t(\hat{y}^i)$ is a one-hot vector we can write [Bis06, Chapter 4.3]:

$$p_{model}(t(\hat{y}^i)|\boldsymbol{x}^i;\theta) = \prod_{j=1}^{m} P(t_j(\hat{y}^i)|\boldsymbol{x}^i;\theta)^{t_j(\hat{y}^i)} = \prod_{i=1}^{m} (\sigma(f_r(\boldsymbol{x}^i;\theta))_j)^{t_j(\hat{y}^i)} = \prod_{i=1}^{m} (y^i_j)^{t_j(\hat{y}^i)}. \tag{2.29}$$

Algorithm 2.1 Forward pass

Input: Image $\boldsymbol{x}$ with label $\hat{y} \in \{1, \dots, m\}$
$a^0 \leftarrow \boldsymbol{x}$
for $\ell = 1, 2, \dots, L$ **do**
$z^{(\ell)} \leftarrow W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}$
$a^{(\ell)} \leftarrow \phi^{(\ell)}(z^{(\ell)})$
end for
$y \leftarrow a^{(L)}$
$c \leftarrow \arg\max_i \ y_i$
$C \leftarrow -\log y_{\hat{y}}$
Output: predicted label c and loss $C = C_s(\boldsymbol{x}, \hat{y}; \theta)$

Substituting (2.29) into (2.27) we get

$$\theta^* = \arg\max_\theta \sum_{i=1}^{N} \log \left(\prod_{j=1}^{m} (y_j^i)^{t_j(\hat{y}^i)} \right) \tag{2.30}$$

$$= \arg\max_\theta \sum_{i=1}^{N} \sum_{j=1}^{m} \log \left((y_j^i)^{t_j(\hat{y}^i)} \right) \tag{2.31}$$

$$= \arg\max_\theta \sum_{i=1}^{N} \sum_{j=1}^{m} t_j(\hat{y}^i) \log y_j^i, \tag{2.32}$$

with $y^i = f(\boldsymbol{x}^i; \theta)$. We can scale this function by $\frac{1}{N}$ without changing the arg max to get as error function the empirical distribution defined over the given data $(\boldsymbol{X}, \hat{Y})$. We scale it by -1 as we rather want to have a minimization problem. The argument of the solution does not change by doing so. Thus, we can define the error function

$$C_0 : \mathbb{R}^{d \times N} \times \mathbb{R}^N \times \mathbb{R}^n, \quad C_0(\boldsymbol{X}, \hat{Y}; \theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{m} t_j(\hat{y}^i) \log \left(f(\boldsymbol{x}^i; \theta) \right)_j . \tag{2.33}$$

The function C_0 is known as the *cross entropy function* for multi-class classification problems. We will denote the cross entropy loss on a single example $(\boldsymbol{x}, \hat{y})$ by the function

$$C_s : \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}, \quad C_s(\boldsymbol{x}, \hat{y}; \theta) = -\sum_{j=1}^{m} t_j(\hat{y}) \log \left(f(\boldsymbol{x}; \theta) \right)_j \tag{2.34}$$

such that the loss on the whole dataset becomes $C_0(\boldsymbol{X}, \hat{Y}; \theta) = \frac{1}{N} \sum_{i=1}^{N} C_s(\theta, \boldsymbol{x}^i, \hat{y}^i)$. This function can be used as a reasonable error function as it is always positive since $t_j(\hat{y}^i) \in \{0, 1\}$, $\left(f(\boldsymbol{x}^i; \theta) \right)_j \in (0, 1)$ for all $i = 1, \dots, N$ and all $j = 1, \dots, m$ and $\log x \in (-\infty, 0)$ for $x \in (0, 1)$. Furthermore, the loss value $C_0(\boldsymbol{x}, \hat{y}; \theta)$ for one single example $\boldsymbol{x}$ with ground truth $\hat{y}$ is 0, if the probability of the correct class is 1, and it grows as the uncertainty of the network's output to be the target class increases. The function C_s is differentiable as a composition of differentiable functions. Therefore, the function C_0 is differentiable as well.

The algorithm for a forward pass and loss computation on a single example is summarized in Algorithm 2.1. Note that the loss computation does not include the classification function h. This is important as h is not continuous and thus not differentiable, which will be necessary to compute the gradient of C_0 in the next section.

The loss function measures the performance of the neural network that is represented by its architecture and the corresponding parameters θ. This parameter vector will be the optimization variable in the minimization problem. We obviously want the error to be as

small as possible. Therefore, the goal in training neural networks is to find a parameter vector θ^* that minimizes the loss function $C_0(\boldsymbol{X}, \hat{Y}; \theta)$:

$$\theta^* = \arg\min_{\theta} \; C_0(\boldsymbol{X}, \hat{Y}; \theta). \tag{2.35}$$

Since n can be a very large number ($\sim 10^6$) depending on the size and number of layers in the network, the training of a deep network results in a large-scale optimization problem. The optimization is usually done by iterative methods which are described in Chapters 5 and 6.

2.5 Backpropagation

All of the algorithms to be discussed later use the gradient with respect to the parameters to iteratively compute new parameters that are ideally converging to the global minimum. An efficient method to compute the gradient of neural networks is the *backpropagation algorithm* which was presented by Rumelhart et al. [RHW85]. The goal of backpropagation is to compute the partial derivatives to all components of the parameter vector θ, i.e. to all elements of the weights W and biases b. In its core the backpropagation algorithm is just a composed application of the chain rule. Thus, backpropagation is a special application of automatic differentiation [BPRS15]. For each example it starts to compute the derivative of the error function with respect to the activations $y_j = a_j^{(L)}$ in the last layer, i.e. the output of the network function f. The error is propagated back through the net. This computation is therefore called *backward pass* – in contrast to the *forward pass* for computing the loss. In the proof of Theorem 2.1 we will derive the formulae for the special case of cross entropy loss and softmax activation for one single example here. The total gradient is then just the average of all N sample gradients:

$$\nabla_\theta C_0(\boldsymbol{X}, \hat{Y}; \theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\left(\frac{\partial C_s}{\partial W_{1,1}^{(1)}}, \ldots, \frac{\partial C_s}{\partial b_1^{(1)}}, \ldots, \frac{\partial C_s}{\partial W_{1,1}^{(2)}}, \ldots, \frac{\partial C_s}{\partial b_m^{(L)}} \right) (\boldsymbol{x}^i, \hat{y}^i; \theta) \right)^\top. \tag{2.36}$$

We followed the description of backpropagation by Goodfellow et al. [GBC16] and adapted the details and notations to formulate and prove the following theorem.

Theorem 2.1. *Let us consider a neural network for multi-class classification, represented by a function f (2.7), where the last layer is a softmax activation function. Assume that all activation functions $\phi^{(\ell)}$, $\ell = 1, \ldots, L$, of the network are differentiable.*

Then, the corresponding loss function C_s on a single sample, given by (2.34), is differentiable, and its gradient with respect to the parameters $\theta \in \mathbb{R}^n$ can be computed by the backpropagation algorithm 2.2.

Proof. The function C_s is differentiable as a composition of differentiable functions.

For the gradient computation we first of all assume that a forward pass has already been executed and its results are available for the backward pass.

Recall the cross entropy error function for one single example:

$$C_s(\boldsymbol{x}, \hat{y}; \theta) = -\sum_{j=1}^{m} t_j(\hat{y}) \log(f(\boldsymbol{x}; \theta))_j, \tag{2.37}$$

where $t_j(\hat{y}) = 1$ if and only if $\boldsymbol{x}$ belongs to class j, i.e. $\hat{y} = j$. Its derivative with respect to $y_i = (f(\boldsymbol{x}; \theta))_i \in (0, 1)$ is given by

$$\frac{\partial C_s}{\partial y_i} = -\frac{t_i(\hat{y})}{y_i}. \tag{2.38}$$

Then, according to the chain rule, this derivative is used to calculate the derivative with respect to the logit $z_i^{(L)}$ of the last layer. As $z_i^{(L)}$ affects all activations y_j in the last layer, one has to sum over all j:

$$\frac{\partial C_s}{\partial z_i^{(L)}} = \sum_{j=1}^{m} \frac{\partial C_s}{\partial y_j} \frac{\partial y_j}{\partial z_i^{(L)}}. \tag{2.39}$$

y_j is given by the softmax function on $z^{(L)}$:

$$y_j = \frac{\exp(z_j^{(L)})}{\sum_k \exp(z_k^{(L)})}. \tag{2.40}$$

Thus, the partial derivatives of y_j with respect to $z_i^{(L)}$ are given by

$$\frac{\partial y_j}{\partial z_i^{(L)}} = \frac{\exp(z_j^{(L)}) \left(\sum_k \exp(z_k^{(L)}) - \exp(z_j^{(L)}) \right)}{\left(\sum_k \exp(z_k^{(L)}) \right)^2} = y_j(1 - y_j) \tag{2.41}$$

if $j = i$. And for $j \neq i$ we have

$$\frac{\partial y_j}{\partial z_i^{(L)}} = \frac{-\exp(z_j^{(L)}) \exp(z_i^{(L)})}{\left(\sum_k \exp(z_k^{(L)}) \right)^2} = -y_j y_i. \tag{2.42}$$

Substituting (2.38),(2.41) and (2.42) into (2.39), we get

$$\begin{aligned} \frac{\partial C_s}{\partial z_i^{(L)}} &= \sum_{j=1}^{m} \frac{\partial C_s}{\partial y_j} \frac{\partial y_j}{\partial z_i^{(L)}} \\ &= -\frac{t_i(\hat{y})}{y_i} y_i(1 - y_i) - \sum_{j \neq i} \frac{t_j(\hat{y})}{y_j}(-y_j y_i) \\ &= -t_i(\hat{y}) + \sum_j t_j(\hat{y}) y_i \\ &= y_i - t_i(\hat{y}), \end{aligned} \tag{2.43}$$

where we used in the last equation that $t(\hat{y})$ is a one-hot-vector and its components therefore sum to 1.

Now let us consider the derivatives of the general forward computations (2.3) and (2.4) of the neural network. Let $\delta_i^{(\ell)}$ be the partial derivatives of the loss function C_0 with respect to all logits $z_i^{(\ell)}$ in layer ℓ. Then the partial derivative of C_0 with respect to the weight $W_{i,j}^{(\ell)}$ connecting the units $a_j^{(\ell-1)}$ and $z_i^{(\ell)}$ is computed as

$$\frac{\partial C_s}{\partial W_{i,j}^{(\ell)}} = \frac{\partial C_s}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial W_{i,j}^{(\ell)}} = \delta_i^{(\ell)} a_j^{(\ell-1)} \tag{2.44}$$

and similarly the derivative with respect to the bias $b_i^{(\ell)}$ is given by

$$\frac{\partial C_s}{\partial b_i^{(\ell)}} = \frac{\partial C_s}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = \delta_i^{(\ell)}. \tag{2.45}$$

Furthermore, we can compute the partial derivatives with respect to $a_i^{(\ell-1)}$ by

$$\frac{\partial C_s}{\partial a_i^{(\ell-1)}} = \sum_j \frac{\partial C_s}{\partial z_j^{(\ell)}} \frac{\partial z_j^{(\ell)}}{\partial a_i^{(\ell-1)}} = \sum_j \delta_j^{(\ell)} W_{j,i}^{(\ell)}. \tag{2.46}$$

To propagate the error further down through the layers, we can continue to compute the partial derivative with respect to $z_i^{(\ell-1)}$ by applying the chain rule to equation (2.4):

$$\frac{\partial C_s}{\partial z_i^{(\ell-1)}} = \frac{\partial C_s}{\partial a_i^{(\ell-1)}} \frac{\partial a_i^{(\ell-1)}}{\partial z_i^{(\ell-1)}} = \frac{\partial C_s}{\partial a_i^{(\ell-1)}} \phi^{(\ell-1)'}(z_i^{(\ell-1)}) \tag{2.47}$$

This computation is continued through all layers until the partial derivatives for all weights are computed. In Algorithm 2.2 we introduced new notations for convenience. The partial derivatives are denoted by $\mathcal{D}\cdot$:

$$\mathcal{D}W^{(\ell)} \in \mathbb{R}^{n^{(\ell+1)} \times n^{(\ell)}}, \quad \mathcal{D}W_{i,j}^{(\ell)} = \frac{\partial C_s}{\partial W_{i,j}^{(\ell)}}, \quad i = 1, \ldots, n^{(\ell)+1}, \quad j = 1, \ldots, n^{(\ell)} \tag{2.48}$$

$$\mathcal{D}b^{(\ell)} \in \mathbb{R}^{n^{(\ell)}}, \quad \mathcal{D}b_i^{(\ell)} = \frac{\partial C_s}{\partial b_i^{(\ell)}}, \quad i = 1, \ldots, n^{(\ell)} \tag{2.49}$$

$$\mathcal{D}a^{(\ell-1)} \in \mathbb{R}^{n^{(\ell-1)}}, \quad \mathcal{D}a_i^{(\ell-1)} = \frac{\partial C_s}{\partial a_i^{(\ell-1)}}, \quad i = 1, \ldots, n^{(\ell)} \tag{2.50}$$

and $\odot$ denotes the elementwise product.

□

Algorithm 2.2 Backpropagation

Input: Image $\boldsymbol{x}$ with label $\hat{y}$
  $\delta^{(L)} \leftarrow y - t(\hat{y})$
  for $\ell = L, L-1, \ldots, 1$ **do**
    $\mathcal{D}W^{(\ell)} \leftarrow \delta^{(\ell)}(a^{(\ell-1)})^T$
    $\mathcal{D}b^{(\ell)} \leftarrow \delta^{(\ell)}$
    $\mathcal{D}a^{(\ell-1)} \leftarrow (W^{(\ell)})^T \delta^{(\ell)}$
    $\delta^{(\ell-1)} \leftarrow \mathcal{D}a^{(\ell-1)} \odot \phi^{(\ell-1)'}(z^{(\ell-1)})$
  end for
Output: $\nabla_\theta C_0(\boldsymbol{x}, \hat{y}; \theta)$

This method is very efficient since it uses the structure of the neural network and memorizes the error terms in each unit to be used for all lower layers. Its complexity is the same as for a forward pass.

Chapter 3

Convolutional neural networks

The type of neural networks that we considered so far are also called *fully connected* neural networks as each of their units is connected to all units in the previous layer. Now we want to consider another type, the *locally connected* or *convolutional* neural networks (CNNs), which are for example described by LeCun and Bengio [LB+95].

3.1 Basic structure

CNNs have proven to be a good choice for image classification problems (see for example [KSH12, SLJ+15, HZRS15]). One reason for this is that CNNs can be much deeper than fully connected networks while still having the same number of parameters. With more layers, a network is more expressive, i.e. more complex functions can be represented. Furthermore, in images, single pixels do not contain much information. It is the combination of pixels over a larger domain that expresses information about what is present in the image. The convolutions address and exploit this property of images.

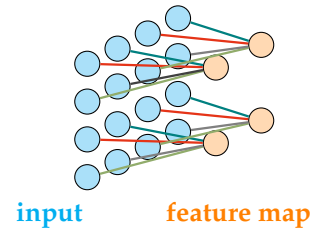


Figure 3.1: Local receptive field with weight sharing.

CNNs consist of two main parts. First, features are extracted from the input data and then these features are fed into a fully connected network for classification. The feature extraction part is built of convolutional layers. CNNs combine several architectures that are very useful in the task of image classification. First, they do not apply fully connected layers to the input image as the number of pixels in a typical image is very large. Consider the case of Krizhevsky [KSH12], where the input images to classify are RGB color images, e.g. arrays of $227 \times 227 \times 3$ pixels, resulting in a total input dimension of 154587. It is not advisable to use every single pixel as individual feature. What is more meaningful are for instance edges, corners and more complex and detailed features such as body shapes when classifying animals for example. Therefore, *local receptive fields* are applied. In a local receptive field each unit gets input from a small patch of units in the previous layer. In Figure 3.1 the scheme of a local receptive field is shown. Each line in the figure corresponds to a number that weights the value of the input unit. All output units (four in this example) are computed with different input units, but the same weights indicated by the line color. We will call the local receptive field represented by its weights a *filter* that is applied to each patch of the image to extract features. This concept of local receptive fields combined with the idea of *weight*

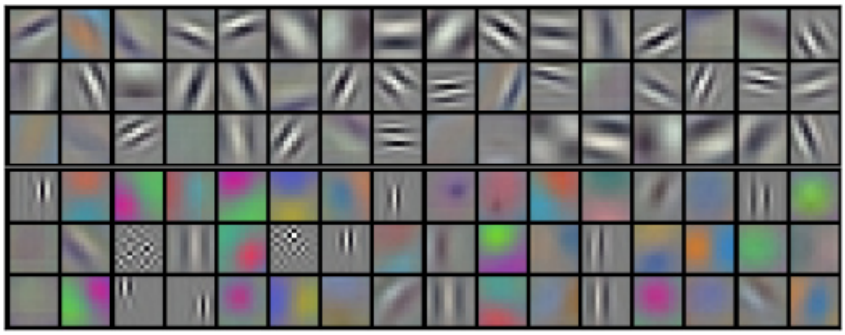


Figure 3.2: Learned filters of the first layer of AlexNet [KSH12].

sharing results in a *convolution* of the filter with the image. Weight sharing is based on the fact that it does not matter where in the image a specific feature occurs [LBBH98]. Thus, a filter representing one feature is shifted over the whole image and computes the activation at every single location. This makes the feature extraction translationally invariant. In Figure 3.1 the filter is of size 2×2 and is applied with a stride of 2, i.e. the patches are nonoverlapping, in order to keep the figure clear. The output of the convolution of the image with a filter is called *feature map*.
A *convolutional layer* consists of multiple filters, each of which produces an individual feature map. In the following layers, these maps are combined to extract features with a higher complexity. Figure 3.2 shows all the filters from the first layer in Krizhevsky's network architecture AlexNet [KSH12] that was trained on the ImageNet classification dataset with 1000 different classes and over 1.2 million training images. We see that these filters are mainly feature extractors for oriented edges, the simplest type of features. Filters that are located deeper in the network extract more detailed features. Thus, the network learns a hierarchical representation of the image which matches the human intuition.

The convolutional layer is followed by the usual nonlinearity, i.e. the activation function described in Section 2.3. In some cases, a so called *pooling layer* is positioned between two convolution-activation-layer pairs. The pooling performs a subsampling of the feature maps which has two benefits. Firstly, the dimension of the data is reduced which is necessary to be able to efficiently classify the input. Secondly, one introduces thereby some scale and translation invariance to the network. The main type of pooling that is used in CNNs is *MaxPooling*, where only the maximal value of a patch is kept. Another possibility is to average the activation functions output over the patch. But as Cohen and Shashua claimed in [CS16], convolutional networks with MaxPooling are universal in contrast to those using average pooling. Universality means that the network can realize any arbitrary function if its size is big enough. Therefore, the MaxPooling is preferable over average pooling.

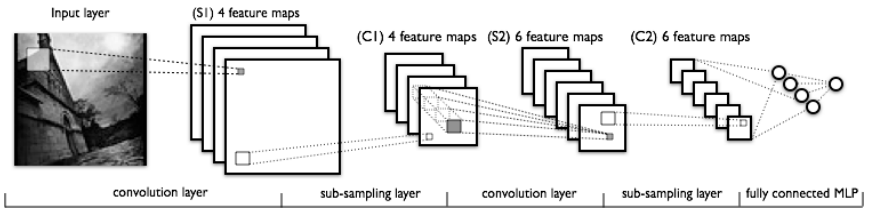


Figure 3.3: Structure of a convolutional neural network. [1]

The number of convolutional layers in a network varies in each architecture (see Section 3.3). The output of the last convolutional layer represents the features which are then used for classification. As shown in Figure 3.3, by the convolutions and poolings the dimension of

[1] http://deeplearning.net/tutorial/lenet.html

the input has been reduced to a sufficient extent so that a small fully connected network with typically between one and three hidden layers [KSH12, SLJ+15, HZRS15] can be applied to predict the class label.

3.2 Mathematical description

As before, we denote the activation of a layer ℓ by $a^{(\ell)}$ and the activation function by $\phi^{(\ell)}(\cdot)$. The variables $w_i^{(\ell)} \in \mathbb{R}^{k_x^{(\ell)} \times k_y^{(\ell)} \times p^{(\ell-1)}}$, $i = 1, \dots, p^{(\ell)}$ are the filters that represent the convolutions in layer ℓ. $k_x^{(\ell)}$ and $k_y^{(\ell)}$ are the kernel sizes of the filters in layer ℓ in x- and y-direction respectively. However, we will consider only quadratic filters here, i.e. we set $k^{(\ell)} := k_x^{(\ell)} = k_y^{(\ell)}$. $p^{(\ell)}$ is the number of channels of the data in layer ℓ, i.e. the input layer has $p^{(0)} = 1$ for grayscale input images and $p^{(0)} = 3$ for colored input images. Furthermore, we denote the first and second dimensions of $a_i^{(\ell)}, s_i^{(\ell)}$ and $z_i^{(\ell)}$ by $n_x^{(\ell)}$ and $n_y^{(\ell)}$, which are the same for all $i = 1, \dots, p$. The pooling function is denoted by pool and $b_i^{(\ell)} \in \mathbb{R}$ is the bias term for all dimensions $i = 1, \dots, p^{(\ell)}$, which is the same scalar value for one feature map. The matrix of dimension $n_x^{(\ell)} \times n_y^{(\ell)}$ with each entry equal to 1 is denoted by $\mathbb{1}_{(n_x^{(\ell)}, n_y^{(\ell)})}$. Then, passing the input image through one layer consisting of convolution, activation and pooling is described by

$$\begin{aligned} z_i^{(\ell)} &= a^{(\ell-1)} * w_i^{(\ell)} + b_i^{(\ell)} \mathbb{1}_{(n_x^{(\ell)}, n_y^{(\ell)})}, \\ s_i^{(\ell)} &= \phi(z_i^{(\ell)}), \qquad\qquad \forall i = 1, \dots, p^{(\ell)} \\ a_i^{(\ell)} &= \mathrm{pool}(s_i^{(\ell)}), \end{aligned} \tag{3.1}$$

where $*$ denotes the convolution of array $a \in \mathbb{R}^{x \times y \times z}$ with filter $w \in \mathbb{R}^{k_x \times k_y \times z}$ which is defined as

$$(a * w)_{u,v} = \sum_{l=-\lfloor k_x/2 \rfloor}^{\lfloor k_x/2 \rfloor} \sum_{m=-\lfloor k_y/2 \rfloor}^{\lfloor k_y/2 \rfloor} \sum_{n=1}^{z} a_{u+\lfloor k_x/2 \rfloor - l, v+\lfloor k_y/2 \rfloor - m, n} w_{l,m,n}. \tag{3.2}$$

Here, $z_i^{(\ell)}$ is the i-th feature map in layer ℓ. When using MaxPooling the pooling unit $\mathrm{pool}(\cdot)$ is defined as

$$\mathrm{pool}(s)_j = \max_{(x,y) \in R_j} s_{x,y} \tag{3.3}$$

and for average pooling it is

$$\mathrm{pool}(s)_j = \frac{1}{|R_j|} \sum_{(x,y) \in R_j} s_{x,y}, \tag{3.4}$$

where in both cases $R_j \subset \{0, \dots, x_s\} \times \{0, \dots, y_s\}$ is the j-th patch of the two dimensional array $s \in \mathbb{R}^{x_s \times y_s}$:

$$R_j = \{jk_x, \dots, (j+1)k_x - 1\} \times \left\{ \left\lfloor \frac{jk_x}{x_s} \right\rfloor, \dots, \left(\left\lfloor \frac{jk_x}{x_s} \right\rfloor + 1 \right) k_y - 1 \right\}. \tag{3.5}$$

By $|R_j|$ we denote the cardinality of R_j, i.e. we have $|R_j| = k_x k_y$.

The complete procedure for a forward pass for an image is shown in Algorithm 3.1. There, L_C is the number of convolutional layers, L_F the number of fully connected layers and t is defined as in Section 2.4. The activation functions $\phi^{(\ell)}$ can vary, except for the last one $\phi^{(L_F)}$ which is in our case the softmax function. Also the pooling function $\mathrm{pool}^{(\ell)}$ is not fixed for every layer. It can either be the MaxPooling, average pooling or – if in one layer no

pooling is applied – the identity function.

Analogously to the function f representing a fully connected network (2.7), a function $f_C : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ for a convolutional network can be defined:

$$f_C(\boldsymbol{x}, \theta) = \sigma(W^{(L_F)}\phi^{(L_F-1)}([\dots]\,\mathrm{pool}^{(L_C)}(\phi^{(L_C)}(\mathrm{pool}^{(L_C-1)}(\dots) * w^{(L_C)} + b^{(L_C)}))) + b_F^{(L_F)}), \tag{3.6}$$

with $\theta = \left(w^{(1)}, \dots, w^{(L_C)}, b^{(1)}, \dots, b^{(L_C)}, W^{(1)}, \dots, W^{(L_F)}, b_F^{(1)}, \dots, b_F^{(L_F)}\right)^\top \in \mathbb{R}^n$. The loss function C_0 for the whole dataset is defined as in (2.33):

$$C_0(\boldsymbol{X}, \hat{Y}; \theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{m} t_j(\hat{y}^i)\log\left(f_C(\boldsymbol{x}^i;\theta)\right)_j. \tag{3.7}$$

The loss on one single example is accordingly defined as

$$C_s(\boldsymbol{x}, \hat{y}; \theta) = -\sum_{j=1}^{m} t_j(\hat{y})\log\left(f_C(\boldsymbol{x};\theta)\right)_j. \tag{3.8}$$

Algorithm 3.1 Forward pass (CNN)

Input: Image $\boldsymbol{x}$ with label $\hat{y}$
  $a^0 \leftarrow \boldsymbol{x}$
  for $\ell = 1, 2, \dots, L_C$ **do**
    for $i = 1, 2, \dots, p^{(\ell)}$ **do**
      $z_i^{(\ell)} \leftarrow a^{(\ell-1)} * w_i^{(\ell)} + b_i^{(\ell)}$
      $s_i^{(\ell)} \leftarrow \phi^{(\ell)}(z_i^{(\ell)})$
      $a_i^{(\ell)} \leftarrow \mathrm{pool}^{(\ell)}(s_i^{(\ell)})$
    end for
    $a^{(\ell)} \leftarrow \mathrm{concat}_{i=1,\dots,p^{(\ell)}}(a_i^{(\ell)})$
  end for
  $a_F^0 \leftarrow \mathrm{vec}(a^{(L_C)})$
  for $\ell = 1, \dots, L_F$ **do**
    $z_F^{(\ell)} \leftarrow W^{(\ell)}a_F^{(\ell-1)} + b_F^{(\ell)}$
    $a_F^{(\ell)} \leftarrow \phi^{(\ell)}(z_F^{(\ell)})$
  end for
  $y \leftarrow a_F^{(L_F)}$
  $c \leftarrow \arg\max_i\ y_i$
  $C \leftarrow -\log y_{\hat{y}}$
Output: Predicted label c and loss $C = C_s(\boldsymbol{x}, \hat{y}; \theta)$

Similar to the Backpropagation Algorithm 2.2 for a standard neural network the gradient of the loss in a CNN with respect to the weights can be computed as stated by LeCun et al. [LBBH98]. Again, we show the gradient computation of C_s for one single example. The gradient of C_0 is then given as the average of the gradients of C_s over all samples. The basic procedure of backpropagation is described by Goodfellow et al. [GBC16]. The special case of backpropagation for CNNs is briefly given in an unpublished technical report by Bouvrie [Bou14]. However, to the best of our knowledge, there is no comprehensive and mathematically rigorous formulation of the backpropagation algorithm for convolutional neural networks. Therefore, we formulated and proved the following theorem.

Theorem 3.1. *Let us consider a convolutional neural network for multi-class classification, represented by a function f_C (3.6), where the last layer is a softmax activation function. Assume that all activation functions $\phi^{(\ell)}$, $\ell = 1, \dots, L$, and all pooling functions $\mathrm{pool}^{(\ell)}$, $\ell = 1, \dots, L$, of the network are differentiable.*

Then, the corresponding loss function C_s on a single sample, given by (3.8)*, is differentiable, and its gradient with respect to the parameters $\theta \in \mathbb{R}^n$ can be computed by the Backpropagation Algorithm 3.2.*

Proof. We assume that the results from the forward pass (Algorithm 3.1) are given. Analogously to (2.48), (2.49) and (2.50), we again denote the partial derivatives of the loss function to an array x by $\mathcal{D}x$, and the partial derivatives with respect to $z^{(\ell)}$ by $\delta^{(\ell)}$.

The beginning of the backpropagation is exactly the same as in Algorithm 2.1 since the last layers of the CNN consist of fully connected layers. It remains to compute the partial derivatives of convolutional and pooling layers. The backpropagation of the error through the pooling function depends on the type of pooling. Intuitively, in case of MaxPooling the unit which contained the maximum value in a patch gets all the error while for all other units the gradient is set to 0, since a small change of these values does not affect the output of the pooling. However, we have to be careful here, since the max-operator is not differentiable and we need to use a smooth approximation to it, like we already did in the ReLUs. Here, we take the maximum over more than two values, so we generalize the version of approximation (2.13) to smooth the max-operator:

$$\max_{i=1,\dots,l} x_i = \lim_{k\to\infty} \frac{1}{k} \log\left(\sum_{i=1}^{l} e^{kx_i}\right) \tag{3.9}$$

Using this differentiable approximation, we can define the derivative for MaxPooling as

$$\mathrm{pool}'(s)_{x,y} = \sum_{j|(x,y)\in R_j} \frac{e^{ks_{x,y}}}{\sum_{(x',y')\in R_j} e^{ks_{x',y'}}} \tag{3.10}$$

for some arbitrary but fixed large $k > 0$. With this smooth approximation to the nondifferentiable max-operator, the function f_C representing the CNN is differentiable as a composition of differentiable functions. Thus, the loss function C_s is differentiable as well, and its gradient with respect to θ is well-defined.

When using average pooling the error is uniformly split into all units since they all have the same contribution:

$$\mathrm{pool}'(s)_{x,y} = \sum_{j|(x,y)\in R_j} \frac{1}{|R_j|}. \tag{3.11}$$

In Algorithm 3.2 we take either (3.10) or (3.11) depending on whether we apply MaxPooling or average pooling respectively whenever we refer to $\mathrm{pool}'(s)$.

Note that, while $\mathrm{pool}(\cdot)$ is a downsampling operator, its derivative $\mathrm{pool}'(\cdot)$ is an upsampling operator.

For deriving the partial derivatives of convolutional layers let us assume that for a convolutional layer ℓ the derivatives of the loss function (3.7) with respect to the activations $\partial C_0/\partial a_i^{(\ell)}$ for all feature maps i are given. In fact, we get these derivatives for the last convolutional layer by applying Algorithm 2.2 to the fully connected part of the CNN. In general, the derivation of the backpropagation looks as follows

$$\frac{\partial C_s}{\partial s_i^{(\ell)}} = \frac{\partial C_s}{\partial a_i^{(\ell)}} \odot \mathrm{pool}^{(\ell)'}(s_i^{(\ell)}). \tag{3.12}$$

Then, the partial derivative with respect to z is the same as before

$$\frac{\partial C_s}{\partial z_i^{(\ell)}} = \frac{\partial C_s}{\partial s_i^{(\ell)}} \odot \phi^{(\ell)'}(z_i^{(\ell)}). \tag{3.13}$$

Now let us, for the moment, omit the subscript i denoting the feature map and instead denote by $z_{x,y}^{(\ell)}$ the unit in row x and column y of one feature map in layer ℓ. Analogously,

$w^{(\ell)}_{x,y}$ shall denote the entry at position (x, y) in the corresponding filter $w^{(\ell)}$. The partial derivatives with respect to the bias are the same as before. However, here each component in one feature map gets the same bias term, so one has to sum over all $z_{x,y}$ to get the derivative:

$$\frac{\partial C_s}{\partial b^{(\ell)}} = \sum_u \sum_v \frac{\partial C_s}{\partial z^{(\ell)}_{u,v}} \frac{\partial z^{(\ell)}_{u,v}}{\partial b^{(\ell)}} = \sum_u \sum_v \delta^{(\ell)}_{u,v}. \tag{3.14}$$

Furthermore, using the definition of the convolution (3.2), we can compute the partial derivative with respect to $w^{(\ell)}_{i_{x,y,z}}$ by

$$\begin{aligned} \frac{\partial C_s}{\partial w^{(\ell)}_{i_{x,y,z}}} &= \sum_u \sum_v \frac{\partial C_s}{\partial z^{(\ell)}_{i_{u,v}}} \frac{\partial z^{(\ell)}_{i_{u,v}}}{\partial w^{(\ell)}_{i_{x,y,z}}} \\ &= \sum_u \sum_v \delta^{(\ell)}_{i_{u,v}} a^{(\ell-1)}_{u+x,v+y,z} \\ &= (a^{(\ell-1)}_{\cdot,\cdot,z} * \delta^{(\ell)})_{x,y}, \end{aligned} \tag{3.15}$$

i.e. taking the derivative with respect to a filter is equivalent to convolving the corresponding feature map with the partial derivative with respect to the activation related to the filter.

Similarly, the partial derivative of the loss function with respect to the activations $a^{(\ell-1)}_{x,y}$ is computed by

$$\begin{aligned} \frac{\partial C_s}{\partial a^{(\ell-1)}_{x,y,z}} &= \sum_i \sum_u \sum_v \frac{\partial C_s}{\partial z^{(\ell)}_{i_{u,v}}} \frac{\partial z^{(\ell)}_{i_{u,v}}}{\partial a^{(\ell-1)}_{x,y,z}} \\ &= \sum_i \sum_u \sum_v \delta^{(\ell)}_{i_{u,v}} w^{(\ell)}_{i_{x-u,y-v,z}} \\ &= \sum_i (\delta^{(\ell)}_i * \text{flipped}(w^{(\ell)}_{i_{\cdot,\cdot,z}}))_{x,y}, \end{aligned} \tag{3.16}$$

where i is the index that represents the individual feature maps and flipped$(w_{i_{\cdot,\cdot,z}})$ denotes the $z-$th channel of filter w after one horizontal and one vertical reflection. The convolution in (3.16) is a *full* convolution, that is, $\delta^{(\ell)}$ is padded with zeroes to be able to apply the filter weights at the border. Hence, $\mathcal{D}a^{(\ell-1)}$ is of higher dimension than $\delta^{(\ell)}$, whereas during the forward pass a *valid* convolution is executed that reduces the dimension of $z^{(\ell)}$ compared to $a^{(\ell-1)}$.

□

We see that the backpropagation in a convolutional neural network again is just a sequential application of convolutions and simple elementwise multiplications and hence of the same complexity as a forward pass.

3.3 Examples of deep convolutional neural networks

We will give a short overview over the most common CNN architectures that evolved in the past years. Not all of them are used in our experiments but we list them here to get an overview of state-of-the-art architectures and their evolution.

LeNet

LeNet was the first successful convolutional neural network that was applied to image classification problems. It was designed in 1998 by Yann LeCun [LBBH98] and was mainly applied on handwritten digit classification. Therefore, the input is an image of size 28×28. The depth of LeNet is still moderate. At that time, the computational power of computers

Algorithm 3.2 Backpropagation (CNN)

Input: Image $\boldsymbol{x}$ with label $\hat{y}$
$\delta^{(L)} \leftarrow y - t(\hat{y})$
for $\ell = L_F, L_F - 1, \ldots, 1$ **do**
$\mathcal{D}W^{(\ell)} \leftarrow \delta^{(\ell)}(a^{(\ell-1)})^\top$
$\mathcal{D}b_F^{(\ell)} \leftarrow \delta^{(\ell)}$
$\mathcal{D}a^{(\ell-1)} \leftarrow (W^{(\ell)})^\top \delta^{(\ell)}$
$\delta^{(\ell-1)} \leftarrow \mathcal{D}a^{(\ell-1)} \odot \phi^{(\ell)'}(z^{(\ell-1)})$
end for
$\mathcal{D}a^{(L_C)} \leftarrow \text{reshape}(\mathcal{D}a^{(0)})$
for $\ell = L_C, L_C - 1, \ldots, 1$ **do**
for $i = 1, 2, \ldots, p^{(\ell)}$ **do**
$\mathcal{D}s_i^{(\ell)} \leftarrow \mathcal{D}a_i^{(\ell)} \odot \text{pool}^{(\ell)'}(s_i^{(\ell)})$
$\delta_i^{(\ell)} \leftarrow \mathcal{D}s_i^{(\ell)} \odot \phi^{(\ell)'}(z_i^{(\ell)})$
$\mathcal{D}w_i^{(\ell)} \leftarrow a^{(\ell-1)} * \delta_i^{(\ell)}$
$\mathcal{D}b_i^{(\ell)} \leftarrow \sum_{x,y} \delta_{i_{x,y}}^{(\ell)}$
end for
$\mathcal{D}a^{(\ell-1)} \leftarrow \sum_i \left(\delta_i^{(\ell)} * \text{flipped}(w_i^{(\ell)})\right)$
end for
Output: $\nabla_\theta C_s(\boldsymbol{x}, \hat{y}; \theta)$

was not sufficient to solve very large problems. The network consists of two convolutional layers each followed by a subsampling layer for feature extraction and two fully connected layers for classification.

AlexNet

An architecture which is closely related to that of LeNet is AlexNet which was introduced in 2012 by Krizhevsky et al. in [KSH12]. With five convolutional layers it is deeper than LeNet. It was designed for the image classification task ILSVRC2012 (ImageNet Large-Scale Visual Recognition Challenge) [RDS+15]. The ImageNet dataset consists of color images of size 227×227×3 from 1000 classes. Using this net architecture, the best error rate from the ILSVRC2010 has been improved by more than 10 percentage points. In total, AlexNet has about 60 million parameters.

VGGNet

The VGGNet was developed by Simonyan and Zissermann (from the 'Visual Geometry Group') [SZ14]. It extends the architectural structure of LeNet and AlexNet to a deeper network (up to 19 layers) by using smaller convolution filters (3×3) to keep the computational cost in a feasible range. In the ILSVRC2014 classification task it reached the second place. With around 138 million this network is one of the architectures with the highest number of parameters.

GoogLeNet

For the ILSVRC2014 Szegedy et al. developed a new CNN architecture that is fundamentally different from the ones before. It is described in [SLJ+15]. The researchers also named it 'Inception' as it captures the idea of the 'Network-in-Network' architecture by Lin et al. [LCY13] to increase the depth and therefore improve the classification. The idea is to use filters of different sizes in a convolutional layer and concatenate the outputs in order to achieve different scales of features. By exploiting the property of dimension reduction of 1×1-convolutions both the depth (by means of layers) and the width (by means of units

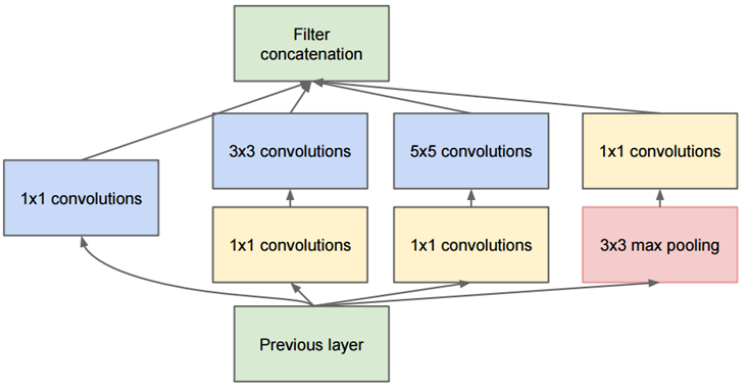


Figure 3.4: Inception Module of GoogLeNet [SLJ+15].

in one layer) of the network can be increased without increasing the computational complexity. Furthermore, the classification is performed by placing only one fully connected layer at the end of the network. Thus, the number of parameters is comparably low, counting less than 10 million. In Figure 3.4 the structure of an inception convolutional layer in GoogLeNet is shown. With this strategy the depth could be increased to 22 layers (including one fully connected layer). The GoogLeNet achieved the best classification performance in the ILSVRC2014.

ResNet

Another type of CNN named ResNet was presented by He et al. in [HZRS15]. In the ILSVRC2015 classification task it won the first place. There are different instances of ResNet, one of which has more than 1000 layers which is over 50 times deeper than VGGNet. De-

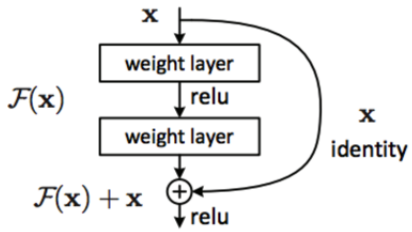


Figure 3.5: Building block of ResNet [HZRS15].

spite its great depth, ResNet still has lower complexity than VGG. This great reduction of parameters is achieved as in GoogLeNet by canceling the fully connected layers between the convolutional layers and the classification layer. The name ResNet comes from the newly introduced idea of residual learning. In order to avoid the phenomenon of vanishing gradients, shortcut connections are used. These shortcut connections (in this case the identity mapping) skip one or more layers and add their output to the output from the usual convolutional layers (see Figure 3.5).

Chapter 4

Fast curvature matrix-vector products for deep neural networks

The second order type methods that we are going to discuss need to evaluate the product of the Hessian matrix $\mathbf{H}(\theta) \in \mathbb{R}^{n\times n}$ or the Gauss-Newton matrix $\mathbf{G}(\theta) \in \mathbb{R}^{n\times n}$ with an arbitrary vector $v \in \mathbb{R}^n$. Because of memory limitations, it is infeasible to store the whole matrix. Fortunately, there is a way to compute curvature matrix-vector products with a cost of $\mathcal{O}(n)$ in both time and memory.

4.1 Exact multiplication by the Hessian matrix

The procedure of computing $\mathbf{H}(\theta)v$ is known as the "Pearlmutter trick" as it was first proposed by Barak Pearlmutter in 1994 [Pea94]. It uses the layerwise structure of the neural network, similar to the backpropagation algorithm. It is thus a further application of the automatic differentiation techniques. The idea is to compute the product $\mathbf{H}(\theta)v$ as the directional derivative of the gradient of the loss function in direction v. Let $C : \mathbb{R}^n \to \mathbb{R}$ denote the loss function. Assume that it is differentiable and its gradient is Gâteaux differentiable. Then the Hessian-vector product is given as

$$\mathbf{H}(\theta)v = \lim_{\epsilon\to 0} \frac{\nabla C(\theta + \epsilon v) - \nabla C(\theta)}{\epsilon}. \tag{4.1}$$

Indeed, as the max-operators can be replaced by smooth approximations, the basic loss functions for a neural network (2.33) and a convolutional neural network (3.7) respectively are continuously differentiable since they are composed of continuously differentiable functions. Hence, the directional derivative of their gradients exists for every direction $v \in \mathbb{R}^n$.

Pearlmutter introduced the $\mathcal{R}_v$-operator that is defined on the set

$$\mathcal{F}_v^{n,q} = \{x : \mathbb{R}^n \to \mathbb{R}^q \mid x \text{ is Gâteaux differentiable in direction } v\}$$

of Gâteaux differentiable functions in direction $v \in \mathbb{R}^n$. The operator $\mathcal{R}_v$ applied to $x \in \mathcal{F}_v^{n,q}$ is just the directional derivative of x in direction v:

$$(\mathcal{R}_v x)(\theta) = \lim_{\epsilon\to 0} \frac{x(\theta + \epsilon v) - x(\theta)}{\epsilon}. \tag{4.2}$$

Hence, $\mathbf{H}(\theta)v$ can be written as the $\mathcal{R}_v$-operator applied to the gradient $\nabla C \in \mathcal{F}_v^{n,1}$ and evaluated at θ:

$$\mathbf{H}(\theta)v = (\mathcal{R}_v \nabla C)(\theta). \tag{4.3}$$

We will drop the subscript v for convenience in the following considerations. Since $\mathcal{R}$ is a differential operator, the usual rules for derivatives hold. Let $x, y \in \mathcal{F}_v^{n,q}$ and $\gamma \in \mathcal{F}_v^{q,p}$. Then the following rules hold:

$$\begin{aligned}
\mathcal{R}(x + y) &= \mathcal{R}x + \mathcal{R}y, && \text{(sum rule)}\\
\mathcal{R}(xy) &= (\mathcal{R}x)y + x(\mathcal{R}y), && \text{(product rule)}\\
\mathcal{R}(\gamma(x)) &= (\mathcal{R}x)\mathbf{J}_\gamma(x), && \text{(chain rule)}
\end{aligned}$$

where $\mathbf{J}_\gamma$ denotes the Jacobian of γ. We can just apply this $\mathcal{R}$-operator to all computations in the forward and backward pass to get the desired output $(\mathcal{R}\nabla C)(\theta)$ as it is done in [Mar16]. Since $\nabla C \in \mathcal{F}_v^{n,1}$ for every $v \in \mathbb{R}^n$, this procedure is well-defined. The complete computation is given in Algorithm 4.1 where the parameters θ are split into the weight matrices $W^{(\ell)}$ and biases $b^{(\ell)}$ for $\ell = 1, \dots, L$. The function $C_r : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ denotes the last part of the loss function C. It gets as input the values of the forward pass before the softmax activation is applied, i.e. the output of f_r, where the network function f (2.7) is decomposed into $f = \sigma \circ f_r$. We have

$$C_r(z, \hat{y}) = -\sum_{j=1}^{m} t_j(\hat{y}) \log\left(\frac{\exp(z_j)}{\sum_i \exp(z_i)}\right). \tag{4.4}$$

Note, that the last activation function – the softmax – is included in the function C_r, so technically, in Algorithm 4.1 we have that the last activation function $\phi^{(L)}$ is the identity function.

The initialization of $\mathcal{R}\mathcal{D}a^{(L)}$ is derived by

$$\begin{aligned} \mathcal{R}\mathcal{D}a^{(L)} &= \mathcal{R}\left(\left.\frac{\partial C_r(z, \hat{y})}{\partial z}\right|_{z=z^{(L)}}\right) = \\ &= \left.\frac{\partial^2 C_r(z, \hat{y})}{\partial z^2}\right|_{z=z^{(L)}} \mathcal{R}z^{(L)} \\ &= \nabla^2_{zz} C_r(z^{(L)}, \hat{y}) \mathcal{R}z^{(L)}. \end{aligned} \tag{4.5}$$

It remains to compute the Hessian $\nabla^2_{zz} C_r(z^{(L)}, \hat{y}) \in \mathbb{R}^{m \times m}$. In Section 2.5 we already computed the first derivative of $C_r(z, \hat{y})$ with respect to z:

$$\frac{\partial C(z, \hat{y})}{\partial z_k} = \sigma_k(z) - t_k(\hat{y}). \tag{4.6}$$

Hence, we obtain for the second derivatives

$$\frac{\partial^2 C(z, \hat{y})}{\partial z_k^2} = \frac{\partial \sigma_k(z)}{\partial z_k} = \sigma_k(z) - \sigma_k(z)^2, \tag{4.7}$$

$$\frac{\partial^2 C(z, \hat{y})}{\partial z_k \partial z_j} = \frac{\partial \sigma_k(z)}{\partial z_j} = -\sigma_k(z)\sigma_j(z) \tag{4.8}$$

for $j \neq k$. Thus, we can write the Hessian matrix of $C_r(z, \hat{y})$ as

$$\nabla^2_{zz} C_r(z, \hat{y}) = \operatorname{diag}(\sigma(z)) - \sigma(z)\sigma(z)^\top. \tag{4.9}$$

For convolutional neural networks, the application of the $\mathcal{R}$-operator may seem to be more involved as it is not immediately clear how to work out $\mathcal{R}(a * w)$ for an array $a \in \mathbb{R}^{x \times y \times z}$ and a convolution filter $w \in \mathbb{R}^{k_x \times k_y \times z}$. However, the convolution is just a linear function and can be replaced by a matrix-vector-multiplication $\mathbf{A}^\top W$. To get the matrix $\mathbf{A} \in \mathbb{R}^{k_x k_y z \times q}$, where q is the spatial dimension of the matrix $a * w$, for each entry $a_{x,y}$ in each channel of a the patch of size $k_x \times k_y$ around $a_{x,y}$ is written in one column of $\mathbf{A}$, with the channels for each patch stacked in the same column. Analogously, the vector $W \in \mathbb{R}^{k_x k_y z}$ is formed by stacking all elements of w in the according order. Then, by computing $Z = \mathbf{A}^\top W$ the filter weights are exactly applied to the correct entries of a for each possible patch and the output $Z \in \mathbb{R}^q$ is a vectorized form of $a * w$ which can then be reshaped for further processing. Hence, the computation of the Hessian-vector product $\mathbf{H}v$ for a vector $v \in \mathbb{R}^n$ for a CNN is completely analogous to the one described in Algorithm 4.1 for a fully connected neural network.

Algorithm 4.1 Computation of $\mathbf{H}(\theta)v$ in a feed-forward neural network

Input: Image $\boldsymbol{x}$ with label $\hat{y}$ and direction v mapped to $(\mathcal{R}W^{(\ell)}, \mathcal{R}b^{(\ell)})_{\ell=1,\dots,L}$
$a^{(0)} \leftarrow \boldsymbol{x}$
$\mathcal{R}a^{(0)} \leftarrow 0$
for $\ell = 1, 2, \dots, L$ **do**
$\mathcal{R}z^{(\ell)} \leftarrow (\mathcal{R}W^{(\ell)})a^{(\ell-1)} + W^{(\ell)}\mathcal{R}a^{(\ell-1)} + \mathcal{R}b^{(\ell)}$
$\mathcal{R}a^{(\ell)} \leftarrow (\mathcal{R}z^{(\ell)}) \odot \phi^{(\ell)'}(z^{(\ell)})$
end for
$\mathcal{R}\mathcal{D}a^{(L)} \leftarrow \nabla^2_{zz} C_r(z^{(L)}, \hat{y})\mathcal{R}a^{(L)}$
for $\ell = L, \dots, 1$ **do**
$\mathcal{R}\delta^{(\ell)} \leftarrow (\mathcal{R}\mathcal{D}a^{(\ell)}) \odot \phi^{(\ell)'}(z^{(\ell)}) + \mathcal{D}a^{(\ell)} \odot \phi^{(\ell)''}(z^{(\ell)}) \odot \mathcal{R}z^{(\ell-1)}$
$\mathcal{R}\mathcal{D}W^{(\ell)} \leftarrow (\mathcal{R}\delta^{(\ell)})(a^{(\ell-1)})^\top + \delta^{(\ell)}\mathcal{R}(a^{(\ell-1)})^\top$
$\mathcal{R}\mathcal{D}b^{(\ell)} \leftarrow \mathcal{R}\delta^{(\ell)}$
$\mathcal{R}\mathcal{D}a^{(\ell-1)} \leftarrow (\mathcal{R}(W^{(\ell)})^\top)\delta^{(\ell)} + (W^{(\ell)})^\top\mathcal{R}\delta^{(\ell)}$
end for
Output: $\mathbf{H}v$ mapped from $(\mathcal{R}\mathcal{D}W^{(\ell)}, \mathcal{R}\mathcal{D}b^{(\ell)})_{\ell=1,\dots,L}$

4.2 Exact multiplication by the Gauss-Newton matrix

The Hessian matrix of the objective function $C : \mathbb{R}^n \to \mathbb{R}$ has one big drawback. It is not necessarily positive semidefinite because C is not convex. However, the later discussed Hessian-free method expects the Hessian to be positive semidefinite as the convergence of the conjugate gradient algorithm relies on this fact. To overcome this problem the Hessian matrix can be approximated by the Gauss-Newton matrix.

The Gauss-Newton matrix

The Gauss-Newton matrix is also referred to as the outer product approximation of the Hessian. Let us first define the Gauss-Newton matrix following [Sch02].

We again decompose the function f representing the network into $f = \sigma \circ f_r$, with σ being the softmax activation function and $f_r : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ mapping the input $\boldsymbol{x}$ and the parameters θ to the logits $z^{(L)}$ of the last layer. We decompose the loss function C into $C = C_r \circ f_r$, with $C_r : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ mapping the logits $z^{(L)}$ and the ground truth $\hat{y}$ to the cross entropy loss.

We denote the Jacobian of C_r with respect to the logits $z^{(L)}$ by $\mathbf{J}_{C_r}$, the Hessian of the same function accordingly by $\mathbf{H}_{C_r}$ and analogously we denote the Jacobian of the function f_r with respect to the parameters θ by $\mathbf{J}_{f_r}$. Then, we can define the Hessian matrix $\mathbf{H}$ of the loss function C with respect to the parameters θ by

$$\mathbf{H} = \frac{d}{d\theta}(\mathbf{J}_{C_r}\mathbf{J}_{f_r}), \tag{4.10}$$

which leads to

$$\mathbf{H} = \mathbf{J}_{f_r}^\top \mathbf{H}_{C_r}\mathbf{J}_{f_r} + \sum_{i=1}^{m} \left(\mathbf{J}_{C_r}\right)_i \mathbf{H}_{(f_r)_i}, \tag{4.11}$$

where i ranges over all output units and $(f_r)_i$ denotes the function that produces the $i-$th output of f_r. The first term of (4.11) includes the second order information of the loss function depending on the network's output while the second term expresses the second order information of the network itself without the loss. Using (4.11) we define the Gauss-Newton matrix $\mathbf{G}$ by ignoring the second term and get

$$\mathbf{G} \coloneqq \mathbf{J}_{f_r}^\top \mathbf{H}_{C_r}\mathbf{J}_{f_r}. \tag{4.12}$$

From this definition it is immediately clear that $\mathbf{G}$ approximates $\mathbf{H}$ in the sense that it consists of one summand of the formula (4.11) for computing the Hessian matrix. Its most important

property is, that it is always positive semidefinite as long as the Hessian $\mathbf{H}_{C_r}$ of the loss function combined with the activation function of the last layer with respect to the logits is positive semidefinite. We have that

$$\mathbf{H}_{C_r} = \nabla^2_{zz} C_r(z, \hat{y}) \tag{4.13}$$
$$= \operatorname{diag}(\sigma(z)) - \sigma(z)\sigma(z)^\top \tag{4.14}$$

as we derived in Section 4.1. We will show that $\nabla^2_{zz} C_r(z, \hat{y})$ is positive semidefinite. This essential property then is inherited by the Gauss-Newton matrix. On that account we formulate and prove the following lemma and corollary.

Lemma 4.1. *The Hessian matrix $\nabla^2_{zz} C_r(z, \hat{y}) \in \mathbb{R}^{m \times m}$ of the function C_r defined in* (4.4) *is positive semidefinite.*

Proof. We have

$$\nabla^2_{zz} C_r(z, \hat{y}) = \operatorname{diag}(\sigma(z)) - \sigma(z)\sigma(z)^\top. \tag{4.15}$$

Let us substitute $\sigma(z)$ by y. By definition of σ as softmax function we have that $0 < y_i < 1$, for all $i = 1, \ldots, m$, and $\sum_{i=1}^m y_i = 1$. Then, we have for any $p \in \mathbb{R}^m \backslash \{0\}$:

$$\begin{aligned} p^\top \left(\operatorname{diag}(y) - yy^\top\right) p &= \sum_{i=1}^m y_i p_i^2 - \left(\sum_{i=1}^m y_i p_i\right)^2 \\ &= \sum_{i=1}^m y_i p_i^2 - 2\left(\sum_{i=1}^m y_i p_i\right)\left(\sum_{j=1}^m y_j p_j\right) + \left(\sum_{i=1}^m y_i p_i\right)^2 \\ &= \sum_{i=1}^m y_i \left(p_i - \sum_{j=1}^m y_j p_j\right)^2 \geq 0. \end{aligned} \tag{4.16}$$

□

Corollary 4.1. *Let $f : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ be the network function decomposed into $f(\boldsymbol{x}; \theta) = \sigma(f_r(\boldsymbol{x}; \theta))$, and let $C : \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ be the objective function decomposed into $C(\boldsymbol{x}, \hat{y}; \theta) = C_r(f_r(\boldsymbol{x}; \theta), \hat{y})$. Let $\boldsymbol{x} \in \mathbb{R}^d$, $\hat{y} \in \mathbb{R}$ and $\theta \in \mathbb{R}^n$. Then the Gauss-Newton matrix*

$$\boldsymbol{G}(\boldsymbol{x}, \hat{y}; \theta) = \boldsymbol{J}^\top_{f_r}(\boldsymbol{x}; \theta) \boldsymbol{H}_{C_r}(f_r(\boldsymbol{x}; \theta), \hat{y}) \boldsymbol{J}_{f_r}(\boldsymbol{x}; \theta) \tag{4.17}$$

$\in \mathbb{R}^{m \times m}$ of the objective function C, is positive semidefinite.

Proof. We have $f_r : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ and thus $\mathbf{J}_{f_r}(\boldsymbol{x}; \theta) \in \mathbb{R}^{m \times n}$.

Let $p \in \mathbb{R}^m \backslash \{0\}$. Then,

$$p^\top \mathbf{G}(\boldsymbol{x}, \hat{y}; \theta) p = p^\top \mathbf{J}^\top_{f_r}(\boldsymbol{x}; \theta) \mathbf{H}_{C_r}(f_r(\boldsymbol{x}; \theta), \hat{y}) \mathbf{J}_{f_r}(\boldsymbol{x}; \theta) p \tag{4.18}$$
$$= q^\top \nabla^2_{zz} C_r(z, \hat{y}) \Big|_{z = f(\boldsymbol{x}; \theta)} q \tag{4.19}$$
$$= q^\top \left(\operatorname{diag}(\sigma(f_r(\boldsymbol{x}; \theta))) - \sigma(f_r(\boldsymbol{x}; \theta))\sigma(f_r(\boldsymbol{x}; \theta))^\top\right) q \tag{4.20}$$
$$\geq 0, \tag{4.21}$$

where we set $q = \mathbf{J}_{f_r}(\boldsymbol{x}; \theta) p,\ q \in \mathbb{R}^m$ and used Lemma 4.1. □

Exact multiplication by the Gauss-Newton matrix

In the following we will assume the data $(\boldsymbol{x}, \hat{y})$ to be fixed and denote the Gauss-Newton matrix by $\mathbf{G}(\theta)$.

Since $\mathbf{G}(\theta) \in \mathbb{R}^{n \times n}$ is too big to be stored in memory, just like the Hessian, we also need an efficient algorithm to compute the Gauss-Newton matrix-vector product $\mathbf{G}(\theta)v$ for some $v \in \mathbb{R}^n$. We can write this product as

$$\mathbf{G}(\theta)v = \mathbf{J}_{f_r}(\theta)^\top \nabla^2_{zz} C_r(f_r(\boldsymbol{x};\theta), \hat{y}) \mathbf{J}_{f_r}(\theta)v. \tag{4.22}$$

As we already mentioned, this term is also part of the Hessian-vector product and thus, the Gauss-Newton-vector product can be computed by a modified version of Algorithm 4.1 [Mar16]. The Gauss-Newton matrix does not include the terms of second derivatives within the network function f_r. These terms occur only in the backward pass of Algorithm 4.1. Therefore, the forward pass is identical in both algorithms. The quantities that are computed in the forward pass (i.e. the $z^{(\ell)}$'s and $a^{(\ell)}$'s) are treated as constants and thus the $\mathcal{R}$-operator applied to those variables give no contribution to the product $\mathbf{G}(\theta)v$. Hence, the algorithm for computing $\mathbf{G}(\theta)v$ is very similar to that for computing $\mathbf{H}(\theta)v$. It is given in Algorithm 4.2. The Hessian matrix $\nabla^2_{zz} C_r(z^{(L)}, \hat{y})$ is given as in (4.9).

Algorithm 4.2 Computation of $\mathbf{G}(\theta)v$ in a feed-forward neural network

Input: Image $\boldsymbol{x}$ with label $\hat{y}$ and direction v mapped to $(\mathcal{R}W^{(\ell)}, \mathcal{R}b^{(\ell)})_{\ell=1,\dots,L}$
  $a^{(0)} \leftarrow \boldsymbol{x}$
  $\mathcal{R}a^{(0)} \leftarrow 0$
  for $\ell = 1, 2, \dots, L$ **do**
    $\mathcal{R}z^{(\ell)} \leftarrow (\mathcal{R}W^{(\ell)})a^{(\ell-1)} + W^{(\ell)}\mathcal{R}a^{(\ell-1)} + \mathcal{R}b^{(\ell)}$
    $\mathcal{R}a^{(\ell)} \leftarrow (\mathcal{R}z^{(\ell)}) \odot \phi^{(\ell)'}(z^{(\ell)})$
  end for
  $\mathcal{RD}a^{(L)} \leftarrow \nabla^2_{zz} C_r(z^{(L)}, \hat{y})\mathcal{R}a^{(L)}$
  for $\ell = L, \dots, 1$ **do**
    $\mathcal{R}\delta^{(\ell)} \leftarrow (\mathcal{RD}a^{(\ell)}) \odot \phi^{(\ell)'}(z^{(\ell)})$
    $\mathcal{RD}W^{(\ell)} \leftarrow (\mathcal{R}\delta^{(\ell)})(a^{(\ell-1)})^\top$
    $\mathcal{RD}b^{(\ell)} \leftarrow \mathcal{R}\delta^{(\ell)}$
    $\mathcal{RD}a^{(\ell-1)} \leftarrow (W^{(\ell)})^\top \mathcal{R}\delta^{(\ell)}$
  end for
Output: $\mathbf{G}(\theta)v$ mapped from $(\mathcal{RD}W^{(\ell)}, \mathcal{RD}b^{(\ell)})_{\ell=1,\dots,L}$

Chapter 5

Training of convolutional neural networks

The training of neural networks has many challenges. In this chapter we present some tools to produce improved results and give the basic mathematical background of the optimization.

5.1 Regularization

With a higher number of parameters, i.e. more units and layers in the net, the function f has more degrees of freedom and is thus more likely to classify the training set correctly. But the ultimate goal is not only to classify the training images correctly. We rather want the neural net to generalize well and thus be able to classify new images correctly that are not present in the training set. The process of adapting well to the training set but having poor performance on new data is called *overfitting* [Die95]. To avoid overfitting to the training data, a so-called regularization of the network is applied. There are several types of regularization which we will describe here. Some of them slightly modify the loss function derived in Section 2.4.

Weight decay

A well-known heuristic for modeling a system is referred to as *Occam's razor* [BEHW87]. It states that a model should only be as complex as necessary and as simple as possible. One way to simplify the neural network that models the target function is to keep the weights small. This can be performed by adding a penalty term to the loss function $C_0(\theta)$. The most frequently used penalty is the ℓ_2-norm of the weights:

$$C(\boldsymbol{X}, \hat{Y}; \theta) = C_0(\boldsymbol{X}, \hat{Y}; \theta) + \frac{\lambda}{2N}\|\theta\|_2^2 \tag{5.1}$$

where $\lambda > 0$ is a parameter that controls the impact of the regularization term. In the minimization problem (2.35) the objective function $C_0(\boldsymbol{X}, \hat{Y}; \theta)$ is replaced by its regularized version $C(\boldsymbol{X}, \hat{Y}; \theta)$. Typically only large weights of connected units are penalized, not the bias parameters. In the backpropagation algorithm this penalty term obviously has to be included in the computation of the derivative of the weight parameters. For all $i = 1, \ldots, n$ we have

$$\frac{\partial C}{\partial \theta_i} = \frac{\partial C_0}{\partial \theta_i} + \frac{\lambda}{N}\theta_i. \tag{5.2}$$

In [KH91] Krogh and Hertz give a theoretical proof that weight decay helps in reducing the generalization error. Krizhevsky et al. [KSH12] also found that adding a small weight decay term even reduced the training error in their experiments.

Instead of using the ℓ_2-norm as penalty, another possibility is to use the ℓ_1-norm. This type of regularization leads to sparse weights, i.e. the neurons use only a sparse subset of their inputs and therefore are less sensitive to noise.

Another form of weight decay can be achieved by adding constraints to the optimization problem which force the weights to have an absolute value smaller than some upper bound.

Model averaging

In model averaging a number of neural networks is trained and the predicted outputs for the same input data are combined to compute the final prediction. While this approach almost always reduces the error on the test set [SHK+14] it has a major drawback. Typically, it is very time consuming to both train a deep neural network and pass the data through many networks.

Dropout

The concept of Dropout was introduced by Hinton et al. [HSK+12]. It only applies to fully connected layers. In each iteration each hidden neuron is randomly omitted with a probability γ, which is typically set to 0.5. This prevents the adaptation of neurons on each other since one neuron cannot rely on all neurons to be present. Thus, each neuron learns to detect features on its own that are helpful in classifying the given data. This makes the prediction more robust. In the test phase all neurons are used but their activations are multiplied by a factor of $1 - \gamma$.

Another way of interpreting Dropout is the inherent application of model averaging as pointed out in [HSK+12]. By applying Dropout during the training of a network, in each iteration another network architecture is present which is basically the same as training different models.

DropConnect

A generalization of Dropout is called DropConnect and was introduced by Wan et al. in 2013 [WZZ+13]. Just like Dropout, this technique is only applied to the fully connected layers of the network. Whereas Dropout sets whole neurons to zero and thus omits all connections associated with that neuron, DropConnect only sets randomly chosen outgoing activations from neurons to zero. A fully connected layer thereby becomes a sparsely connected layer with randomly varying connections during training. Whether an activation is set to zero is individually drawn from a Bernoulli distribution for each sample. In [WZZ+13] it is shown both theoretically and empirically that DropConnect helps to prevent overfitting of neural networks.

Data augmentation

Overfitting occurs, as mentioned earlier, typically in overly complex models. By providing more training data, the mismatch of given information and model complexity represented by the number of parameters can be tackled. While it can be difficult to get more real images (which additionally have to be labeled before being used as training data), it is possible to augment the dataset artificially. The input images can be modified in such a way that they still represent the same class. In [KSH12] for example horizontal reflections, random crops and translations are applied to the original images. Other possibilities include scaling, change of illumination or rotation. Generating new data by these modifications thus provides more generalized class information and therefore leads to a smaller generalization error.

Early stopping

During training, after a certain number of iterations, the current parameters are tested by evaluating the accuracy that is achieved on the test dataset which is distinct from the training dataset. When it occurs that this accuracy is not improved over several iterations or

even drops again, it is very likely that further training does not achieve any more improvements and the state of overfitting is reached. Then the training can be stopped and the set of parameters with the best accuracy on the test set is used as final result.

DisturbLabel

Xie et al. propose in [XWW+16] a new attempt to reduce overfitting which achieves competitive results to image classification benchmarks. The idea is to add noise to the loss layer which then is backpropagated through the network by replacing some labels by incorrect ones in each iteration. For every sample a random class label is chosen with a given probability. This amounts to training a neural network with many noisy datasets which is in some sense the dual approach of model ensemble compared to Dropout regularization, where many different net architectures are trained with one dataset. Both of these methods can be combined to produce even better results [XWW+16].

5.2 Strategies for improved performance

There are some strategies that are not specific to the optimization method used for training, but help to accelerate the training and produce better results. One problem of gradient methods in the context of deep learning is the fact that usually small learning rates are required to ensure convergence. This stems partly from the so called *internal covariate shift* [IS15] which refers to the fact that during training the input to each layer changes as parameters in one layer are affected by all parameters of previous layers. Thus, even small changes amplify with increasing depth of the network. The optimization method has to compensate these changes by a small learning rate. Furthermore, when using saturating nonlinearities as activation functions (hyperbolic tangent, sigmoid) it is likely for many dimensions to reach the saturated region as also this effect is amplified through the network's depth. This, in turn, has the consequence that gradients almost vanish and thus the training is slowed down. While the vanishing gradients can be handled by using ReLUs instead, the internal covariate shift still causes the need for small learning rates. Therefore a strategy that has been widely used already for quite a long time is to precondition the input [LBOM12]. The data is normalized by first shifting it to have zero mean. Then it is ideally decorrelated by applying PCA (Principle Component Analysis) and finally scaled so that all components have the same covariance.

A further stage of this strategy is to normalize not only the original input data but the inputs to all layers. Ioffe and Szegedy proposed the *batch normalization* [IS15] where each feature map is whitened, i.e. shifted and scaled to zero mean and covariance 1. To keep the representational capacity of the network, these transformed feature maps undergo an affine transformation to be able to represent the identity function. The batch normalization also introduces regularization to the network. Ioffe and Szegedy [IS15] reduced other regularization types in their experiments. By applying batch normalization, a five times bigger learning rate could be used. They achieved a speed-up of training the ImageNet dataset with GoogLeNet by up to a factor 15 in terms of iterations to reach the same accuracy on the test set compared to the same network without batch normalization.

Krizhevsky et al. [KSH12] used a different type of normalization, the *local response normalization*. Here, the activations at every spatial position are normalized over a bunch of feature maps. This is in contrast to the batch normalization where the activations are normalized using the information of only one feature map. With the local response normalization Krizhevsky et al. reached an improvement of about 2% test error rate.

A method which demonstrated an improvement of the generalization properties of networks is *unsupervised pretraining*. It was proposed in [HS06] for Restricted Boltzmann Machines. By random initialization of the weights, the training often converged to a bad local minimum. This is mainly due to the saturation property of the nonlinearities. With the pretraining, a set of weights can be found that is already close to a good solution. However,

with the recent innovations of rectifiers as nonlinearity and new regularization techniques such as Dropout, the need for pretraining almost vanished. Even without pretraining top results were achieved in the ILSVRC challenges.

5.3 Problem formulation

The training of CNNs is performed by minimizing the error function. Let us denote the error function on the i-th training sample by $C(\boldsymbol{x}^i, \hat{y}^i; \theta)$. This is the cross entropy loss C_s for convolutional neural networks (3.8) on example i and $\theta \in \mathbb{R}^n$, which is optionally combined with a regularization modifying the loss function like weight decay resulting in

$$C(\boldsymbol{x}^i, \hat{y}^i; \theta) = -\sum_{j=1}^{m} t_j(\hat{y}^i) \log \left(f_C(\boldsymbol{x}^i; \theta)\right)_j + \lambda \|\theta\|_2^2 \tag{5.3}$$

with $t_j(\hat{y}^i) = 1$ for $\hat{y}^i = j$ and $t_j(\hat{y}^i) = 0$ else. The loss calculated on all training tuples $(\boldsymbol{X}, \hat{Y}) = (\boldsymbol{x}^i, \hat{y}^i)_{i \in [N]}$ is the empirical risk function

$$E_N(\boldsymbol{X}, \hat{Y}; \theta) = \frac{1}{N} \sum_{i=1}^{N} C(\boldsymbol{x}^i, \hat{y}^i; \theta). \tag{5.4}$$

Thus, by defining the loss on the whole dataset as

$$C(\boldsymbol{X}, \hat{Y}; \theta) = \frac{1}{N} \sum_{i=1}^{N} C(\boldsymbol{x}^i, \hat{y}^i; \theta) \tag{5.5}$$

we have $E_N(\boldsymbol{X}, \hat{Y}; \theta) = C(\boldsymbol{X}, \hat{Y}; \theta)$. It approximates the expected risk

$$E_p(\theta) = \int C(\boldsymbol{x}, \hat{y}; \theta) dp_{data}(\boldsymbol{x}, y) \tag{5.6}$$

which measures the generalization performance of the classification. According to statistical learning theory it is sufficient to minimize the empirical risk instead of the expected risk [Bot10]. Thus, the goal is to find a set of parameters θ^* that minimizes the expected risk:

$$\theta^* = \arg\min_{\theta} \; E_N(\boldsymbol{X}, \hat{Y}; \theta). \tag{5.7}$$

Related to this minimization task, we face three major problems. First, the dimension n of θ is very high, depending on the neural network it can be in the order of 10^6. Hence, we need optimization methods that can handle such large-scale optimization problems. In particular, the storage requirements of the algorithm cannot be too high. This hampers the usage of second order methods as these usually require the Hessian matrix of dimension $n \times n$ which is too expensive to compute and to store.

Another problem of solving (5.7) is that also the number of training samples N typically is very large (e.g. the ImageNet dataset contains 1.2 million examples). Of course, many training examples are necessary to be able to learn representative features and achieve good classification results. However, the evaluation of $E_N(\theta)$ requires N data passes through the network which consumes a lot of time. It is therefore too expensive to evaluate the loss function and its gradient on the whole dataset at each iteration. To overcome this complication, one can exploit the redundancy of the dataset. Due to the large amount of samples, not each one introduces new information about features. Therefore, a randomly drawn subset of all samples can be used to approximate the true loss and gradient by only evaluating these functions on this so-called *minibatch*.

The third problem is the fact that due to the nesting of convolutions, poolings and nonlinearities the objective function is not convex. The theoretical convergence properties of

algorithms that are mostly shown for convex or even strongly convex functions do not hold and it is possible for the algorithms to run and get stuck in a local minimum. Haeffele and Vidal [HV15] show that a local optimum is a global optimum if certain conditions are fulfilled and thus, from any initialization a global optimum can be found. However, we do not go into detail of this theory and focus on the mere optimization methods for finding any optimum.

The whole labeled dataset that is available is split into two parts: the training data and the test dataset. The training dataset usually contains much more images than the test data. As the names indicate, the training dataset is the dataset that is used for training, i.e. the objective function in the minimization problem (5.7) is based on this dataset. The test dataset is used for evaluating the performance of the neural network. It is the ultimate goal to reduce the error on the test dataset. Thus, the actual data where we want to have a correct classification is not even part of the objective function that we minimize. This exacerbates the problem of deep learning. By classifying the images in the test dataset it can be checked whether the algorithm overfitted the training data. This information can then be used to tweak the hyperparameters for a solution that generalizes well.

5.4 Standard optimization methods

In this section we present the most popular optimization methods in deep learning. All of them are first order methods and rely on the gradient descent algorithm which is an iterative method. We will denote the value of variables at time step t by the subscript t. For the basic gradient descent the update rule for the weights looks as follows:

$$\begin{aligned}\theta_{t+1} &= \theta_t - \alpha_t \nabla_\theta E_N(\boldsymbol{X}, \hat{Y}; \theta_t) \\ &= \theta_t - \alpha_t \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta C(\boldsymbol{x}^i, \hat{y}^i; \theta_t),\end{aligned} \tag{5.8}$$

where α_t is the step size or – in the context of machine learning – the learning rate at time step t.

As mentioned above, we approximate the true objective function and the true gradient by an estimation of the error and a noisy gradient using only the information provided in the minibatch. We denote the size of the minibatch, i.e. the number of examples, by N_B. The update equation (5.8) thus becomes

$$\theta_{t+1} = \theta_t - \alpha_t \frac{1}{N_B} \sum_{i \in \mathcal{I}_t} \nabla_\theta C(\boldsymbol{x}^i, \hat{y}^i; \theta_t), \tag{5.9}$$

where $\mathcal{I}_t \subset \{1, \dots, N\}$ and $|\mathcal{I}_t| = N_B \ll N$ for every t. In each iteration a new minibatch is randomly drawn from the remaining samples in the dataset. When all samples have been used, one *epoch* of the training is complete and the whole dataset is again available for the minibatches.

From this approach arises the simplified *stochastic gradient descent* (SGD), which is still widely used in machine learning problems due to its simplicity. Its convergence properties have been intensively studied. Some main results are for example the requirements $\sum_t \alpha_t^2 < \infty$, $\sum_t \alpha_t = \infty$ on the learning rate schedule to obtain convergence [Bot10]. This implies that the learning rate should become smaller as training proceeds and the parameter vector is approaching an optimum. At the same time the sum of all learning rates should be infinity, so that an optimum can be reached from any initialization. There are also convergence results which include cases where the loss function is not everywhere differentiable. This is especially important regarding neural networks when ReLUs and MaxPooling are used instead of their smooth approximations as they are non-differentiable. The following theorem formalizing a convergence result on convex objective functions, especially for the case $N_B = 1$, is taken from [SSBD14, Chapter 14]:

Theorem 5.1. *Let $B, \rho > 0$. Let $f : \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ be a convex function such that for the expected risk $E_{\mathcal{D}} : \mathbb{R}^n \to \mathbb{R}$ with true data distribution $\mathcal{D}$ it holds that*

$$E_{\mathcal{D}}(\theta) = \mathbb{E}_{(\boldsymbol{x},\hat{y})\sim\mathcal{D}}[f(\boldsymbol{x}, \hat{y}; \theta)]. \tag{5.10}$$

Furthermore, let $\theta^* \in arg\min_{\theta;\|\theta\|_2 \leq B} E_{\mathcal{D}}(\theta)$. Assume that for all t, $\|\nabla f(\boldsymbol{x}_t, \hat{y}_t; \theta_t)\|_2 \leq \rho$ with probability 1. Then, for every $\epsilon > 0$, if we run SGD for minimizing f with respect to θ for T iterations,

$$T \geq \frac{B^2\rho^2}{\epsilon^2} \tag{5.11}$$

with constant step size $\alpha = \sqrt{\frac{B^2}{\rho^2 T}}$, the output $\bar{\theta} = \frac{1}{T}\sum_{t=1}^{T}\theta_t$ of SGD satisfies

$$\mathbb{E}[E_{\mathcal{D}}(\bar{\theta})] \leq \min_{\theta} E_{\mathcal{D}}(\theta) + \epsilon. \tag{5.12}$$

There exist multiple extensions of the simple SGD method. We will describe the most common of them briefly here. However, in the experiments we only test stochastic gradient descent with momentum. These algorithms are implemented for example in the deep learning frameworks *Tensorflow* [AAB+16] and *Caffe* [JSD+14]. We will denote the gradient of the expected risk function evaluated on the current minibatch by $g_t \in \mathbb{R}^n$.

Stochastic gradient descent with momentum

In SGD with momentum an additional hyperparameter controls how much information from gradients on previous minibatches are used to update the current weight vector. Denoting the update quantity by $v_t \in \mathbb{R}^n$, the update equations look as follows:

$$\begin{aligned} v_t &= \eta v_{t-1} + \alpha_t g_t, \\ \theta_t &= \theta_{t-1} - v_t. \end{aligned} \tag{5.13}$$

The momentum parameter $\eta \in [0, 1]$ is usually set to a value around 0.9. This method accelerates the classic SGD algorithm when the optimum lies in a long and narrow valley. In this case the weights approach the optimum in a zig-zag-line, which implies that a lot of iterations are needed to make significant progress when no momentum is used. The momentum term accelerates optimization by using higher learning rates in dimensions where the gradient has not changed its directions for several iterations and using smaller learning rates in dimensions where the gradient is not stable. Thus, the SGD with momentum method accelerates optimization in directions with low curvature by preventing oscillation and instead moving further in these directions as not much change of the objective is expected there [SMDH13].

AdaGrad

AdaGrad stands for 'adaptive (sub)gradient algorithm'. Roughly speaking, the AdaGrad algorithm adapts the learning rate in such a way that it uses high learning rates for rarely occurring features and low learning rates for frequently occurring features [DHS11]. It is therefore suitable for sparse data. In contrast to the standard SGD algorithm AdaGrad applies an individual learning rate for each parameter. There again exist several versions of AdaGrad. In its simplest form the update looks as follows

$$\begin{aligned} v_t &= v_{t-1} + g_t^2, \\ \theta_t &= \theta_{t-1} - \frac{\alpha}{\sqrt{v_t + \epsilon}} \odot g_t, \end{aligned} \tag{5.14}$$

where $\odot$ denotes the elementwise vector-vector-multiplication and g_t^2 denotes the elementwise square $g_t \odot g_t$. The vector v_t thus contains in each element i the sum of the squared

partial derivatives of the objective w.r.t. θ_i up to time step t. The value α is a default basic learning rate and $\epsilon > 0$ is a small stabilizing parameter that prevents division by zero. The main advantages of AdaGrad compared to SGD lie in the fact that it is not necessary anymore to tune the learning rate manually and that it provides a different learning rate for each parameter by recognizing sparse features. The step size decreases automatically during training. While this is a desired property, it also represents a drawback of this algorithm: the learning rate keeps shrinking in each iteration until eventually it becomes infinitesimally small which makes training inefficient. Additionally, this method is very sensitive to initialization and the choice of the global learning rate α. If the gradients are large at the beginning, the learning rate will be small and stay small during the whole training.

AdaDelta

An algorithm similar to AdaGrad is the AdaDelta algorithm. It also applies different learning rates for different parameters. In addition it tackles both the problem of the monotonously decreasing learning rate and the need of the choice of a global learning rate occurring in AdaGrad. Both of the algorithms are computationally only slightly more expensive than standard SGD.

One idea of AdaDelta is to accumulate the sum of squared gradients only over a window of some fixed size w instead of summing up all gradients up to the current iteration like AdaGrad. This has the effect that the denominator in (5.14) cannot become infinitely large and therefore ensures that training continues even after many iterations [Zei12]. To avoid storing all w previous gradients the accumulation is performed by computing an exponentially decaying average with a decay constant $\eta \in [0, 1]$:

$$E[g_t^2] = \eta E[g_{t-1}^2] + (1 - \eta) g_t^2. \tag{5.15}$$

The update is performed using the square root of $E[g_t^2]$, which effectively turns out to be the root mean square (RMS) of all previous gradients:

$$\text{RMS}[g_t] := \sqrt{E[g_t^2] + \epsilon}, \tag{5.16}$$

where $\epsilon > 0$ is a conditioning constant as in the AdaGrad formula (5.14).

The second idea is to correct the units of the updates. SGD both with and without momentum as well as AdaGrad assumes the cost function to be unitless as the units of the update relate to the gradient and not to the parameter itself (see [Zei12]). Similarly to second order methods, AdaDelta approximates the Hessian matrix in order to update the parameter vector with the correct units by multiplying the gradient with the RMS of the previous updates $v_\tau = \Delta\theta_\tau$ for $\tau = 1, \ldots, t$ with the same constant ϵ. The iteration thus becomes:

$$\begin{aligned} v_t &= \frac{\text{RMS}[v_{t-1}]}{\text{RMS}[g_t]} \odot g_t, \\ \theta_t &= \theta_{t-1} - v_t. \end{aligned} \tag{5.17}$$

AdaDelta removed the necessity of choosing a global learning rate. However, it introduced hyperparameters η and ϵ for the moving average and RMS respectively, but the method is much less sensitive to the choice of these parameters than SGD and AdaGrad to the choice of α [Zei12]. In conclusion, AdaDelta is a robust algorithm for training neural networks.

RMSProp

The RMSProp algorithm is an unpublished method proposed by Hinton [TH12]. In fact, it is almost identical to the AdaDelta update. RMSProp only computes a moving average of the gradients but does not incorporate the second order approximation used in AdaDelta

to correct the parameter units. Thus, when using RMSProp there is still the necessity to choose a global learning rate α. But RMSProp also presents the advantage of per-dimension learning rates. The update rule looks as follows

$$\begin{aligned} v_t &= \frac{\alpha}{\text{RMS}[g_t]} \odot g_t, \\ \theta_t &= \theta_{t-1} - v_t. \end{aligned} \tag{5.18}$$

Adam

The name Adam stems from 'adaptive moment estimation'. It was proposed in [KB14]. The method works well with sparse gradients like AdaGrad and is also computationally efficient. It estimates the first and second moments of the gradient in order to find a good search direction and adaptive learning rate. Instead of moving in the direction of the negative gradient, Adam uses the exponential moving average of the gradient as update direction and scales it by the exponential moving average of the squared gradient. As the moving averages are initialized by zeroes, they are biased towards zero, which has to be corrected in every iteration. The update iteration looks as follows

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}, \\ \theta_t &= \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \end{aligned} \tag{5.19}$$

The variables m_t and v_t denote the moving averages of the first and second moment respectively and $\hat{m}_t$ and $\hat{v}_t$ are their bias-corrected versions. As in the previous methods α is the global learning rate and ϵ a parameter that prevents from dividing by zero. The hyperparameters $\beta_1,\ \beta_2 \in [0, 1)$ control the decay rate of the moving averages. In [KB14] the choices $\alpha = 0.001$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are proposed and claimed to work well with many problems.

Although Kingma and Ba [KB14] found in their experiments on CNNs that the second moment estimate approximates the properties of the cost function badly, the overall performance of Adam is much better than AdaGrad. This is partly due to the first moment estimation that reduces the minibatch variance, which improves the convergence speed.

Chapter 6

Second order type methods

The versions of SGD described in Chapter 5 achieve at maximum a sublinear rate of convergence in real problems [DHS11, KB14]. It is desired to improve the training of convolutional neural networks both in time and performance. While the just described methods partly include some approximated second order information by the squared gradient, in this chapter we are going to discuss optimization methods that make use of the Hessian matrix and have better theoretical convergence properties.

Starting from the basic gradient descent equation (5.8), we can obtain optimization methods that incorporate second order information, by additionally multiplying the gradient in equation (5.8) by a positive definite matrix $\mathbf{B}(\theta) \in \mathbb{R}^{n \times n}$ which represents the exact or an approximated inverse of the Hessian matrix of the loss function with respect to the parameters evaluated at θ. In the following we consider an objective function $f : \mathbb{R}^n \to \mathbb{R}$ depending on θ that is twice continuously differentiable. It generalizes the loss function of a neural network that optionally includes a regularization term from Section 5.1. Thus, the problem

$$\theta^* = \arg\min_{\theta} f(\theta) \tag{6.1}$$

that we want to solve by using the second order optimization methods, is large-scale, nonlinear and nonconvex.

In the following, we denote the objective value of the whole dataset $(\boldsymbol{X}^i, \hat{Y}^i) = (\boldsymbol{x}^i, \hat{y}^i)_{i=1,\dots,N}$ at parameter vector θ by $f(\theta)$. If we refer to the objective value computed only over a minibatch $B \subseteq \{1, \dots, N\}$, we write $f_B(\theta)$. The notation for gradients and Hessian matrices is analog.

We then have the general update scheme:

$$\theta_{t+1} = \theta_t - \alpha_t \mathbf{B}(\theta_t) \nabla f(\theta_t) \tag{6.2}$$

By choosing $\mathbf{B}(\theta_t) = I$, the standard gradient descent from equation (5.8) is recovered. A more elaborate choice would be $\mathbf{B}(\theta_t) = -[\nabla^2 f(\theta_t)]^{-1}$. Then the iteration would become Newton's method which has desirable convergence properties. The major drawback of this method is the computation of the inverse Hessian matrix. Even the storage of the matrix is too expensive since its dimension $n \times n$ is too large as n is in the order of millions. Thus, to solve such large-scale optimization problems, algorithms are needed that avoid computing the Hessian matrix of the objective function. However, these algorithms still use second order information as this provides better theoretical convergence properties than the first order type methods considered in Section 5.4 can achieve. Nevertheless, we will start by discussing Newton's method as it is the basis for other more practical algorithms.

6.1 Newton's method

Newton's method, just like gradient descent, is an iterative optimization method. The main idea behind it is to approximate the objective function at the updated parameter values $\theta + d$ for a vector $d \in \mathbb{R}^n$ and fixed parameters $\theta \in \mathbb{R}^n$ by its second order Taylor polynomial $q : \mathbb{R}^n \to \mathbb{R}$, defined as

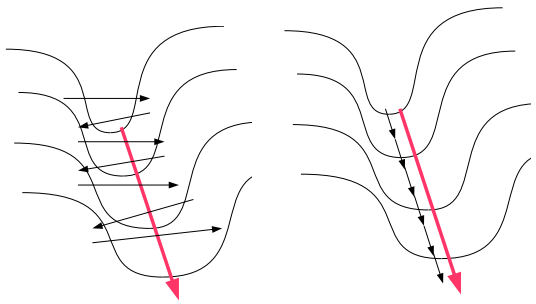


Figure 6.1: Descent in a long narrow valley [Mar10].

$$q(d) := f(\theta) + \nabla f(\theta)^\top d + \frac{1}{2} d^\top \nabla^2 f(\theta) d \approx f(\theta + d). \tag{6.3}$$

In order to minimize this function with respect to d we have to ensure the optimality condition

$$\nabla q(d) = \nabla f(\theta) + \nabla^2 f(\theta) d \stackrel{!}{=} 0, \tag{6.4}$$

which results in the search direction

$$d = -\nabla^2 f(\theta)^{-1} \nabla f(\theta). \tag{6.5}$$

Here, it is assumed that the Hessian matrix $\mathbf{H}(\theta) = \nabla^2 f(\theta)$ is positive definite because otherwise it is not even guaranteed that the function (6.3) has a minimum. Furthermore, positive definiteness implies that the matrix is invertible. In practice, due to the nonconvexity of the objective function in deep learning, it is thus necessary to add a positive definite matrix to the Hessian $\mathbf{H}(\theta)$, such as the scaled identity matrix $\lambda \mathbf{I}$ for some $\lambda \geq 0$. This technique is called *damping* [Mar16].

An important property of Newton's method is its "scale invariance" as stated in [Mar10]. The parameters θ can be linearly rescaled without entailing a different behavior of the Newton iteration. This helps in optimizing the cost function since a parameterwise adjustment of the learning rate is not necessary. The curvature information is taken into account to rescale the gradient in order to get a more promising search direction. Roughly speaking, we want the step taken in a direction of low curvature to be large compared to a direction of high curvature. The image of a long narrow slopy valley describing the objective function is a good intuition. In Figure 6.1 an example is shown. Here, we need to follow the bottom of the valley in order to make progress. The black arrows indicate the steps taken by gradient descent, on the left with a large step size compared to the right with a smaller step size. The thick red arrow shows one step that is taken by a Newton iteration. We see that gradient descent takes much more updates to get to the same result as one single Newton step. Formally, Newton's method performs this intuition by computing the step size α for minimizing the quadratic loss function along the ray $\theta + \alpha d$ for a given search direction d as

$$\alpha = \frac{-\nabla f(\theta)^\top d}{d^\top \mathbf{H}(\theta) d}. \tag{6.6}$$

Another problem that can be overcome by Newton's method is the phenomenon of vanishing or exploding gradients and curvature. When propagating the error back through the net, the gradient can become very small or very large in the layers close to the input relative to the gradient associated with the layers closer to the output. Analogously, the curvature of the objective can shrink or grow excessively. The nature of the Newton update can compensate this unbalance by rescaling the search direction accordingly.

Since, as discussed above, in deep learning it is infeasible to compute or even store the Hessian matrix, let alone inverting it, we cannot apply Newton's method directly. A method that avoids this problem is discussed in the next section.

6.2 Conjugate gradient method

We can apply the conjugate gradient method to solve (6.4) without explicitly computing or inverting the Hessian matrix $\mathbf{H}(\theta) = \nabla^2 f(\theta)$. Hence, the powerful Newton iteration can be executed in spite of the large problem size. We will describe the conjugate gradient method in general before discussing its application to neural networks in detail.

The conjugate gradient method (CG) [NW06, Chapter 5] is the standard algorithm for solving a large linear system

$$\mathbf{A}x = b, \tag{6.7}$$

where $\mathbf{A}$ is a symmetric and positive definite matrix. Solving (6.7) is equivalent to minimizing the function $\psi : \mathbb{R}^n \to \mathbb{R}$,

$$\psi(x) = \frac{1}{2}x^\top \mathbf{A}x - b^\top x, \tag{6.8}$$

i.e. applying the conjugate gradient method to (6.7) with $\mathbf{A} = \mathbf{H}(\theta)$ and $b = \nabla f(\theta)$ generates the search direction $x = d$ that minimizes the approximated error function (6.3). Note, that the constant term in (6.3) can be omitted.
Following [NW06, Chapter 5] we will now derive the conjugate gradient algorithm. The CG method finds the solution to (6.8) by iteratively computing conjugate directions p_i and minimizing $\psi(x_i)$ along these directions.

Definition 6.1. *A set of vectors $\{p_0, \dots, p_k\} \subset \mathbb{R}^n \backslash \{0\}$ is called conjugate with respect to the symmetric positive definite matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ if $p_i^\top \boldsymbol{A} p_j = 0$ for all $i, j = 0, \dots, k,\ i \neq j$.*

Lemma 6.1. *A set $\mathcal{P} \subset \mathbb{R}^n$ of conjugate vectors with respect to some symmetric positive definite matrix $\boldsymbol{A}$ is linearly independent.*

Proof. Assume that two vectors $p_i, p_j \in \mathcal{P}$ are not linearly independent. Then, there exists a number $a \in \mathbb{R},\ a \neq 0$ such that $p_i = ap_j$. We have

$$p_i^\top \mathbf{A} p_j = a p_j^\top \mathbf{A} p_j \neq 0 \tag{6.9}$$

since $a \neq 0$ and $p_j^\top \mathbf{A} p_j > 0$ because $\mathbf{A}$ is positive definite. This contradicts the assumption that p_i and p_j are conjugate with respect to $\mathbf{A}$. □

The first search direction p_0 is initialized with $p_0 = -\nabla\psi(x_0) = -\mathbf{A}x_0 + b$, the steepest descent direction at the initial point x_0. Minimizing the function $\psi(x_k)$ along the direction p_k leads to the step $\alpha_k = -\frac{r_k^\top p_k}{p_k^\top \mathbf{A} p_k}$ where $r_k = \mathbf{A}x_k - b = \nabla\psi(x_k)$ denotes the residual. The next conjugate direction p_{k+1} is computed by only using the residual at $x_{k+1} = x_k + \alpha_k p_k$ and the previous conjugate direction p_k:

$$p_{k+1} = -r_{k+1} + \beta_k p_k. \tag{6.10}$$

By premultiplying this equation by $p_k^\top \mathbf{A}$ and requiring $p_k^\top \mathbf{A} p_{k+1} = 0$ for conjugacy, we get

$$\beta_k = \frac{-r_{k+1}^\top \mathbf{A} p_k}{p_k^\top \mathbf{A} p_k}. \tag{6.11}$$

To make the whole method computationally less expensive, one can equivalently define $\alpha_k = \frac{r_k^\top r_k}{p_k^\top \mathbf{A} p_k}$ and $\beta_k = \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$ (see [NW06] for more details). The algorithm is summarized in Algorithm 6.1.

The computational cost of the CG method is $\mathcal{O}(n)$ per iteration for $\mathbf{A} \in \mathbb{R}^{n\times n}$. Its rate of convergence is strongly dependent on the distribution of the eigenvalues of $\mathbf{A}$. The following theorem is taken from [NW06].

Theorem 6.1. *Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be symmetric and positive definite. If $\boldsymbol{A}$ has r distinct eigenvalues, the conjugate gradient algorithm finds the true solution of* (6.7) *in at most r iterations.*

Algorithm 6.1 Conjugate gradient algorithm (CG)

Input: $x_0, \mathbf{A}, b$
  $r_0 \leftarrow \mathbf{A}x_0 - b$
  $p_0 \leftarrow -r_0$
  $k \leftarrow 0$
  while stopping criterion is not fulfilled **do**
    $\alpha_k \leftarrow \frac{r_k^\top r_k}{p_k^\top \mathbf{A} p_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k + \alpha_k \mathbf{A} p_k$
    $\beta_k \leftarrow \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$
    $p_{k+1} \leftarrow -r_{k+1} + \beta_k p_k$
    $k \leftarrow k + 1$
  end while
Output: x_k

Proof. We begin the proof by expressing iterate x_{k+1} as the polynomial

$$x_{k+1} = x_0 + \alpha_0 p_0 + \alpha_1 p_1 + \cdots + \alpha_k p_k. \tag{6.12}$$

It can be shown [NW06], that

$$\operatorname{span}\{p_0, p_1, \ldots, p_k\} = \operatorname{span}\{r_0, \mathbf{A}r_0, \ldots, \mathbf{A}^k r_0\}. \tag{6.13}$$

Thus, there exist $\tau_i \in \mathbb{R}$, $i = 0, \ldots, k$ inducing the polynomial $P_k^* : \mathbb{R}^{n\times n} \to \mathbb{R}^n$, $\mathbf{A} \mapsto \tau_0 \mathbf{I} + \tau_1 \mathbf{A} + \cdots + \tau_k \mathbf{A}^k$ of degree k, for which

$$x_{k+1} = x_0 + P_k^*(\mathbf{A}) r_0. \tag{6.14}$$

Using the weighted norm associated with matrix $\mathbf{A}$, defined by

$$\|z\|_{\mathbf{A}}^2 := z^\top \mathbf{A} z, \tag{6.15}$$

the distance of the iterate x to the solution x^* of minimizing (6.8) can be written as

$$\frac{1}{2}\|x - x^*\|_{\mathbf{A}}^2 = \frac{1}{2}(x - x^*)^\top \mathbf{A}(x - x^*) = \psi(x) - \psi(x^*). \tag{6.16}$$

Since x_{k+1} minimizes ψ over the set $\{x_0 + \operatorname{span}\{p_0, \ldots, p_k\}\}$ ([NW06, Theorem 5.2]), the polynomial P_k^* is the solution to the problem

$$\min_{P_k} \|x_0 + P_k(\mathbf{A}) r_0 - x^*\|_{\mathbf{A}}^2. \tag{6.17}$$

Let $0 < \lambda_1 \leq \cdots \leq \lambda_n$ be the eigenvalues of $\mathbf{A}$ and $v_1, \ldots, v_n \in \mathbb{R}^n$ the corresponding orthonormal eigenvectors. The eigenvectors span the whole space $\mathbb{R}^n$, i.e. we can write

$$x_0 - x^* = \sum_{i=1}^{n} \xi_i v_i \tag{6.18}$$

for some coefficients $\xi_i \in \mathbb{R}$, $i = 1, \ldots n$. Thus, we get

$$\begin{aligned} x_{k+1} - x^* &= x_0 + P_k^*(\mathbf{A}) r_0 - x^* = \left(\mathbf{I} + P_k^*(\mathbf{A})\mathbf{A}\right)(x_0 - x^*) \\ &= \sum_{i=1}^{n} \left(1 + \lambda_i P_k^*(\lambda_i)\right) \xi_i v_i, \end{aligned} \tag{6.19}$$

where we used that $r_0 = \mathbf{A}x_0 - b = \mathbf{A}(x_0 - x^*)$ and $P_k^*(\mathbf{A})v_i = P_k^*(\lambda_i)v_i$ for $i = 1, \dots, n$, as the polynomial P_k^* can also be defined on scalars and $P_k^*(\mathbf{A})$ has the same eigenvalues as $\mathbf{A}$. Substituting (6.19) into (6.17), we get the estimate

$$\begin{aligned} \|x_{k+1} - x^*\|_{\mathbf{A}}^2 &= \sum_{i=1}^{n} \lambda_i (1 + \lambda_i P_k^*(\lambda_i))^2 \xi_i^2 \\ &= \min_{P_k} \sum_{i=1}^{n} \lambda_i (1 + \lambda_i P_k(\lambda_i))^2 \xi_i^2 \\ &\leq \min_{P_k} \max_{1 \leq i \leq n} (1 + \lambda_i P_k(\lambda_i))^2 \sum_{j=1}^{n} \lambda_j \xi_j^2 \\ &= \min_{P_k} \max_{1 \leq i \leq n} (1 + \lambda_i P_k(\lambda_i))^2 \|x_0 - x^*\|_{\mathbf{A}}^2 \end{aligned} \tag{6.20}$$

Hence, we want to find a polynomial that minimizes the expression

$$\min_{P_k} \max_{1 \leq i \leq n} (1 + \lambda_i P_k(\lambda_i))^2 \tag{6.21}$$

Let us now consider the special case that the eigenvalues take only r distinct values $0 < \rho_1 < \cdots < \rho_r$ and define the polynomial

$$Q_r(\lambda) = \frac{(-1)^r}{\rho_1 \cdots \rho_r} (\lambda - \rho_1) \cdots (\lambda - \rho_r). \tag{6.22}$$

For all λ_i, $i = 1, \dots, n$ it holds that $Q_r(\lambda_i) = 0$ and $Q_r(0) = 1$. Thus, the polynomial $Q_r(\lambda) - 1$ is a polynomial of degree r with a root at 0, and we can define the polynomial of degree $r - 1$

$$\bar{P}_{r-1}(\lambda) = \frac{Q_r(\lambda) - 1}{\lambda}. \tag{6.23}$$

By setting $k = r - 1$ in (6.21) we get

$$0 \leq \min_{P_{r-1}} \max_{1 \leq i \leq n} (1 + \lambda_i P_{r-1}(\lambda_i))^2 \leq \max_{1 \leq i \leq n} (1 + \lambda_i \bar{P}_{r-1}(\lambda_i))^2 = \max_{1 \leq i \leq n} Q_r(\lambda_i)^2 = 0 \tag{6.24}$$

We can substitute this in (6.20) and get

$$\|x_r - x^*\|_{\mathbf{A}}^2 = 0, \tag{6.25}$$

which implies $x_r = x^*$ as claimed. □

So for any distribution of eigenvalues no more than n steps are needed to terminate. But if the eigenvalues are clustered and the condition number $\kappa(\mathbf{A}) := \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$, which is the ratio of the largest to the smallest eigenvalue, is close to 1, the iterates x_k approach the true solution x^* much faster. The following inequality holds for the CG method [NW06]:

$$\|x_k - x^*\|_{\mathbf{A}} \leq \left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1} \right)^{2k} \|x_0 - x^*\|_{\mathbf{A}}. \tag{6.26}$$

This is a rather coarse estimation, but it can be useful if only the extreme eigenvalues are known.

Thus, it is desirable to accelerate the convergence of CG by transforming the linear system (6.7) to get a new matrix $\hat{\mathbf{A}}$ with an improved distribution of eigenvalues. This *preconditioning* is performed by transforming the variable x by a regular matrix $\mathbf{C}$

$$\hat{x} = \mathbf{C}x. \tag{6.27}$$

Transforming (6.8) accordingly results in the new function $\hat{\psi} : \mathbb{R}^n \to \mathbb{R}$,

$$\hat{\psi}(\hat{x}) = \psi(\mathbf{C}^{-1}\hat{x}) = \frac{1}{2}\hat{x}^\top \mathbf{C}^{-\top}\mathbf{A}\mathbf{C}^{-1}\hat{x} - (\mathbf{C}^{-\top}b)^\top \hat{x}. \tag{6.28}$$

Minimizing $\hat{\psi}$ thus requires solving the linear system

$$\mathbf{C}^{-\top}\mathbf{A}\mathbf{C}^{-1}\hat{x} = \mathbf{C}^{-\top}b. \tag{6.29}$$

Hence, the rate of convergence now depends on the eigenvalue distribution of the matrix $\mathbf{C}^{-\top}\mathbf{A}\mathbf{C}^{-1}$. The preconditioning matrix $\mathbf{C}$ should be chosen such that the condition number of $\mathbf{C}^{-\top}\mathbf{A}\mathbf{C}^{-1}$ gets smaller. Now the CG algorithm can be applied to (6.29) to solve for $\hat{x}$ and then transforming the result back to x. It turns out that we do not need the matrix $\mathbf{C}$ but rather the symmetric and positive definite matrix $\mathbf{P} := \mathbf{C}^\top\mathbf{C}$ [NW06]. The resulting preconditioned conjugate gradient algorithm (PCG) is shown in Algorithm 6.2.

Algorithm 6.2 Preconditioned conjugate gradient algorithm (PCG)

Input: $x_0, \mathbf{A}, b$, preconditioning matrix $\mathbf{P}$
  $r_0 \leftarrow \mathbf{A}x_0 - b$
  $y_0 \leftarrow$ solution of $\mathbf{P}y = r_0$
  $p_0 \leftarrow -y_0$
  $k \leftarrow 0$
  while stopping criterion is not fulfilled **do**
    $\gamma \leftarrow \frac{r_k^\top y_k}{p_k^\top \mathbf{A}p_k}$
    $x_{k+1} \leftarrow x_k + \gamma p_k$
    $r_{k+1} \leftarrow r_k + \gamma \mathbf{A}p_k$
    $y_{k+1} \leftarrow$ solution of $\mathbf{P}y = r_{k+1}$
    $\beta \leftarrow \frac{r_{k+1}^\top y_{k+1}}{r_k^\top y_k}$
    $p_{k+1} \leftarrow -y_{k+1} + \beta p_k$
    $k \leftarrow k + 1$
  end while
Output: x_k

By setting $\mathbf{P} = \mathbf{I}$, the standard conjugate gradient algorithm 6.1 is recovered. PCG has the computational overhead compared to CG of solving the linear system $\mathbf{P}y = x$ in each iteration. It is therefore important for $\mathbf{P}$ to be simple in the sense of being easy to invert. Thus, $\mathbf{P}$ is often designed as a diagonal matrix.

6.3 Hessian-free method

The Hessian-free method (HF) combines the ansatz of Newton's method of approximating the objective function by a quadratic function with the conjugate gradient method to solve the resulting linear system [Mar10]. With the conjugate gradient algorithm we have an efficient tool to perform Newton's iteration because it does not require inverting the (damped) Hessian matrix $\mathbf{H}(\theta)$. We only have to compute the product of the Hessian with arbitrary vectors. But these products can be approximated without explicit knowledge about the Hessian by computing the finite directional difference of the gradient in direction $v \in \mathbb{R}^n$

$$\mathbf{H}(\theta)v = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon v) - \nabla f(\theta)}{\epsilon} \tag{6.30}$$

which also justifies the name 'Hessian-free'. Since finite differences can be inexact and numerically unstable, we instead use Algorithm 4.1 to compute the Hessian-vector products $\mathbf{H}(\theta)v$ used in the CG algorithm.

However, it is impractical to perform n steps of the CG algorithm to find the exact solution to (6.4). Typically, a significant progress is achieved within few iterations. Thus, due to this approach of solving (6.4) only approximately, the Hessian-free method is also called *truncated* or *inexact Newton's method*. This optimization method has already been studied for decades (e.g., [Nas82, NW06]) but only in 2010 it was seriously applied to deep neural networks by Martens [Mar10]. The basic Hessian-free algorithm in pseudo-code is described in Algorithm 6.3. In the following we will describe the details for the implementation.

Although the application of CG to perform the approximated Newton step seems to be straightforward, there are several issues arising from the specific setup of deep learning that have to be taken care of.

Algorithm 6.3 Hessian-free algorithm (pseudo-code)

Input: $\theta_0, \lambda_0 > 0, \zeta \in (0,1)$
  $d_{-1} \leftarrow 0$
  for $t = 0, \dots, T$ **do**
    choose minibatch B_t
    $g_t \leftarrow \nabla f_{B_t}(\theta_t)$
    compute preconditioning matrix $\mathbf{P}(\theta_t)$
    choose new minibatch B_t'
    define $\mathbf{A}(d) := \nabla^2 f_{B_t'}(\theta_t)d + \lambda_t d$
    $d_t \leftarrow \text{PCG}(\zeta d_{t-1}, \mathbf{A}, -g_t, \mathbf{P}(\theta_t))$
    select learning rate α_t
    $\theta_{t+1} \leftarrow \theta_t + \alpha_t d_t$
    compute new damping parameter λ_{t+1}
  end for

6.3.1 Minibatching

The first question is about how to handle the large training set. As explained in Section 5.3 we compute the loss only on a minibatch, a small subset of the data, in each iteration. When should we replace the minibatch by a new one though? Martens [Mar10] and Byrd et al. [BCNN11] both suggested not to change the sample batch during one CG run because the conjugate gradient algorithm relies on a fixed matrix to ensure the conjugacy property of the computed directions. Furthermore, the authors claim that it is not necessary to compute the Hessian matrix of the objective on a large minibatch as Newton-like optimization methods can handle inexact curvature information. Using smaller batches reduces the computational cost significantly. However, in both works the gradient was computed on the whole dataset because this computational overhead compared to SGD can be compensated by the greater progress that is achieved by the second order approach. In our experiments, however, we will use the same batch size both for the gradient and for the Hessian computation. Moreover, we found that the algorithm is more efficient when selecting a different minibatch for the curvature information.

6.3.2 Damping

Recall that within the CG iteration we do not minimize the true objective function but its second order Taylor polynomial around θ. This approximation is only reliable close to θ unless the loss function f was quadratic. Since for convolutional neural networks the function f is neither quadratic nor convex, the Hessian matrix $\mathbf{H}(\theta) = \nabla^2 f(\theta)$ is not even positive definite as it is required for the CG method to generate a descent direction. When $\mathbf{H}(\theta)$ is indefinite, a minimizer of (6.3) may not exist and CG may produce an unreasonable update vector d which does not lie in the trust region of the quadratic approximation of the objective. This problem can be solved by adding a penalty term to the quadratic approximation

(6.3) to keep the update d small [Mar16]. In order to regularize the loss function we will use the Tikhonov style damping to regularize the update and minimize the function $\hat{q} : \mathbb{R}^n \to \mathbb{R}$,

$$\hat{q}(d) := f(\theta) + \nabla f(\theta)^\top d + \frac{1}{2} d^\top \mathbf{H}(\theta) d + \frac{\lambda}{2} \|d\|_2^2 \tag{6.31}$$

instead of (6.3). This results in applying the conjugate gradient method with $\mathbf{A} := \hat{\mathbf{H}}(\theta) = \mathbf{H}(\theta) + \lambda \mathbf{I}$. Here, we see that this damping technique can also overcome the lack of positive definiteness of $\mathbf{H}(\theta)$ when λ is big enough. It is important to find an appropriate value for λ. If it is too small, the damping has no effect and the CG algorithm may diverge. If it is too large, the updates stay too close to the current value and no significant progress is made. Especially for arbitrary initialization of the parameters θ it is important that the optimization algorithm can find a minimum which does not lie in the vicinity of the initial θ_0 in a reasonable amount of iterations.

The damping is in particular necessary at the beginning of the optimization since this is where the objective function may be highly nonlinear and nonconvex. When the iterates θ_t approach a local minimum the quadratic approximation becomes more accurate and less damping is needed [Mar16]. Therefore, the choice of λ should not be constant throughout the whole optimization but adapt to the local shape of the graph of the loss function. Martens [Mar10] proposed to update λ after each HF iteration using the Levenberg-Marquardt heuristic based on a measure ρ for the relation between the actual reduction of the loss $f(\theta_t + \alpha_t d_t)$ and the predicted reduction using its quadratic approximation $q(\alpha_t d_t)$:

$$\rho = \frac{f(\theta_t + \alpha_t d_t) - f(\theta_t)}{q(\alpha_t d_t) - q(0)}. \tag{6.32}$$

If ρ is too small, the quadratic model overestimates the reduction and we should increase λ to make the trust region smaller. If ρ is close to 1, we can decrease λ because we are more confident in the approximation and can afford to allow bigger and more substantial update steps. Martens chose the following, somewhat arbitrary, values for updating λ:

$$\begin{aligned} &\text{if } \rho < 0.25 : \ \lambda_{t+1} \leftarrow 1.5 \lambda_t, \\ &\text{if } \rho > 0.75 : \ \lambda_{t+1} \leftarrow \frac{2}{3} \lambda_t. \end{aligned} \tag{6.33}$$

Another type of damping that we also applied in our experiments is line searching. The CG algorithm generates an update proposal d_t, which is – as long as $\hat{\mathbf{H}}(\theta_t)$ is positive definite – a descent direction [NW06]. Hence, there exists a step size α_t which produces a loss value $f(\theta_t + \alpha_t d_t) < f(\theta_t)$. We use the line search method of sufficient decrease and backtracking with initial $\alpha = 1$ as defined in Algorithm 6.4 [NW06, Chapter 3]. There it is assumed that the search direction d_t is a descent direction at θ_t, i.e. $\nabla f(\theta_t)^\top d_t < 0$. The value for c is typically small, for example $c = 0.01$ or $c = 0.001$.

Algorithm 6.4 Backtracking line search

Input: $\alpha > 0, \beta \in (0,1), c \in (0,1)$
  while $f(\theta_t + \alpha d_t) > f(\theta_t) + c\alpha \nabla f(\theta_t)^\top d_t$ **do**
    $\alpha \leftarrow \beta \alpha$
  end while
Output: α

If the approximation $\hat{q}(d_t)$ is accurate enough and $\hat{\mathbf{H}}(\theta_t)$ is positive definite, the choice $\alpha_t = 1$ fulfills the sufficient decrease condition

$$f(\theta_t + \alpha_t d_t) \leq f(\theta_t) + c\alpha_t \nabla f(\theta_t)^\top d_t. \tag{6.34}$$

Thus, we initialize Algorithm 6.4 with $\alpha = 1$ and only if the Tikhonov damping failed, resulting in a search direction d_t that is too big and therefore not trustworthy enough because of

a bad approximation or indefiniteness of the Hessian matrix, the learning rate α_t is reduced to ensure a reduction of the loss. This line search is a popular method for Newton-like algorithms as it guarantees some local convergence theorems [NW06]. Due to the Tikhonov damping additionally applied, it rarely occurs that $\alpha = 1$ is not a suitable step size. Hence, the additional computational cost of the line search procedure is small.

6.3.3 The Gauss-Newton matrix

Martens further adapted the Hessian-free method to nonconvex deep learning problems by replacing the Hessian matrix by the *Generalized Gauss-Newton matrix* [Mar10] which is positive semidefinite. This property is exploited by the CG-algorithm and therefore makes the Hessian-free method more robust and improves its performance. In Section 4.2 we define the Gauss-Newton matrix $\mathbf{G}(\theta)$ and with Algorithm 4.2 we give an explicit algorithm for computing the product $\mathbf{G}(\theta)v$ exactly for some $v \in \mathbb{R}^n$.

By replacing the Hessian matrix in the quadratic approximation (6.3) by the Gauss-Newton matrix, we obtain a minimization problem that has a local minimum and the CG method applied to the corresponding linear system will generate for any $\lambda \geq 0$ a search direction that is a descent direction. The damping term $\lambda\|d\|_2^2$ is thus less important when using the Gauss-Newton matrix since it only has to compensate the inaccuracy of the quadratic approximation. However, it should not be omitted completely.

6.3.4 Preconditioning

Preconditioning is a method to accelerate the conjugate gradient algorithm. It is not obvious what the preconditioning matrix should look like. Therefore, Martens [Mar10] experimented with different matrices. We modified his choice slightly to

$$\mathbf{P}(\theta) = \left(\operatorname{diag}\left[\left(\sum_{i=1}^{N_B} \nabla f(\boldsymbol{x}^i, \hat{y}^i; \theta) \right) \odot \left(\sum_{i=1}^{N_B} \nabla f(\boldsymbol{x}^i, \hat{y}^i; \theta) \right) \right] + \lambda \mathbf{I} \right)^{\gamma}, \tag{6.35}$$

where $\nabla f(\boldsymbol{x}^i, \hat{y}^i; \theta)$ is the gradient of the loss function on the i-th example in the current batch of size N_B. Martens chose the value γ smaller than 1, in order to suppress extreme values. The parameter λ is the same as used in the Tikhonov damping, since the choice of $\mathbf{P}(\theta)$ aims to transform the eigenvalues of the matrix $\mathbf{A}(\theta) = \mathbf{H}(\theta) + \lambda\mathbf{I}$ in such a manner that the condition number of the resulting matrix is decreased. As a diagonal matrix $\mathbf{P}(\theta)$ is easy to invert and thus the computational extra effort of PCG compared to CG is negligible.

6.3.5 Information sharing across HF iterations

In a standard run of the conjugate gradient method the search direction is initialized with 0. This makes sense since we want to compute a new value of the parameters that should not lie too far from the current parameter vector as the update is computed based only on a local approximation of the loss function. However, similar to the momentum method used in stochastic gradient descent, we can use the previous search direction for computing the new one. This idea is based on the assumption that the curvature of the loss function changes only slowly. Therefore, a search direction that was a descent direction for one minibatch at θ_t is likely to be a promising search direction for a new minibatch at θ_{t+1} as well. To incorporate the previous information about a descent direction in the next HF iteration one can simply share this information by initializing the new CG run at time step $t+1$ with the old search direction d_t. Martens applied this method with the small modification of scaling d_t by a factor $\zeta \in (0, 1)$ when initializing the new CG run [Mar16]. Although, this starting point can be a worse choice than 0 at the beginning, it is very likely that within a few iterations the CG algorithm can generate a better search direction from d_t as starting point, since the possibly bad impact of d_t comes presumably from the directions with high curvature which are quickly corrected by CG. On the contrary the low-curvature directions are more stable

and will rather stay descent directions for several successive HF iterations [Mar10]. By this information sharing the HF method can be accelerated and improved by a great factor as the CG method terminates after significantly less iterations.

6.3.6 Terminating the CG iteration

It remains to find a good stopping criterion for the CG algorithm. As previously mentioned, it is not desirable to run a full cycle of n iterations to find the exact solution of Newton's equation. A typical condition for terminating CG is when the norm of the residual $r_t = \mathbf{A}(\theta)d_t + \nabla f(\theta_t)$ is smaller than a certain threshold, for example as in [NW06]:

$$\|r_t\|_2 < \min\left\{\frac{1}{2}, \sqrt{\|\nabla f(\theta_t)\|_2}\right\}\|\nabla f(\theta_t)\|_2. \tag{6.36}$$

However, as Martens stated [Mar10] the CG method is not designed to minimize $\|\mathbf{A}d - b\|_2^2$ but rather the function

$$\psi(d) = \frac{1}{2}d^\top \mathbf{A}d - b^\top d, \tag{6.37}$$

with $\mathbf{A}$ the damped Hessian or Gauss-Newton matrix and b the negative gradient. A good solution for one function is not automatically a good solution for the other one as well. Therefore, using the stopping condition (6.36) may lead to a suboptimal result. We instead used, following [Mar10], the stopping criterion that measures the relative reduction per iteration of ψ across several CG iterations. Martens experimented with several different termination conditions and found that the following turned out to be the best choice: CG is terminated after iteration i when

$$i > k,\ \ \psi(d_i) < 0, \text{ and } \frac{\psi(d_i) - \psi(d_{i-k})}{\psi(d_i)} < k\epsilon \tag{6.38}$$

where ϵ is a small constant such as 0.0005 and k is increased as the CG algorithm proceeds. Martens' choice is $k = \max(10, 0.1i)$. The condition $\psi(d_i) < 0$ is due to the fact that with $d = 0$ we would already have $\psi(d) = 0$ and we want the search direction to be a better solution than this trivial one since it would lead to no progress. The per iteration reduction is averaged over k iterations to eliminate some of the variance and get a reliable estimate of the reduction.

6.4 Scaled conjugate gradient method

Møller [Møl93] introduced a variation of the conjugate gradient method that does not need a hyperparameter for the learning rate nor involves a line search. Instead, a Levenberg-Marquardt approach is used to scale the step size. This method is called scaled conjugate gradient (SCG). It can also handle the possible indefiniteness of the Hessian matrix. The approach is very similar to the Hessian-free method with Tikhonov damping described above. Recall, that whenever we compute $\mathbf{H}(\theta)v$ for some vector v, we added the damping term λv to the product in order to overcome the lack of positive definiteness of $\mathbf{H}(\theta)$. In Hessian-free optimization λ is updated at the end of one iteration, but it is only an estimate and does not ensure that the quantity $d^\top(\mathbf{H}(\theta) + \lambda\mathbf{I})d$ used in the CG algorithm is indeed greater than 0.

In contrast, in the scaled conjugate gradient algorithm the parameter λ is adjusted after every evaluation of $d^\top(\mathbf{H}(\theta) + \lambda\mathbf{I})d$ so that for the new value of λ it holds that $d^\top(\mathbf{H}(\theta) + \lambda\mathbf{I})d > 0$. The procedure is as follows. We will adopt the notations of [Møl93]. Let us denote the product $(\mathbf{H}(\theta) + \lambda\mathbf{I})d$ by s and $d^\top s$ by δ for some $d \in \mathbb{R}^n$. Then, if $\delta \leq 0$ we know that the Hessian is not positive definite and we have to raise λ. We denote the adjusted λ by $\bar{\lambda}$, the resulting new s by $\bar{s}$ and the new δ by $\bar{\delta}$. We have

$$\bar{s} = s + (\bar{\lambda} - \lambda)d \tag{6.39}$$

and thus

$$\bar{\delta} = d^\top \bar{s} = d^\top (s + (\bar{\lambda} - \lambda)d) = \delta + (\bar{\lambda} - \lambda)\|d\|_2^2. \tag{6.40}$$

We want $\bar{\delta}$ to be greater than 0, so we require for $\bar{\lambda}$

$$\bar{\lambda} > \lambda - \frac{\delta}{\|d\|_2^2}. \tag{6.41}$$

It is not clear what is an optimal value for $\bar{\lambda}$. We chose

$$\bar{\lambda} = 2\lambda - \frac{\delta}{\|d\|_2^2} \tag{6.42}$$

as suggested in [Nab02]. The step size α_t is then computed with the new $\bar{\delta}$ and the loss is evaluated at the weight parameters $\theta_{k+1} = \theta_t + \alpha_t d_t$. This step is only accepted if a loss reduction is achieved. After each SCG iteration λ is again updated based on the measure ρ of how well the second order Taylor polynomial approximates the true loss function, just like the damping parameter in the Hessian-free method:

$$\rho = \frac{f(\theta_k + \alpha_k d_k) - f(\theta_k)}{q(\alpha_k d_k) - q(0)}. \tag{6.43}$$

If ρ is close enough to 1, i.e. the approximation is good enough, λ is decreased. If it is too small, we have to increase λ. In the case of $\rho < 0$ the Hessian matrix was not positive definite and no error reduction could be achieved with the step $\alpha_t d_t$. Again, λ has to be increased and the weight parameters are not updated. We chose the following thresholds and update rule for λ as in [Nab02]:

$$\begin{aligned} &\text{if } \rho < 0.25: \ \lambda_{t+1} \leftarrow 4\lambda_t, \\ &\text{if } \rho > 0.75: \ \lambda_{t+1} \leftarrow 0.5\lambda_t. \end{aligned} \tag{6.44}$$

If the step was accepted by the algorithm, a new search direction d_{k+1} is computed according to the Polak-Ribière formula [Nab02]:

$$d_{k+1} = \frac{(\nabla f(\theta_{k+1}) - \nabla f(\theta_k))^\top \nabla f(\theta_{k+1})}{\nabla f(\theta_k)^\top \nabla f(\theta_k)} d_k + \nabla f(\theta_{k+1}). \tag{6.45}$$

We have implemented the scaled conjugate gradient method following [Nab02]. The resulting algorithm is shown in Algorithm 6.5. For the algorithm to stop before reaching the maximum number of iterations, we require

$$\max(|\alpha d_k|) < \epsilon_1 \text{ and} \tag{6.46}$$

$$|f(\theta_k + \alpha d_k) - f(\theta_k)| < \epsilon_2, \tag{6.47}$$

where we chose $\epsilon_1 = \epsilon_2 = 10^{-4}$. Algorithm 6.5 is defined for one single batch. Since we want to save computation time, and avoid overfitting on one small batch, the algorithm is not executed until convergence. Instead, we run this algorithm only for a few iterations before replacing the minibatch and starting Algorithm 6.5 again. So, technically the stopping criteria (6.46) and (6.47) are not necessary in our framework as they are nearly never satisfied before the maximum number of iterations is reached.

Since we face the same problem with the indefiniteness of the Hessian matrix as we have seen for the Hessian-free method, we experimented also with the positive semidefinite Gauss-Newton matrix instead of the Hessian. In this case the parameter λ fulfills the only purpose to keep the update step small, which is necessary since the Gauss-Newton matrix is only an approximation of the true second derivative of the objective function. However, λ can stay smaller when the Gauss-Newton matrix is used, since it does not have to compensate indefiniteness. This, in turn, results in more effective update steps.

Algorithm 6.5 Scaled conjugate gradient algorithm (SCG)

Input: $\theta_0, \lambda > 0$, minibatch $B, \mathbf{H}_B(\theta_0)$
  $d_0 \leftarrow -\nabla f_B(\theta_0)$
  $r_0 \leftarrow \nabla f_B(\theta_0)$
  success $\leftarrow$ true
  for $k = 0, 1, \ldots,$ maxiter **do**
    if success = true **then**
      $\mu \leftarrow d_k^\top r_k$
      if $\mu \geq 0$ **then**
        $d_k \leftarrow -r_k$
        $\mu \leftarrow d_k^\top r_k$
      end if
    end if
    $\delta_k \leftarrow d_k^\top \mathbf{H}_B(\theta_k) d_k + \lambda \|d_k\|_2^2$
    if $\delta_k \leq 0$ **then**
      $\delta_k \leftarrow \lambda \|d_k\|_2^2$
      $\lambda \leftarrow \lambda - \frac{d_k^\top \mathbf{H}_B(\theta_k) d_k}{\|d_k\|_2^2}$
    end if
    $\alpha \leftarrow -\frac{\mu}{\delta_k}$
    compute ρ
    if $\rho \geq 0$ **then**
      success $\leftarrow$ true
      $\theta_{k+1} \leftarrow \theta_k + \alpha d_k$
      if stopping condition is fulfilled **then**
        return θ_{k+1}
      else
        $r_{k+1} \leftarrow \nabla f_B(\theta_{k+1})$
        $\gamma \leftarrow \frac{(r_k - r_{k+1})^\top r_{k+1}}{\mu}$
        $d_{k+1} \leftarrow \gamma d_k - r_{k+1}$
      end if
    else
      success $\leftarrow$ false
      $\theta_{k+1} \leftarrow \theta_k$
      $r_{k+1} \leftarrow r_k$
    end if
    update λ
  end for
Output: θ_{k+1}

6.5 Stochastic L-BFGS method

By replacing the Hessian matrix in (6.3) by a symmetric positive definite approximation $\mathbf{B}(\theta) \in \mathbb{R}^{n\times n}$ to $\nabla^2 f(\theta)$ and minimizing the resulting quadratic function $q : \mathbb{R}^n \to \mathbb{R}$,

$$q(d) = f(\theta) + \nabla f(\theta)^\top d + \frac{1}{2} d^\top \mathbf{B}(\theta) d \tag{6.48}$$

we get the class of *Quasi-Newton* optimization methods. The main appeal of these optimization methods is that they do not need second derivatives of the objective function – just as gradient descent methods – but still have improved convergence properties. Moreover, they can be more efficient than Newton's method for to the same reason.

The most popular quasi-Newton method is the BFGS algorithm, named after its developers Broyden, Fletcher, Goldfarb and Shanno. We will describe this method in its standard form as it is derived in [NW06, Chapter 6] before specifying its variant L-BFGS that is applied to large scale optimization problems.

In the following we will use the notation $\mathbf{B}_t := \mathbf{B}(\theta_t)$.

6.5.1 BFGS method

When minimizing (6.48) at time step t, the solution is analogously to (6.5) given as

$$d_t = -\mathbf{B}_t^{-1} \nabla f(\theta_t) \tag{6.49}$$

and the update step is performed by

$$\theta_{t+1} = \theta_t + \alpha_t d_t, \tag{6.50}$$

where α_t is a step size that satisfies the Wolfe conditions explained in the following.

Definition 6.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function, and let $0 < c_1 < c_2 < 1$. A step size α satisfies the Wolfe conditions for the descent direction d if it satisfies both the sufficient decrease condition*

$$f(\theta + \alpha d) \leq f(\theta) + \alpha \nabla f(\theta)^\top d, \tag{6.51}$$

and the curvature condition

$$\nabla f(\theta + \alpha d)^\top d \geq c_2 \nabla f(\theta)^\top d. \tag{6.52}$$

The key idea behind the BFGS method is to update the matrix $\mathbf{B}_t$ at every time step based on the latest curvature information, rather than computing it from scratch. Assume that we have generated new parameters θ_{t+1}. Then the new quadratic model of the loss function f at θ_{t+1} is given by $q_{t+1} : \mathbb{R}^n \to \mathbb{R}$

$$q_{t+1}(d) = f(\theta_{t+1}) + \nabla f(\theta_{t+1})^\top d + \frac{1}{2} d^\top \mathbf{B}_{t+1} d. \tag{6.53}$$

A reasonable requirement to impose on $\mathbf{B}_{t+1}$ is that the gradient of q_{t+1} should agree with the true gradient evaluated at the latest two iterates θ_{t+1} and θ_t. As $\nabla q_{t+1}(0) = \nabla f(\theta_{t+1})$, this is trivially fulfilled for θ_{t+1}. Considering θ_t, we have

$$\nabla q_{t+1}(-\alpha_t d_t) = \nabla f(\theta_{t+1}) - \alpha_t \mathbf{B}_{t+1} d_t. \tag{6.54}$$

By requiring this expression to be equal to $\nabla f(\theta_t)$, we obtain the so called *secant equation*

$$\mathbf{B}_{t+1} \alpha_t d_t = \nabla f(\theta_{t+1}) - \nabla f(\theta_t). \tag{6.55}$$

For convenience, we will define the vectors

$$s_t = \theta_{t+1} - \theta_t = \alpha_t d_t \text{ and} \tag{6.56}$$

$$y_t = \nabla f(\theta_{t+1}) - \nabla f(\theta_t), \tag{6.57}$$

so that (6.55) becomes

$$\mathbf{B}_{t+1} s_t = y_t. \tag{6.58}$$

We will refer to the vectors $\{s_t, y_t\}$ by the term *correction pair*.

As we require $\mathbf{B}_{t+1}$ to be positive definite, it is only possible to satisfy the secant equation (6.58) if the two vectors s_t and y_t satisfy the *curvature condition*

$$s_t^\top y_t > 0. \tag{6.59}$$

This condition is always fulfilled if f is a strongly convex function. As, in our case, the loss function f is nonconvex, we need to guarantee the curvature condition (6.59). Indeed, by imposing the Wolfe conditions as restrictions to the step size α_t, the curvature condition is automatically satisfied, as the common term in Definition 6.2 already suggests:

$$\begin{aligned} & \nabla f(\theta_{t+1})^\top \alpha_t d_t \geq c_2 \nabla f(\theta_t)^\top \alpha_t d_t \\ \iff \quad & \left(\nabla f(\theta_{t+1}) - \nabla f(\theta_t)\right)^\top s_t \geq (c_2 - 1) \nabla f(\theta_t)^\top s_t \\ \iff \quad & y_t^\top s_t > 0, \end{aligned} \tag{6.60}$$

where the last inequality holds since $c_2 < 1$ and s_t is a descent direction at θ_t, i.e. $\nabla f(\theta_t)^\top s_t < 0$.

As we see from equation (6.49), we actually do not need the matrix $\mathbf{B}_t$ to compute the search direction d_t but the inverse $\mathbf{B}_t^{-1}$. Therefore, we formulate the secant equation (6.58) in terms of $\mathbf{A}_t := \mathbf{B}_t^{-1}$

$$\mathbf{A}_{t+1} y_t = s_t \tag{6.61}$$

and directly update $\mathbf{A}_t$ in each iteration. The solution to (6.61) is not unique. $\mathbf{A}_{t+1}$ is thus chosen to be the closest matrix to $\mathbf{A}_t$ with respect to some matrix norm $\|\cdot\|_{\mathbf{M}}$ (see [NW06, Chapter 6] for details). It is the solution to the problem

$$\begin{aligned} \min_{\mathbf{A}} \ & \|\mathbf{A} - \mathbf{A}_t\|_{\mathbf{M}} \\ \text{s.t.} \quad & \mathbf{A} = \mathbf{A}^\top, \\ & \mathbf{A} y_t = s_t \end{aligned} \tag{6.62}$$

In the BFGS method the standard choice for the matrix norm $\|\cdot\|_{\mathbf{M}}$ results in the unique solution to (6.62)

$$\mathbf{A}_{t+1} = \mathbf{V}_t^\top \mathbf{A}_t \mathbf{V}_t + \rho_t s_t s_t^\top, \tag{6.63}$$

with

$$\rho_t = \frac{1}{y_t^\top s_t} \quad \text{and} \quad \mathbf{V}_t = \mathbf{I} - \rho_t y_t s_t^\top. \tag{6.64}$$

$\mathbf{A}_0$ is initialized with an approximation to $\nabla^2 f(\theta_0)^{-1}$. In practice this is often done by setting it to a multiple of the identity matrix $\beta\mathbf{I}$. Unfortunately, there is no choice of β that works well in general. A heuristic that is often chosen for initializing $\mathbf{A}_0$ is to set it to the identity matrix and scale it after the first iteration has been executed, but before $\mathbf{A}_1$ is computed. The scaling factor is chosen such that $\mathbf{A}_0$ has eigenvalues that approximate an eigenvalue of $\nabla^2 f(\theta_0)$ [NW06, Chapter 6]. We then have

$$\mathbf{A}_0 = \frac{y_0^\top s_0}{y_0^\top y_0} \mathbf{I}. \tag{6.65}$$

The complete algorithm is shown in Algorithm 6.6. Being an instance of an algorithm of the Broyden class [NW06] it has nice convergence properties which we will state now. The following theorems are only applicable for a certain class of functions. The precise assumptions are given as follows. Since our loss function actually does not fulfill these assumptions we will skip the proofs. The interested reader can refer to [NW06, Chapter 6.4].

Algorithm 6.6 BFGS algorithm

Input: $\theta_0, \mathbf{A}_0, \epsilon > 0$
  $t \leftarrow 0$
  while $\|\nabla f(\theta_t)\| > \epsilon$ **do**
    $p_t \leftarrow -\mathbf{A}_t \nabla f(\theta_t)$
    $\theta_{t+1} \leftarrow \theta_t + \alpha_t p_t$
      where α_t is computed by a line search to satisfy the Wolfe conditions (6.51), (6.52)
    $s_t \leftarrow \theta_{t+1} - \theta_t$
    $y_t \leftarrow \nabla f(\theta_{t+1}) - \nabla f(\theta_t)$
    $\mathbf{A}_{t+1}$ is computed by means of (6.63)
    $t \leftarrow t + 1$
  end while

Assumption 6.1.

1. *The objective function f is twice continuously differentiable.*
2. *The level set $\mathcal{L} = \{\theta \in \mathbb{R}^n | f(\theta) \leq f(\theta_0)\}$ is convex, and there exist positive constants m and M such that*
$$m\|z\|^2 \leq z^\top \nabla^2 f(\theta) z \leq M\|z\|^2 \tag{6.66}$$
for all $z \in \mathbb{R}^n$ and $\theta \in \mathcal{L}$.

The second assumption guarantees that f has a unique minimizer θ^* in $\mathcal{L}$. For these assumptions the BFGS algorithm is globally convergent, as stated in the following theorem.

Theorem 6.2. *Let $\boldsymbol{A}_0$ be any symmetric positive definite initial matrix, and let θ_0 be a starting point for which Assumption 6.1 is satisfied. Then the sequence $\{\theta_t\}$ generated by Algorithm 6.6 (with $\epsilon = 0$) converges to the minimizer θ^* of f.*

Now, we state the convergence rate of Algorithm 6.6.

Assumption 6.2.
The Hessian matrix $\nabla^2 f$ of f is Lipschitz continuous at θ^*, i. e. there exists a positive constant L such that
$$\|\nabla^2 f(\theta) - \nabla^2 f(\theta^*)\| \leq L\|\theta - \theta^*\| \tag{6.67}$$
for all θ near θ^*.

Now we can formulate the convergence theorem.

Theorem 6.3. *Suppose that f is twice continuously differentiable and that the sequence $\{\theta_t\}$ generated by Algorithm 6.6 converges to a minimizer θ^* at which Assumption 6.2 holds. Suppose also that $\sum_{t=1}^{\infty} \|\theta_t - \theta^*\| < \infty$ holds. Then $\{\theta_t\}$ converges to θ^* at a superlinear rate, i. e.*
$$\|\theta_{t+1} - \theta^*\| = o\left(\|\theta_t - \theta^*\|\right) \tag{6.68}$$
for $k \to \infty$.

6.5.2 L-BFGS method

Again, we encounter the problem that the matrices $\mathbf{A}_t$ are too large to be stored as they are of the same dimension as the Hessian matrix $\nabla^2 f(\theta_t) \in \mathbb{R}^{n \times n}$. To circumvent this issue an algorithm is necessary that computes the product $\mathbf{A}_t v$ for a vector $v \in \mathbb{R}^n$ without storing the whole matrix. The L-BFGS method, short for limited memory BFGS method, is a procedure that only uses the latest M vector pairs $\{s_i, y_i\}$, $i = t - M, \ldots, t - 1$, to directly compute the quantity $\mathbf{A}_t \nabla f(\theta_t)$ and is otherwise identical to the standard BFGS method from Algorithm 6.6. The algorithm is based on the following consideration [NW06, Chapter 7.2]. Let $V_t \in$

$\mathbb{R}^{n\times n}$ denote the matrix $\mathbf{I}-\rho_t s_t y_t^\top$. By applying M times formula (6.63) with the initial matrix $\mathbf{A}_t^0$ that can vary between each iteration, we obtain that $\mathbf{A}_t$ satisfies the following equation (for $t > M$):

$$\begin{aligned}\mathbf{A}_t = &\left(\mathbf{V}_{t-1}^\top \cdots \mathbf{V}_{t-M}^\top\right) \mathbf{A}_t^0 \left(\mathbf{V}_{t-M} \cdots \mathbf{V}_{t-1}\right)\\ &+ \rho_{t-M}\left(\mathbf{V}_{t-1}^\top \cdots \mathbf{V}_{t-M+1}^\top\right) s_{t-M} s_{t-M}^\top \left(\mathbf{V}_{t-M+1} \cdots \mathbf{V}_{t-1}\right)\\ &+ \rho_{t-M+1}\left(\mathbf{V}_{t-1}^\top \cdots \mathbf{V}_{t-M+2}^\top\right) s_{t-M+1} s_{t-M+1}^\top \left(\mathbf{V}_{t-M+2} \cdots \mathbf{V}_{t-1}\right)\\ &+ \cdots\\ &+ \rho_{t-1} s_{t-1} s_{t-1}^\top.\end{aligned} \tag{6.69}$$

From this expression the well-known two-loop-recursion for efficiently computing $\mathbf{A}_t \nabla f(\theta_t)$ arises. The procedure is shown in Algorithm 6.7.

Algorithm 6.7 Two-loop-recursion

Input: $\nabla f(\theta_t), \mathbf{A}_t^0$
  $q \leftarrow \nabla f(\theta_t)$
  for $i = t-1, t-2, \ldots, t-M$ **do**
    $\alpha_i \leftarrow \rho_i s_i^\top q$
    $q \leftarrow q - \alpha_i y_i$
  end for
  $r \leftarrow \mathbf{A}_t^0 q$
  for $i = t-M, t-M+1, \ldots, t-1$ **do**
    $\beta \leftarrow \rho_i y_i^\top r$
    $r \leftarrow r + s_i(\alpha_i - \beta)$
  end for
Output: $\mathbf{A}_t \nabla f(\theta_t) = r$

It was experimentally shown (e.g. [NW06, BHNS16]) that a small number M between 3 and 20 is sufficient to produce good results. Thus, the computational effort of Algorithm 6.7 of $4Mn + n$ multiplications ($4Mn$ for the loops and n for the multiplication $\mathbf{A}_t^0 q$ in case of diagonal $\mathbf{A}_t^0$) remains small. Furthermore, the storage requirement for the M correction pairs is affordable.

A typical method to initialize $\mathbf{A}_t^0$ is to set $\mathbf{A}_t^0 = \gamma_t \mathbf{I}$ with

$$\gamma_t = \frac{s_{t-1}^\top y_{t-1}}{y_{t-1}^\top y_{t-1}}. \tag{6.70}$$

In [NW06] it is argued that this scaling factor tries to estimate the size of the Hessian matrix along the last search direction. Hence, the new search direction is well-scaled which has the desirable effect that the line search usually accepts the step length $\alpha_t = 1$. We give the algorithm for the L-BFGS method in Algorithm 6.8.

Due to the reduced curvature information that is used in L-BFGS compared to standard BFGS method, the rate of convergence is not superlinear but only linear. The following theorem is proved in [XWW08].

Theorem 6.4. *Let θ_0 be a starting point for which Assumption 6.1 holds. Let $\boldsymbol{A}_0^0$ be any positive definite matrix. Then the sequence $\{\theta_t\}$ generated by Algorithm 6.8 converges R-linearly to the unique minimizer $\theta^* \in \mathcal{L}$, i.e. there exists a constant $0 \le r < 1$ such that*

$$f(\theta_t) - f(\theta^*) \le r^t \left(f(\theta_0) - f(\theta^*)\right). \tag{6.71}$$

6.5.3 Stochastic L-BFGS method

Schraudolph et al. [SYG+07] and Byrd et al. [BHNS16] extended the L-BFGS method to be suitable for machine learning problems with big datasets such as the training of neural

Algorithm 6.8 L-BFGS algorithm

Input: $\theta_0, M > 0$
  for $t = 0, 1, \ldots, T$ **do**
    choose $\mathbf{A}_t^0$
    $p_t \leftarrow -\mathbf{A}_t \nabla f(\theta_t)$ with Algorithm 6.7
    $\theta_{t+1} \leftarrow \theta_t + \alpha_t p_t$
      where α_t is computed by a line search to satisfy the Wolfe conditions (6.51), (6.52)
    if $t \geq M$ **then**
      discard correction pair $\{s_{t-M}, y_{t-M}\}$
    end if
    $s_t \leftarrow \theta_{t+1} - \theta_t$
    $y_t \leftarrow \nabla f(\theta_{t+1}) - \nabla f(\theta_t)$
  end for

networks. In their works, the gradient of the objective function ∇f is evaluated only on a minibatch B – as we have seen it in all the previous algorithms. That is, we will denote the gradient by ∇f_B. We will focus on the variant of the L-BFGS method of Byrd et al. since it is more robust than Schraudolph's proposed algorithm. Besides the stochasticity, there are some more modifications of the just derived version of the L-BFGS method that improve the algorithm's performance on the considered problem class.

In [BHNS16] the correction vector y_t is not computed by the standard formula of taking the difference of the two most recent gradients $\nabla f(\theta_{t+1}) - \nabla f(\theta_t)$. Instead, to overcome potential numerical issues when $\|\theta_{t+1} - \theta_t\|$ is small, this difference is approximated using the first order Taylor series. Hence, y_t can be computed by a Hessian-vector product

$$y_t = \nabla^2 f_{B'}(\theta_{t+1})(\theta_{t+1} - \theta_t). \tag{6.72}$$

Here, the Hessian $\nabla^2 f(\theta_{t+1})$ is also evaluated on a minibatch, but – as indicated by B' – a different one than the gradient, because we found that this is more efficient just like mentioned in Section 6.3 about the Hessian-free method. y_t is computed by Algorithm 4.1.

Because of the stochastic regime it is advisable to collect curvature information over more iterates to update the model. Thus, Byrd et al. do not compute a new correction pair $\{s_t, y_t\}$ at each iteration but only every L iterations. This also reduces the computational cost, since the Hessian-vector products are rather expensive to compute. With this approach, we have two different time schedules. t denotes the total number of iterations that have been executed so far, i.e. the number of performed parameter updates. The newly introduced subscript k denotes the number of correction pairs that have currently been computed. The correction pairs can be defined using the averaged weight parameters of the last L iterations

$$s_k = \bar{\theta}_k - \bar{\theta}_{k-1}, \tag{6.73}$$

$$y_k = \nabla f_{B'}(\bar{\theta}_k) s_k, \tag{6.74}$$

where

$$\bar{\theta}_k = \sum_{i=t-L+1}^{t} \theta_i. \tag{6.75}$$

This implies that there are $2L - 1$ iterations necessary to compute the first correction pair and thus the L-BFGS update cannot be performed until then. Therefore, this method uses the standard stochastic gradient descent update rule for the first $2L$ iterations.

For these updates we use a predefined schedule for the learning rate α as in SGD. Byrd et al. [BHNS16] perform all updates by taking a schedule of decaying learning rates during training

$$\alpha_t = \frac{\beta}{t} \tag{6.76}$$

for some $\beta > 0$. However, our implementation differs from this approach described in [BHNS16]. For the real L-BFGS updates we perform an extended version of the backtracking line search 6.4. If the sufficient decrease condition (6.34) is not satisfied after $r \in \mathbb{N}$ reductions of the step size, the search direction is not likely to be a descent direction, as this is not ensured by the procedure due to stochasticity. Then, we take the negative search direction instead and perform again the backtracking line search for maximal r iterations. If still the sufficient decrease condition cannot be fulfilled, we discard the search direction and make no update step at all. However, mostly only a few iterations in the line search are necessary before a learning rate is accepted because of the well-scaled initialization of $\mathbf{A}_t^0$ mentioned before. This approach turns out to be more effective then the learning rate schedule chosen in [BHNS16].

Another difference between our implementation and the one in [BHNS16] is that we always use the same batch size for B and B'.

We have summarized the stochastic L-BFGS method in Algorithm 6.9.

Algorithm 6.9 Stochastic L-BFGS algorithm

Input: θ_0, M, L, α_t for $t = 0, \dots, 2L$
  $k \leftarrow 0$
  $\bar{\theta}_k \leftarrow 0$
  for $t = 0, 1, \dots, T$ **do**
    choose minibatch B_t
    compute $\nabla f_{B_t}(\theta_t)$
    $\bar{\theta}_k \leftarrow \bar{\theta}_k + \theta_t$
    if $t \leq 2L$ **then**
      $\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla f_{B_t}(\theta_t)$
    else
      $\theta_{t+1} \leftarrow \theta_t - \alpha_t \mathbf{A}_t \nabla f_{B_t}(\theta_t)$
        where $\mathbf{A}_t \nabla f_{B_t}(\theta_t)$ is computed by Algorithm 6.7
        and α_t is computed by a line search
    end if
    if $\mod(t+1, L) = 0$ **then**
      $\bar{\theta}_k \leftarrow \bar{\theta}_k / L$
      if $k > 0$ **then**
        if $k \geq M$ **then**
          discard correction pair $\{s_{k-M}, y_{k-M}\}$
        end if
        choose minibatch B'_t
        $s_k \leftarrow \bar{\theta}_k - \bar{\theta}_{k-1}$
        $y_k \leftarrow \nabla^2 f_{B'_t}(\bar{\theta}_k) s_k$
      end if
      $k \leftarrow k + 1$
      $\bar{\theta}_k \leftarrow 0$
    end if
  end for

The convergence analysis in [BHNS16] is again only applicable to convex functions. We will still give the main result, as at least close to a minimum the loss function that we use, is assumed to be approximately convex.

The assumptions made in [BHNS16] are given as follows.

Assumption 6.3.

1. *The objective function $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable.*

2. *There exist positive constants λ and Λ such that*

$$\lambda\|z\|^2 \leq z^\top \nabla^2 f_B(\theta) z \leq \Lambda\|z\|^2 \tag{6.77}$$

for all $\theta \in \mathbb{R}^n$ and all batches $B \subseteq \{1, \ldots, N\}$.

3. *There exists a constant γ such that, for all $\theta \in \mathbb{R}^n$,*

$$\|\nabla f(\theta)\|^2 \leq \gamma^2. \tag{6.78}$$

These assumptions ensure in particular that f is strongly convex and thus has a unique minimizer θ^*. The following convergence theorem is based on the step size choice (6.76) which is a special case of the more general assumptions $\sum \alpha_t = \infty$, $\sum \alpha_t^2 < \infty$.

Theorem 6.5. *Suppose that Assumptions 6.3 hold. Let $\{\theta_t\}$ be the sequence generated by Algorithm 6.9, where*

$$\mu_1\|z\|^2 \leq z^\top \boldsymbol{A}_t z \leq \mu_2\|z\|^2 \tag{6.79}$$

holds for constants $0 < \mu_1 \leq \mu_2$, all $z \in \mathbb{R}^n$ and all $t = 0, 1, \ldots$. Furthermore, let the step sizes α_t be given as

$$\alpha_t = \frac{\beta}{t} \qquad with \quad \beta > \frac{1}{2\mu_1\lambda}. \tag{6.80}$$

Then, for all $t \geq 1$

$$f(\theta_t) - f(\theta^*) \leq \frac{Q(\beta)}{t} \tag{6.81}$$

where

$$Q(\beta) = \max\left\{\frac{\Lambda\mu_2^2\beta^2\gamma^2}{2(2\mu_1\lambda\beta - 1)}, f(\theta_0) - f(\theta^*)\right\}. \tag{6.82}$$

Thus, for ideal properties of the objective function we have a sublinear rate of convergence. With the line search, a better estimation of the loss offset can be achieved, but we want to emphasize that these theoretical convergence properties cannot be realized in practice, as the loss function of neural networks is not convex, let alone strongly convex. Therefore, we do not give the proof, and evaluate the convergence rate only experimentally.

Chapter 7

Experiments

To test and evaluate the algorithms described in Chapter 6 on deep learning problems, we implemented them in Caffe [JSD+14]. This framework is written in C++, and designed to exploit the network structure to parallelize computations on the GPU by using Nvidia's CUDA parallel computing platform and cuDNN, a library designed especially for deep neural networks. It accelerates the training significantly. The drawback is that GPUs do not provide as much memory as CPUs. Thus, the size of the minibatches as well as the number of correction pairs in the stochastic L-BFGS algorithm are limited. Both of these limitations restrict the performance of the algorithms.

For the tests we used different datasets and network architectures to compare the optimization methods on problems with different number of parameters. The datasets are summarized in Table 7.1. We compare the three second order optimization methods Hessian-free, Scaled Conjugate Gradient and stochastic L-BFGS as well as standard SGD with momentum. The momentum term is in all experiments set to 0.9. If not otherwise stated we apply ℓ_2-regularization to the weights with a regularization parameter $\lambda = 0.0005$.

Dataset	**Image size**	**# training images**	**# test images**	**# classes**
MNIST	28×28	60000	10000	10
SVHN	$32 \times 32 \times 3$	73257	26032	10
Supermarket	$227 \times 227 \times 3$	408 (original) 31008 (augmented)	4769	20

Table 7.1: Datasets used for the experiments.

When referring to the standard settings of the second order optimization methods we mean the configurations listed in Table 7.2.

Solver	**Curvature matrix**	**Setting**
Hessian-free	Gauss-Newton	preconditioned, 50 inner iterations
SCG	Gauss-Newton	10 inner iterations
LBFGS	Hessian	$L = 0.25$ epochs, $M = 10$

Table 7.2: Standard settings for second order methods.

7.1 MNIST

A common benchmark dataset in image classification is the MNIST dataset of handwritten digits, described by LeCun et al. [LBBH98]. It consists of grayscale images of size 28×28, organized in a training set of 60000 examples and a test set of 10000 examples. The digits are size normalized to fit into a 20×20 box, which is centered in the image. Examples from the dataset are shown in Figure 7.6a. We trained two different network architectures on this dataset: one fully-connected and one convolutional network.

7.1.1 Multilayer perceptron

For the first experiment we used a multilayer perceptron (MLP). This is a fully-connected network, which we equipped with two hidden layers and ReLUs as activation functions. The concrete architecture is listed in Table 7.3. In total it has 406528 parameters. In order to

Layer type	Input size	# neurons	# parameters
fc	28×28	512	401408
ReLU	1×512	512	
fc	1×512	10	5120
ReLU	1×10	10	
softmax	1×10	10	

Table 7.3: Architecture of the MLP.

compare the different optimization methods we trained on the dataset for 20 epochs with a minibatch size of 1024.

For SGD we used an initial learning rate of 0.001, which is reduced after every 8 epochs by a factor of 0.5. The Hessian-free method was set to use the Gauss-Newton matrix, and we set the maximum number of inner CG-iterations to 50. On average, 15 CG-iterations were executed in each outer iteration. We did not apply preconditioning. For the stochastic LBFGS method we set the learning rate for the first iterations to 10^{-5}, stored 20 correction pairs, which were updated every 50 iterations. We ran the SCG method with 10 inner iterations on each minibatch and used the Gauss-Newton matrix.

The best achieved error rates are shown in Table 7.4 and the convergence plots are shown in Figure 7.1.

Considering the error rates on the training and test set the second order methods outperform SGD significantly. It is apparent that SCG rather overfits the training data. For this method the loss decreases very quickly with the best convergence rate and thus achieves the lowest training error. However, the test error is worse than that of Hessian-free and LBFGS. In contrast, the stochastic LBFGS method seems to have superior generalization properties. Its test error rate is the best of all algorithms, while its training error is higher than that of Hessian-free and SCG. The gradient norm achieved by LBFGS is even larger than that of SGD. The performance of the Hessian-free method is in between the overfitting SCG and the excellent generalizing LBFGS. Concerning training time LBFGS outperforms SCG and especially Hessian-free considerably. It is only slightly slower than SGD. We compare the run time of all four algorithms in Figure 7.2a. The experiments were executed on a Nvidia Titan X (Pascal) GPU.

	MLP		LeNet-5	
Opt. method	**Train error** (%)	**Test error** (%)	**Train error** (%)	**Test error** (%)
SGD	3.075	4.72	0.331	1.00
Hessian-free	0.067	3.16	0.088	0.68
LBFGS	0.565	2.66	0.142	0.63
SCG	0.003	3.30	0.047	0.92

Table 7.4: Error rates of MLP and LeNet-5 on MNIST.

In Figure 7.3 we compare the different performances of SCG with the Hessian matrix and Gauss-Newton matrix respectively. The Gauss-Newton matrix gives slightly better results. However, we see that the number of inner iterations is of more importance for the performance. With 10 inner SCG iterations on one minibatch the rate of convergence is of an higher order than for the case of only 1 SCG iteration per batch. The drawback of performing more inner iterations is that naturally the time consumption increases. Although the results are better with more SCG iterations per minibatch, one should not increase this number further, since then the overfitting becomes more and more pronounced.

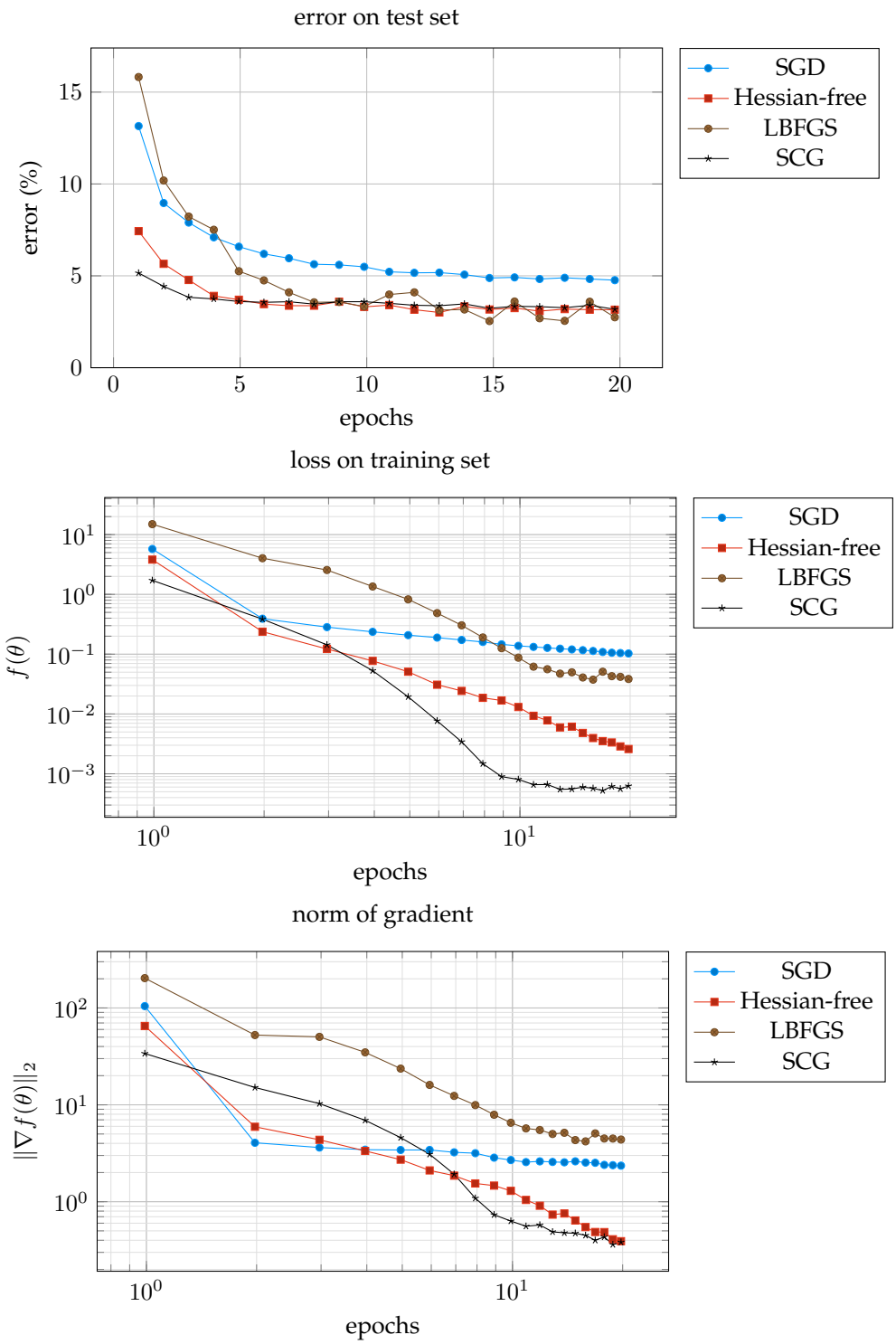
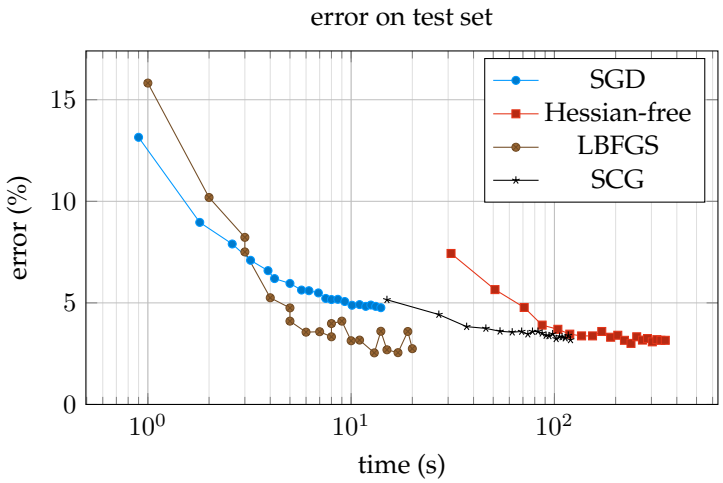
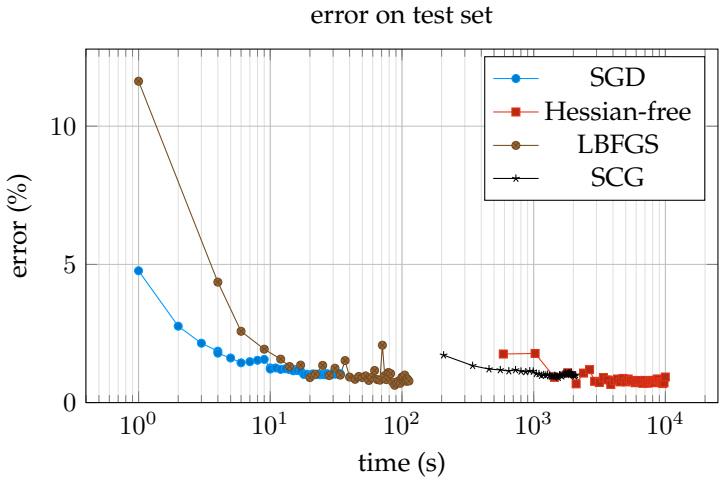


Figure 7.1: Convergence plots of training MLP on MNIST. The minibatch size was chosen to be 1024 for all solvers. Loss and norm of gradient are averaged over one epoch. Hessian-free performed maximum 50 inner CG-iterations, SCG performed 10 inner iterations, LBFGS computed every 50 iterations a new correction pair and 20 pairs were used. All experiments were identically initialized.



(a) MLP



(b) LeNet-5

Figure 7.2: Evolution of the test error over run time of the algorithms applied to training MLP and LeNet-5 on MNIST. Figure 7.2a shows the run time of 20 epochs of training MLP on MNIST starting at the first epoch. Figure 7.2b shows the run time of 40 epochs of training LeNet-5 on MNIST starting at the first epoch.

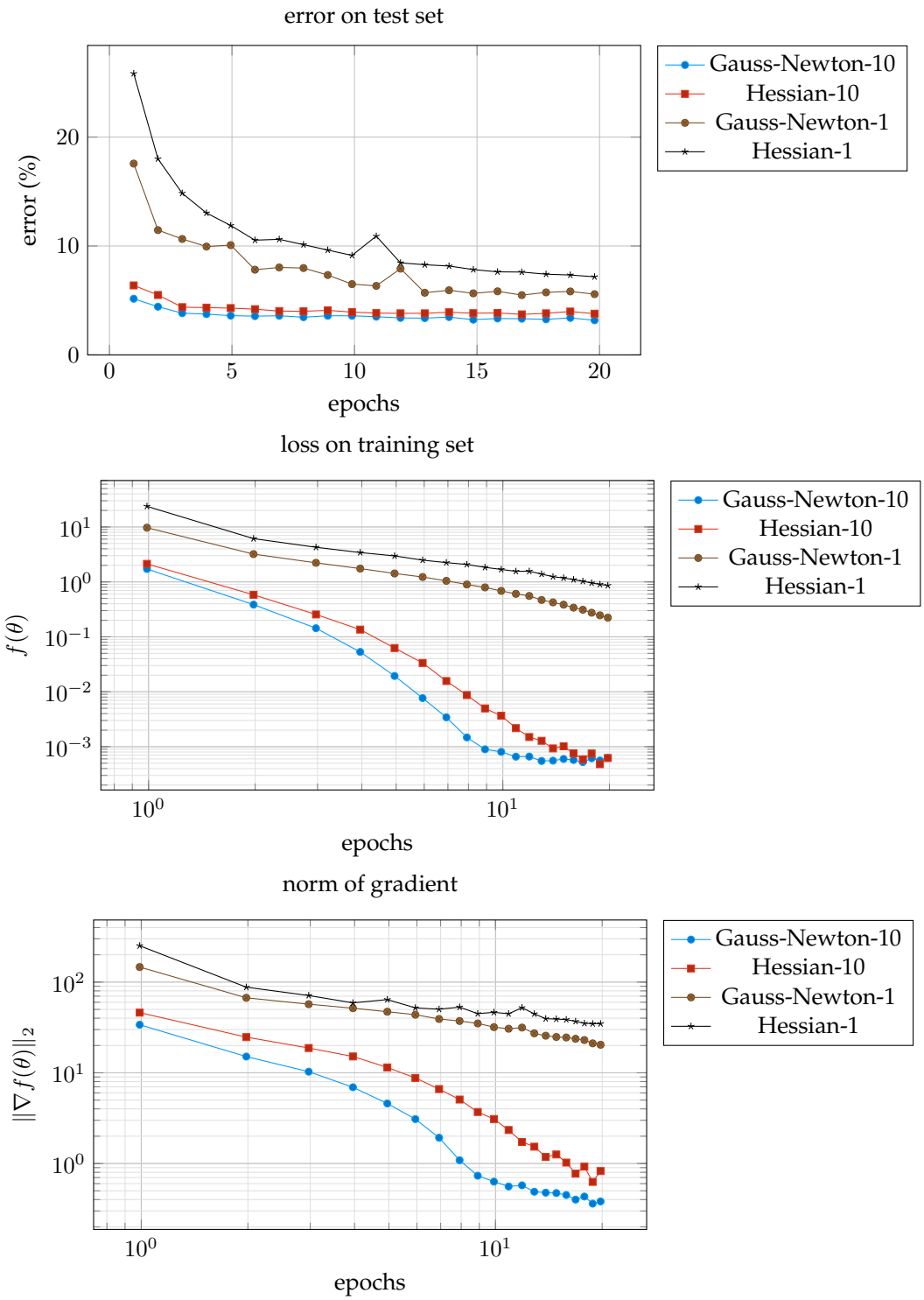


Figure 7.3: Comparison of Gauss-Newton matrix and Hessian matrix with different number of inner iterations in the SCG method applied to training MLP on MNIST.

7.1.2 LeNet-5

To demonstrate the superior classification ability of convolutional networks, we also trained LeNet-5 architecture (see Table 7.5) on the MNIST dataset. This architecture was proposed by LeCun et al. in 1998 [LBBH98]. The input data is multiplied by the factor $1/255$ to scale the pixel values to the interval $[0, 1]$. LeNet-5 has in total 431500 parameters, which is only slightly more than the MLP. However, as we see in Table 7.4, the CNN achieves significantly better results.

Layer type	Input size	Kernel size	Stride	Output size	# parameters
conv	$28 \times 28 \times 1$	5×5	1	$24 \times 24 \times 20$	1500
pool (max)	$24 \times 24 \times 20$	2×2	2	$12 \times 12 \times 20$	
conv	$12 \times 12 \times 20$	5×5	1	$8 \times 8 \times 50$	25000
pool (max)	$8 \times 8 \times 50$	2×2	2	$4 \times 4 \times 50$	
fc	$4 \times 4 \times 50$			$1 \times 1 \times 500$	400000
ReLU	$1 \times 1 \times 500$			$1 \times 1 \times 500$	
fc	$1 \times 1 \times 500$			$1 \times 1 \times 10$	5000
softmax	$1 \times 1 \times 10$			$1 \times 1 \times 10$	

Table 7.5: Architecture of LeNet-5 for MNIST.

The settings, that we used for the optimization methods are as follows: SGD was initialized with a learning rate of 0.01, that we bisect after every 10 epochs. LBFGS was started with a learning rate of 0.001, we used 10 correction pairs for the update and computed a new correction pair after every quarter epoch. The SCG method performed 20 inner iterations and uses the Gauss-Newton matrix. The Hessian-free algorithm uses Gauss-Newton as well and performs at maximum 50 inner CG-iterations. We used a batch size of 512 in all experiments. The convergence plots for these experiments are shown in Figure 7.4. On this rather small convolutional neural network all three of the second order type methods outperform SGD with momentum. They achieve better error rates on both the training and the test set. Again, SCG tends to overfitting, whereas LBFGS reaches the best test error without reducing the loss as much as SCG and Hessian-free. Hessian-free method executed on average 18 inner CG-iterations per minibatch. The discrepancy of run time for the different algorithms becomes even more obvious here as we trained LeNet-5 for 40 epochs. It is shown in Figure 7.2b. The experiments were executed on a Nvidia Titan X (Pascal) GPU.

With this experiment, we also want to demonstrate the differences in performance of the Hessian-free method between using the Gauss-Newton matrix and the Hessian matrix. Apart from changing the curvature matrix, we left all other settings in the experiments identical. The plots are shown in Figure 7.5. We see that the Gauss-Newton matrix is significantly superior to the Hessian matrix. First of all the Hessian-free method with Gauss-Newton reaches a better test error rate and the error is less oscillating throughout the training. Secondly, the loss decreases with a higher order of convergence rate. This implies that the Hessian matrix is not positive definite most of the time and the damping parameter has to be increased, which makes the update steps smaller. Besides, the line search method that used the Gauss-Newton method chose at almost every iteration the step length 1, whereas the update direction computed by the Hessian matrix is badly scaled and often a small step size has to be chosen to guarantee a loss reduction. Another indicator for the bad performance of Hessian-free method with the Hessian matrix is that the norm of the gradient is increasing rather than decreasing.

Furthermore, we want to mention that the differences in performance of the Hessian-free method are only marginal between applying the preconditioned CG algorithm compared to the standard CG algorithm. Also considering the run time these two methods do not differ much from each other. The additional computations of the preconditioning are compensated by the line search, as it tends to choose larger step sizes for the preconditioned version.

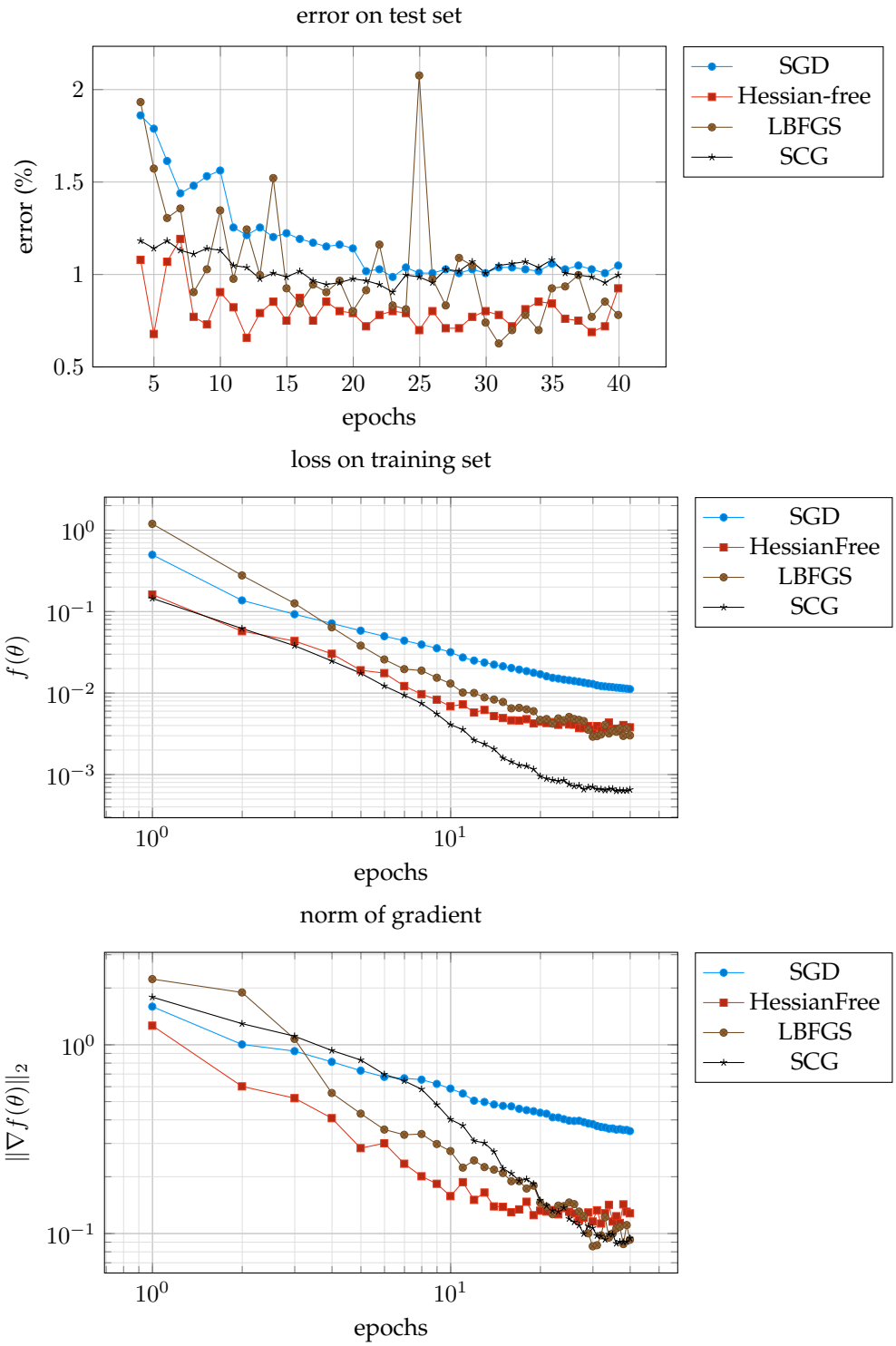


Figure 7.4: Convergence plots of training LeNet-5 on MNIST. The minibatch size was chosen to be 512 for all solvers. Loss and norm of gradient are averaged over one epoch. Hessian-free performed maximum 50 inner CG-iterations, SCG performed 20 inner iterations, LBFGS computed every 29 iterations a new correction pair and 10 pairs were used. All experiments were identically initialized.

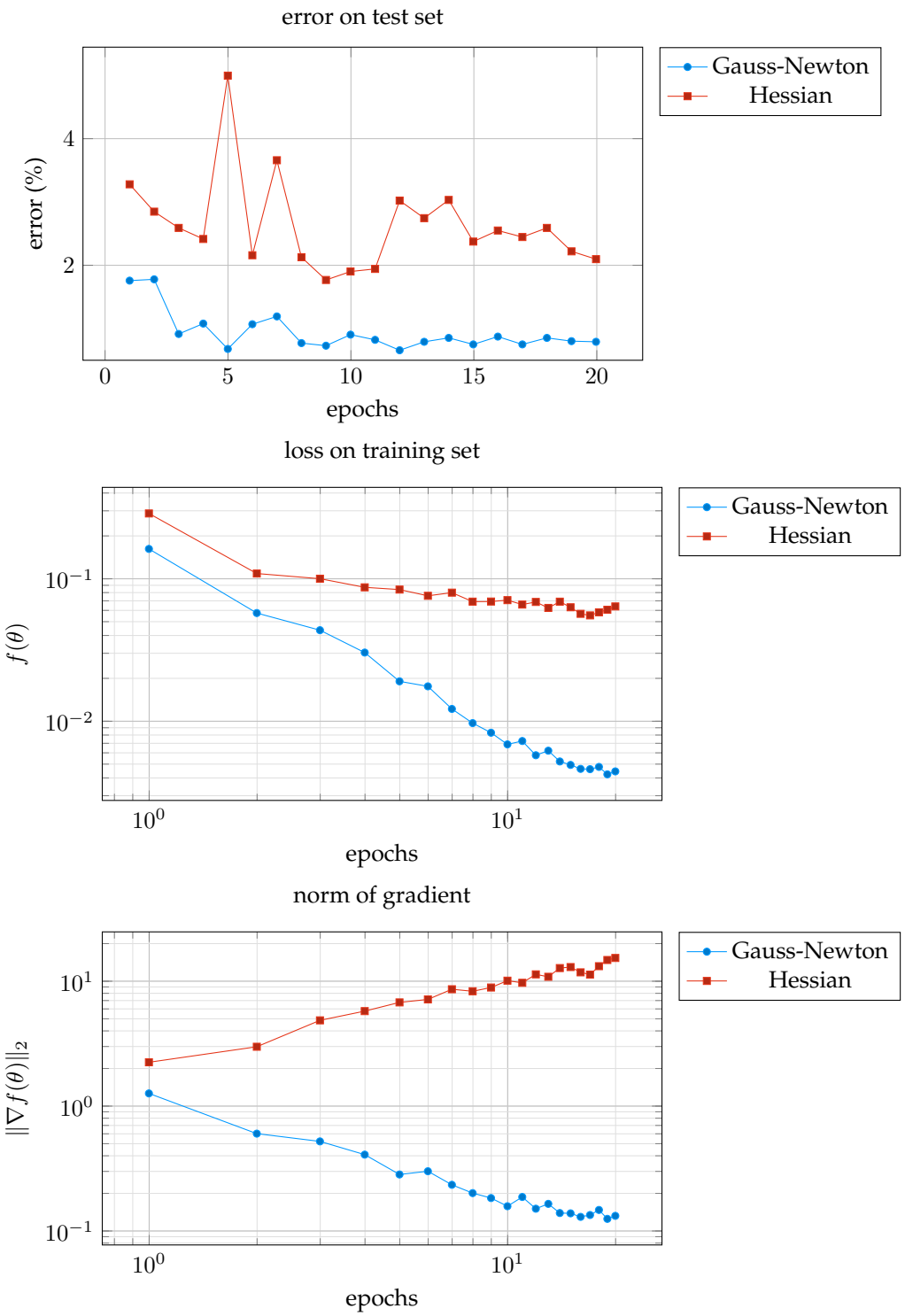


Figure 7.5: Comparison of Hessian-free method with Gauss-Newton and with Hessian matrix applied to training LeNet-5 on MNIST. Both experiments were identically initialized, preconditioning was applied and maximum 50 inner CG-iterations were executed.

7.2 SVHN

The SVHN (street view house numbers) dataset [NWC+11] contains RGB color images of size $32 \times 32 \times 3$. We used 73257 images for training and 26032 for testing. Just like MNIST the SVHN dataset consists of 10 classes. However, the digits are neither centered nor size normalized. Examples from the dataset are shown in Figure 7.6b. Since some of the images contain more than one digit, this dataset is much more difficult to train on than MNIST. As network architecture we also used LeNet-5. Since the images in the SVHN are of a different size than MNIST, the dimensions of the data for the different layers in LeNet-5 are also different from those listed in Table 7.5. We also increased the number of neurons in the first fully-connected layer. The details are shown in Table 7.6. This LeNet-5 architecture has in total $3\ 897\ 220$ parameters. When feeding the SVHN data into the network, we scale it by the factor $1/255$ to squash the pixel values into the interval $[0, 1]$.

(a) MNIST (b) SVHN

Figure 7.6: Examples from datasets for digit recognition.

Layer type	Input size	Kernel size	Stride	Output size	# parameters
conv	$32 \times 32 \times 3$	5×5	1	$28 \times 28 \times 20$	1500
pool (max)	$28 \times 28 \times 20$	2×2	2	$14 \times 14 \times 20$	
conv	$14 \times 14 \times 20$	5×5	1	$10 \times 10 \times 50$	25 000
pool (max)	$10 \times 10 \times 50$	2×2	2	$5 \times 5 \times 50$	
fc	$5 \times 5 \times 50$			$1 \times 1 \times 3072$	3 840 000
ReLU	$1 \times 1 \times 3072$			$1 \times 1 \times 3072$	
fc	$1 \times 1 \times 3072$			$1 \times 1 \times 10$	30 720
softmax	$1 \times 1 \times 10$			$1 \times 1 \times 10$	

Table 7.6: Architecture of LeNet-5 for SVHN.

We trained the network on SVHN with a minibatch size of 512 for 50 epochs. SGD was initialized with a learning rate of 0.01 that was reduced by a factor 0.5 after every 10 epochs. LBFGS was also started with a learning rate of 0.01 for the first gradient steps. A new correction pair was computed after every epoch and we used at maximum 5 correction pairs, i.e. curvature information of the last 5 epochs. The settings for the Hessian-free method were the same as before: PCG with Gauss-Newton matrix and at most 50 iterations per minibatch. The SCG algorithm was run with the Gauss-Newton matrix and 10 iterations per minibatch. The convergence plots are shown in Figure 7.7 and the best achieved error rates in Table 7.7.

	$\lambda = 0.0005$		$\lambda = 0.005$	
Opt. method	**Train error (%)**	**Test error (%)**	**Train error (%)**	**Test error (%)**
SGD	1.578	11.82	5.631	11.40
Hessian-free	0.333	10.64	12.542	14.83
LBFGS	0.025	10.86	6.959	11.05
SCG	0.060	11.71	0.344	11.582

Table 7.7: Error rates of LeNet-5 on SVHN.

Furthermore, we tested the influence of the regularization parameter λ for the weight decay. We ran the experiments with the exact same settings for the solvers again with the changed value for λ of 0.005. It turns out that the regularization parameter does not affect much the resulting classification accuracy for all methods, except for the Hessian-free. However, the error on the training dataset is larger with a larger λ, that is overfitting is less prominent. The curves for errors and loss behave similar as in Figure 7.7 with the difference that the loss value is larger because of the larger regularization term. The error rates are given in Table 7.7.

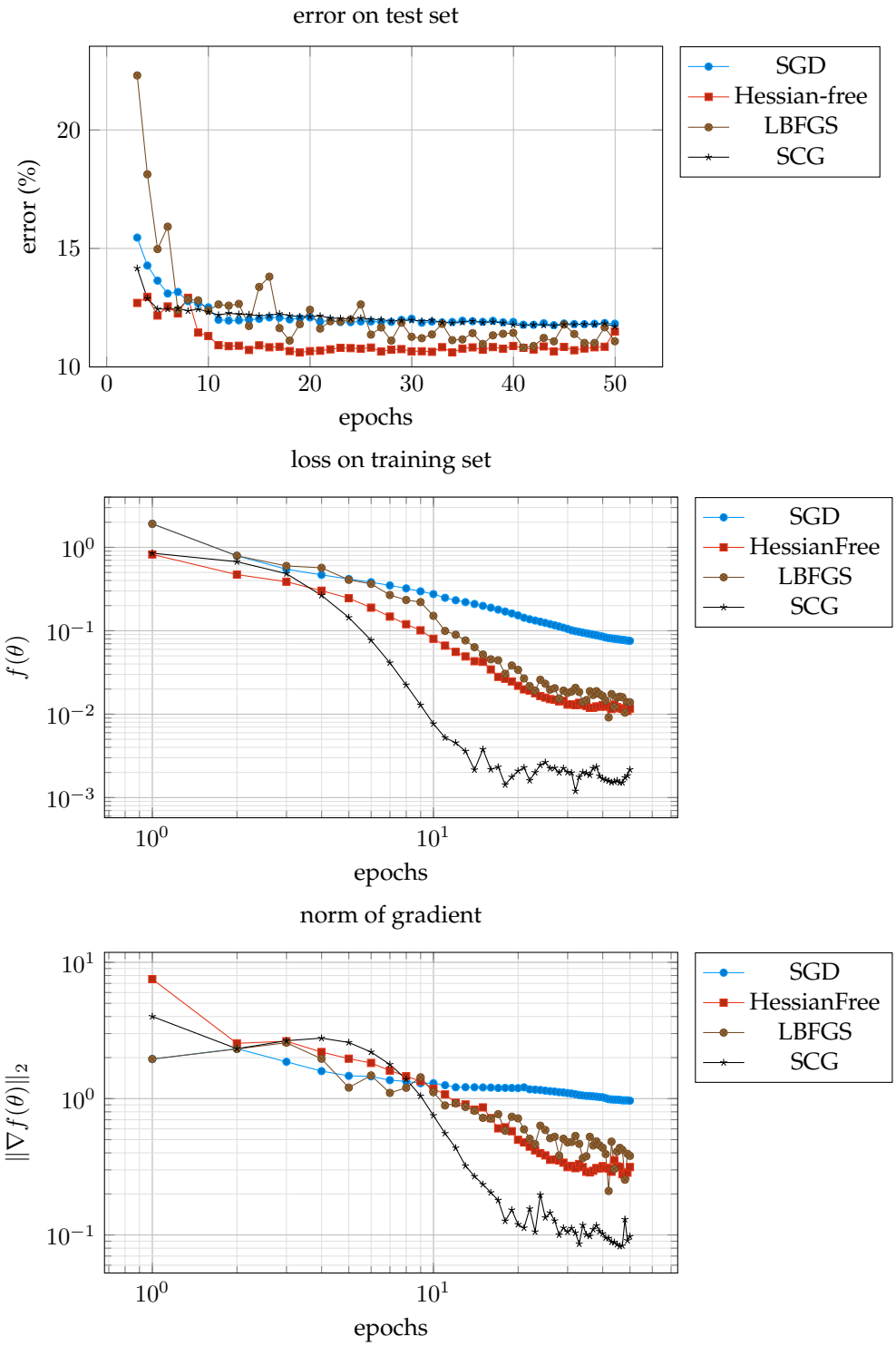


Figure 7.7: Convergence plots of training LeNet-5 on SVHN. The minibatch size was chosen to be 512 for all solvers. Loss and norm of gradient are averaged over one epoch. Hessian-free performed maximum 50 inner CG-iterations, SCG performed 10 inner iterations, LBFGS computed every 143 iterations a new correction pair and 5 pairs were used. All experiments were identically initialized.

7.3 Supermarket data

The third dataset, on which we performed experiments, consists of labeled images of groceries from a supermarket. It was provided by MVTec. The dataset contains 20 classes of RGB color images of size $227 \times 227 \times 3$. The original training dataset consists of only 408 images that show the objects from different directions. Examples are shown in Figure 7.8. The test dataset, however, contains 4769 images where the objects can be scaled, occluded, cluttered, rotated, blurred or varied in brightness. To achieve better classification results, we artificially augmented the original training dataset by imitating the transformations that occur in the test set. On that account for each training image the object is segmented and undergoes affine or perspective transformations or occlusion. Global or local illumination is added to the image and the background color is changed. Thus, we eventually end up with 31008 training images. Figure 7.9 illustrates the result of these transformations on example images.

Figure 7.8: Examples from the original supermarket dataset.

Figure 7.9: Examples from the augmented supermarket dataset.

We trained on both the original and the augmented dataset the network CaffeNet[1] which is detailed in Table 7.8. This network is very similar to Krizhevsky's AlexNet [KSH12]. In total, this architecture has 58 353 696 parameters. There is some stochasticity introduced to this network as two Dropout layers with probability 0.5 are present. We preprocessed the input data to the network by subtracting the mean of all training images.

We first trained the network on the original dataset with a minibatch size of 32 for 40 epochs. SGD was initialized with a learning rate of 0.001, which we reduced after every 10 epochs. LBFGS also has initial learning rate 0.001 for the gradient steps. We used 5 correction pairs, which were updated after every epoch. The Hessian-free and SCG method used the standard settings from Table 7.2. The error rates are shown in Table 7.9 and convergence plots in Figure 7.11. Because of the small amount of training data which is not representative for the test data, all the results are according to expectations not satisfying. Especially, the Hessian-free algorithm does not converge to a good minimum. The LBFGS algorithm, however, demonstrates here again its great generalization properties. The error on the training set is three times as large as for SGD, but the errors on the test set do not differ much for both solver methods. The best result is achieved by SCG algorithm. It actually reaches an

[1] https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet

Layer type	Input size	Kernel size	Stride/Pad	Output size	# parameters
conv	$227 \times 227 \times 3$	11×11	$4/0$	$55 \times 55 \times 96$	34 848
ReLU	$55 \times 55 \times 96$			$55 \times 55 \times 96$	
pool (max)	$55 \times 55 \times 96$	3×3	$2/0$	$27 \times 27 \times 96$	
conv	$27 \times 27 \times 96$	5×5	$1/2$	$27 \times 27 \times 256$	614 400
ReLU	$27 \times 27 \times 256$			$27 \times 27 \times 256$	
pool (max)	$27 \times 27 \times 256$	3×3	$2/0$	$13 \times 13 \times 256$	
conv	$13 \times 13 \times 256$	3×3	$1/1$	$13 \times 13 \times 384$	884 736
ReLU	$13 \times 13 \times 384$			$13 \times 13 \times 384$	
conv	$13 \times 13 \times 384$	3×3	$1/1$	$13 \times 13 \times 384$	1 327 104
ReLU	$13 \times 13 \times 384$			$13 \times 13 \times 384$	
conv	$13 \times 13 \times 384$	3×3	$1/1$	$13 \times 13 \times 256$	884 736
ReLU	$13 \times 13 \times 256$			$13 \times 13 \times 256$	
pool (max)	$13 \times 13 \times 256$	3×3	$2/0$	$6 \times 6 \times 256$	
fc	$6 \times 6 \times 256$			$1 \times 1 \times 4096$	37 748 736
ReLU	$1 \times 1 \times 4096$			$1 \times 1 \times 4096$	
Dropout (0.5)	$1 \times 1 \times 4096$			$1 \times 1 \times 4096$	
fc	$1 \times 1 \times 4096$			$1 \times 1 \times 4096$	16 777 216
ReLU	$1 \times 1 \times 4096$			$1 \times 1 \times 4096$	
Dropout (0.5)	$1 \times 1 \times 4096$			$1 \times 1 \times 4096$	
fc	$1 \times 1 \times 4096$			$1 \times 1 \times 20$	81 920
softmax	$1 \times 1 \times 20$			$1 \times 1 \times 20$	

Table 7.8: Architecture of CaffeNet for supermarket data.

accuracy of 100% on the training dataset. Despite this desolate overfitting, half of the test data is classified correctly by the resulting network parameters.

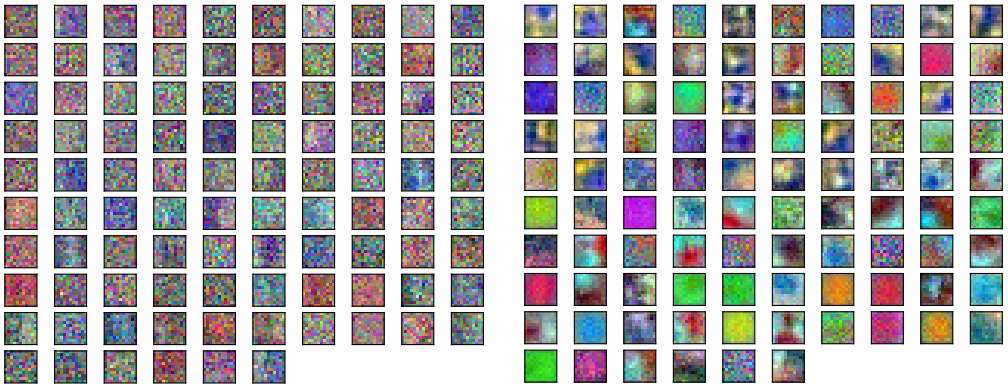
	original		**augmented**	
Opt. method	**Train error** $(\%)$	**Test error** $(\%)$	**Train error** $(\%)$	**Test error** $(\%)$
SGD	4.412	63.37	0.010	15.31
Hessian-free	34.314	76.85	2.670	21.85
LBFGS	12.010	65.19	0.290	15.96
SCG	0.000	50.30	0.126	14.59

Table 7.9: Error rates of CaffeNet on original and augmented supermarket data.

The training on the augmented data was run for 50 epochs with a minibatch size of 64. The settings for all solvers are identical to the training on the original data: SGD was initialized with a learning rate of 0.001, which was divided in half after every 10 epochs. LBFGS was started with gradient steps with learning rate 0.001, stored at most 5 correction pairs, which were updated every epoch. Hessian-free method used the Gauss-Newton matrix, preconditioning and performed at most 50 inner CG-iterations. The SCG algorithm used the Gauss-Newton matrix as well and performed 10 inner iterations per minibatch. The results are shown in Figure 7.12 and Table 7.9.

In Figure 7.10 the resulting filter masks of the first convolutional layer of training CaffeNet with SCG on the original and augmented supermarket data are shown as RGB images. For the original dataset the filters are very noisy, whereas with more training data clearer edges and patterns are formed which are more suitable to extract useful information from the images. This, in turn, results in a better classification accuracy.

With this large network and large images the performance of the second order methods was not in general superior to the first order method SGD with momentum. Especially, the Hessian-free method is not competitive with the other solvers. The rate of convergence is



(a) original dataset (b) augmented dataset

Figure 7.10: Trained filters of the first convolutional layer of CaffeNet achieved by SCG. Figure 7.10a shows the filters after training CaffeNet on the original supermarket data, and Figure 7.10b on the augmented dataset.

very low and additionally it was over 20 times slower than the SGD method in the experiment with the augmented data, where Hessian-free executed on average 14 CG-iterations per minibatch. Presumably with growing depth of the network, the nonconvexity of the resulting optimization problem increases. However, SCG seems to be able to handle the large problem size and nonconvexity better. It reaches the best error on both datasets. In particular, with the small dataset the result on the test set is significantly better than those of the other solvers. The drawback is, that the training data is boldly overfitted by SCG. The LBFGS method achieves very similar results as SGD, with a tendency to better generalization, as the error on the training dataset is still larger.

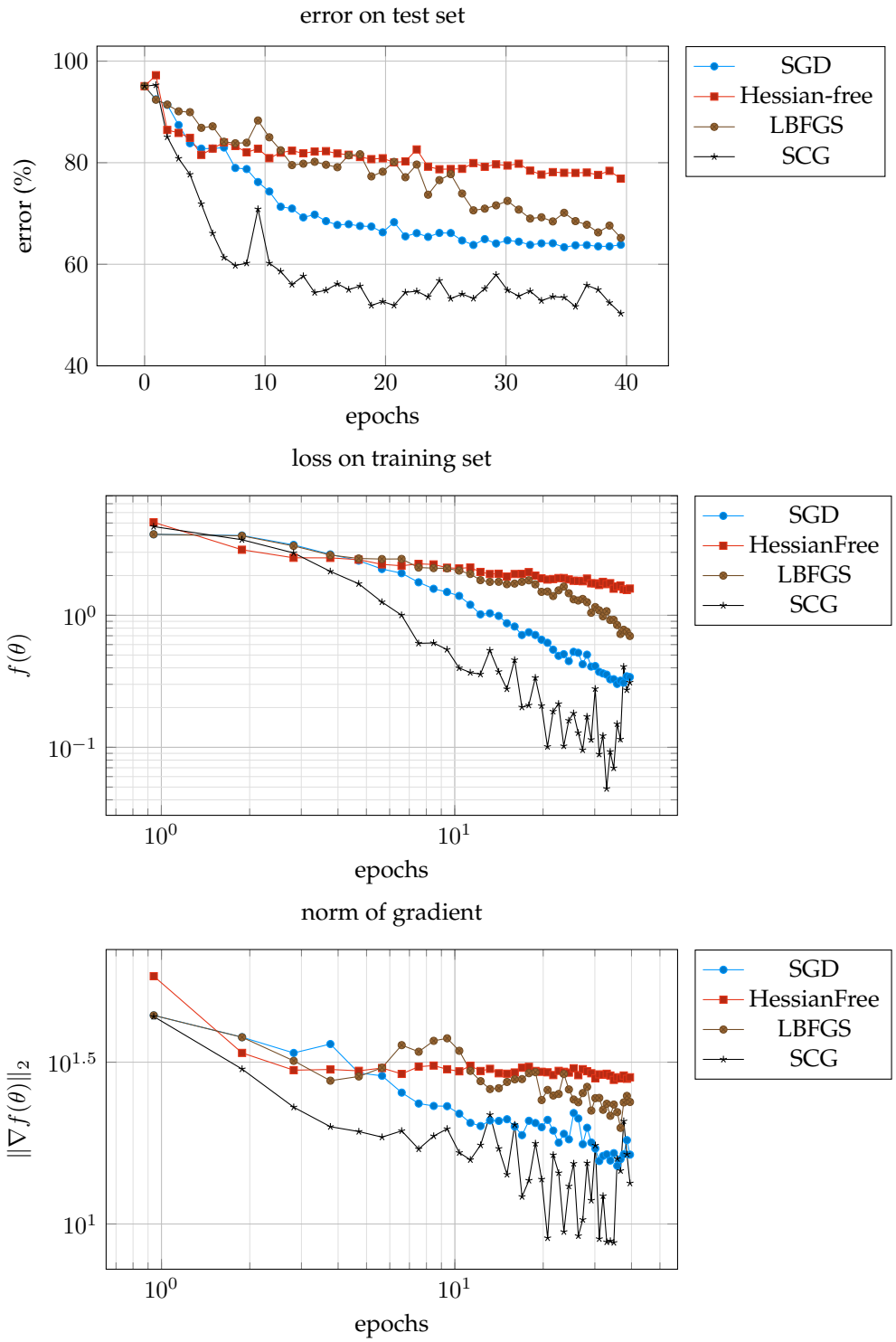


Figure 7.11: Convergence plots of training CaffeNet on original supermarket data. The minibatch size was chosen to be 32 for all solvers. Loss and norm of gradient are averaged over one epoch. Hessian-free performed maximum 50 inner CG-iterations, SCG performed 10 inner iterations, LBFGS computed every 12 iterations a new correction pair and 5 pairs were used. All experiments were identically initialized.

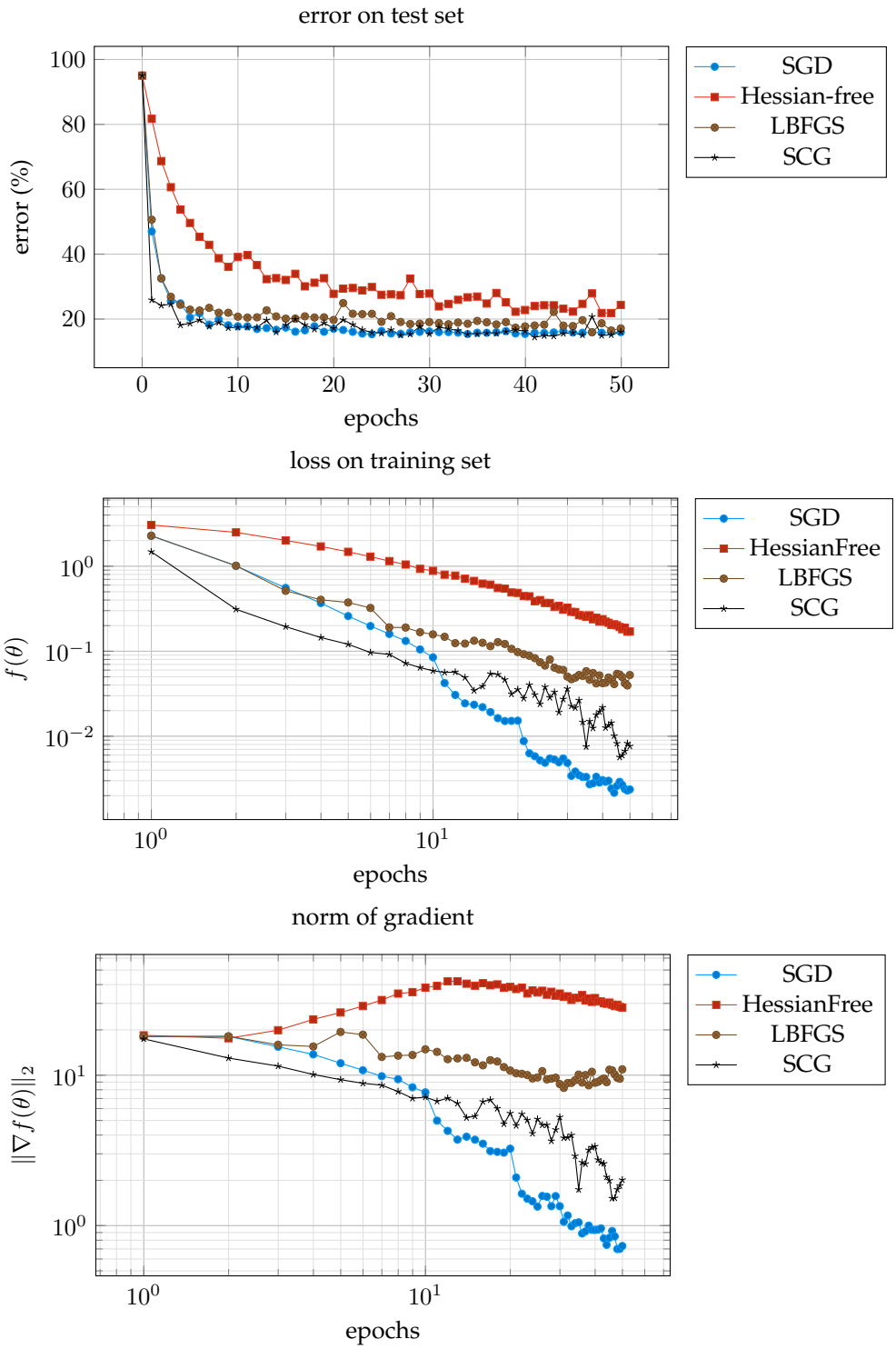


Figure 7.12: Convergence plots of training CaffeNet on augmented supermarket data. The minibatch size was chosen to be 64 for all solvers. Loss and norm of gradient are averaged over one epoch. Hessian-free performed maximum 50 inner CG-iterations, SCG performed 10 inner iterations, LBFGS computed every 484 iterations a new correction pair and 5 pairs were used. All experiments were identically initialized.

7.4 Bayesian optimization

We applied the Bayesian optimization method described by Snoek et al. [SLA12] to selected experiments. This method is implemented in the package Spearmint and is designed to optimize hyperparameters of algorithms.

7.4.1 Training LeNet-5 with SCG on MNIST

For training LeNet-5 on MNIST with SCG we ran a Bayesian optimization method in order to find the best hyperparameters. We fixed that SCG should use the Gauss-Newton matrix and trained the net for 20 epochs. We optimized the minibatch size N_B, the number of inner SCG iterations T and the regularization parameter λ for weight decay. We allowed the following ranges:

$$N_B \in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048\}, \tag{7.1}$$

$$T \in \{1, 2, \ldots, 20\}, \tag{7.2}$$

$$\lambda \in [10^{-5}, 10^{-2}]. \tag{7.3}$$

The Bayesian optimization was performed by training the network with 100 different combinations of these parameters. The results show that the best error rates are achieved with the smallest allowed regularization parameter $\lambda = 10^{-5}$. In combination with the minibatch size, this parameter seems to have the most influence on the results. The number of inner iterations is not as important for the performance. In Figure 7.13 some of the best experiments are shown. There is one experiment for which the evolution of the loss is completely different from all other experiments. It has the parameters $N_B = 32,\ T = 1,\ \lambda = 10^{-5}$. Despite of this unusual convergence of the loss, this experiment reaches with an error on the test set of 0.73% one of the best results of all experiments. The experiment with $N_B = 128,\ T = 14,\ \lambda = 0.003828$ is an instance of the average loss of all experiments. In Table 7.10 the error rates on the train and test set for the plotted example experiments are listed.

Parameters	Train error (%)	Test error (%)
$N_B = 1024, T = 20, \lambda = 10^{-5}$	0.087	0.93
$N_B = 512, T = 17, \lambda = 10^{-5}$	0.042	0.94
$N_B = 64, T = 6, \lambda = 10^{-5}$	0.128	0.91
$N_B = 32, T = 1, \lambda = 10^{-5}$	0.037	0.73
$N_B = 128, T = 14, \lambda = 0.003828$	0.372	0.92

Table 7.10: Error rates of selected experiments of the Bayesian hyperparameter optimization of training LeNet-5 on MNIST with SCG.

7.4.2 Training CaffeNet with SCG on original supermarket data

We furthermore applied the Bayesian hyperparameter optimization to the problem of training CaffeNet with SCG on the original supermarket dataset. The same parameters as in Section 7.4.1 were optimized with the predefined ranges of

$$N_B \in \{16, 32, 64, 128\}, \tag{7.4}$$

$$T \in \{1, 2, \ldots, 20\}, \tag{7.5}$$

$$\lambda \in [10^{-5}, 10^{-2}]. \tag{7.6}$$

The configuration that reached the best test error was $N_B = 16,\ T = 14$ and $\lambda = 0.01$. The error on the test set is 46.34% and on the training set 1.716%. Because of the larger regularization parameter the overfitting which we have seen in Section 7.3 could be resolved.

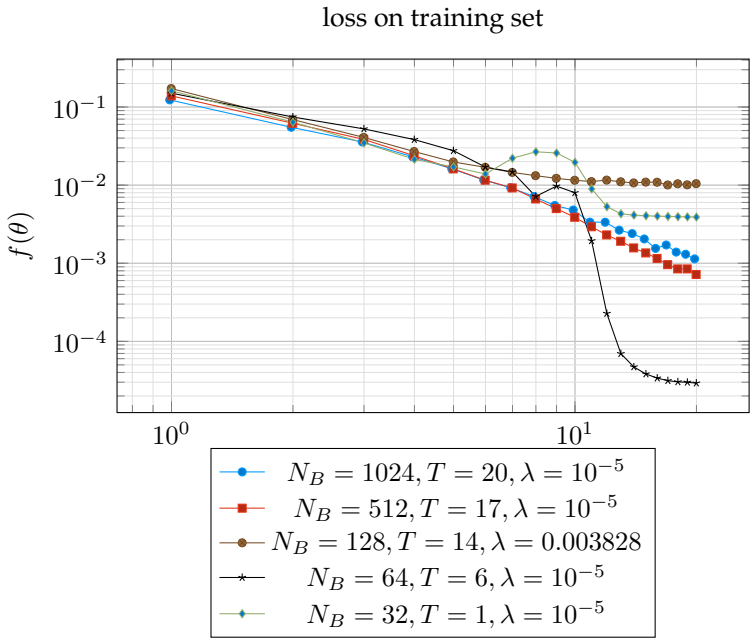


Figure 7.13: The best results of Bayesian hyperparameter optimization applied to training LeNet-5 on MNIST with SCG for different minibatch sizes and an instance of an average result.

The variation in these experiments is very large. It is impossible to recognize a pattern of parameter configuration, such as high or low regularization, that always produces good results.

7.4.3 Training LeNet-5 with stochastic LBFGS on SVHN

For the experiment of training LeNet-5 with stochastic LBFGS on the SVHN dataset we also ran a Bayesian optimization of the hyperparameters. We optimized the size of the minibatches N_B, the base learning rate α for the first gradient descent steps, as well as the parameters L and M that specify after how many iterations a new correction pair is computed and how many correction pairs are stored to be used in the two-loop-recursion respectively. The possible values were chosen as follows:

$$N_B \in \{16, 32, 64, 128, 256, 512, 1024, 2048, 4096\}, \tag{7.7}$$
$$\alpha \in [10^{-4}, 10^{-1}], \tag{7.8}$$
$$L \in \{1, 2, \ldots, 1024\}, \tag{7.9}$$
$$M \in \{1, 2, \ldots, 20\}. \tag{7.10}$$

We trained the network for 20 epochs with 100 different parameter combinations in order to optimize them. Interestingly, one of the best results was achieved with a small batch size of 16 examples, one correction pair and an interval of 1024 iterations between two computations of correction pairs. The chosen learning rate was 10^{-4}. That is, this parameter setting lies on the corner of the hypercube defined by the intervals (7.7), (7.8), (7.9) and (7.10). Comparing all experiments, it becomes obvious that for a base learning rate that is too large, the problem diverges in the first gradient descent steps which cannot be compensated by the LBFGS method afterwards. For a vast majority of all other parameter settings, the method converges to local minima for which the errors on the test set lie in a range of around 5 percentage points. In the plot of the loss function over the iterations we see that most configurations result in the same order of convergence and a similar final loss value. However,

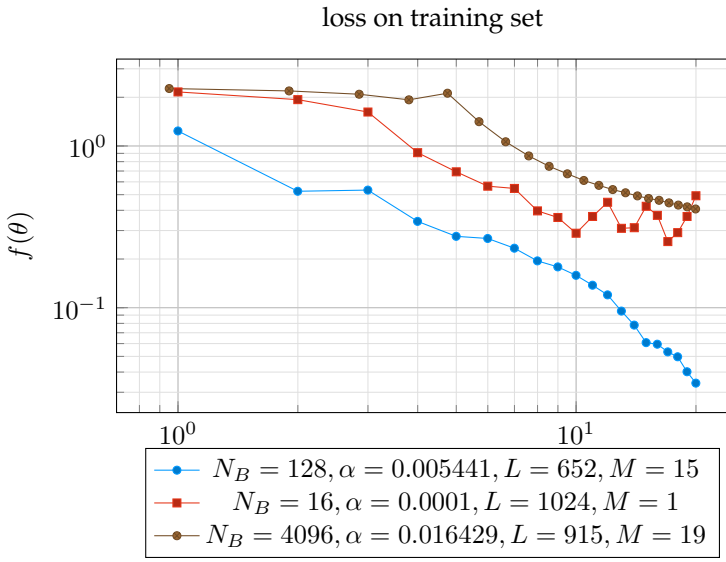


Figure 7.14: The best results of Bayesian hyperparameter optimization applied to training LeNet-5 on SVHN with LBFGS for different parameter configurations.

there are some experiments that demonstrate a higher order of convergence. In Table 7.11 and Figure 7.14 we compare one of these experiments, the one with the best, not overfitted, error rate and one experiment that shows an average plot.

Parameters	**Train error (%)**	**Test error (%)**
$N_B = 128, \alpha = 0.005441, L = 652, M = 15$	1.881	11.17
$N_B = 16, \alpha = 0.0001, L = 1024, M = 1$	5.623	11.57
$N_B = 4096, \alpha = 0.016429, L = 915, M = 19$	10.79	13.54

Table 7.11: Error rates of selected experiments of the Bayesian hyperparameter optimization of training LeNet-5 on SVHN with LBFGS.

Chapter 8

Conclusion

In this thesis we considered the problem of training convolutional neural networks in the context of image classification for computer vision. For this purpose we examined second order optimization methods as alternative to the standard first order gradient type methods to minimize the loss function. In particular, we considered three second order type methods: Hessian-free method, which is a truncated Newton approach; scaled conjugate gradient method, which adapts the concept of the conjugate gradient algorithm to nonconvex objective functions; stochastic L-BFGS method, which is a quasi-Newton type method adapted to large-scale optimization problems. The advantage of second order type methods is the improved theoretical speed of convergence. Alas, convergence analysis only applies to convex objective functions. However, the error function of neural networks, especially convolutional neural networks, is nonconvex. Therefore, we also experimented with replacing the possibly indefinite Hessian matrix of the objective function by the positive semidefinite Gauss-Newton matrix to compute second order information. In order to evaluate whether second order optimization methods are superior to first order methods in practice, we implemented the three second order type methods in the Caffe-framework [JSD+14] and performed numerical experiments on different datasets and network architectures. Furthermore, we compared the second order type methods to SGD with momentum to evaluate the trade-off between faster theoretical convergence and higher computational per-iteration-cost of the second order methods.

The experiments have shown that the performance of the different optimization methods strongly depends on the specific problem.

The Hessian-free method was implemented as described in [Mar10]. It does not need much tuning of the hyperparameters. There is not much improvement by changing the standard settings, that we give in Table 7.2. However, the Hessian-free method converges very slowly when applied to deep networks such as CaffeNet. In addition, typically more CG-iterations per minibatch are necessary to converge than the SCG method needs for good performance. This results in a very long training time of Hessian-free which is why in practice this method is not recommendable. The experiments have shown that the usage of the Gauss-Newton matrix to approximate the true Hessian is significantly more robust and outperforms the method that uses the Hessian matrix clearly.

For the implementation of the stochastic LBFGS method we followed [BHNS16]. However, rather than applying a fixed learning rate schedule, we perform a line search to compute the step size. The stochastic LBFGS method produces good results when applied to problems with few parameters, as in the MNIST experiments in Section 7.1. For such problems this optimization method is better than the other two second order methods we tested because it is much faster and produces superior results. However, with increasing number of parameters the stochastic LBFGS algorithm cannot compete with the stochastic gradient descent. In our experiments with CaffeNet we could not increase the number of correction pairs because of the limited memory on the GPU. Maybe rising this number would stabilize the algorithm. For smaller problems however, the advantage of LBFGS over SGD lies in the greater robustness with respect to its hyperparameters. Apart from the initial learning rate, which is an issue of SGD, LBFGS produces acceptable results for all problems with the standard settings from Table 7.2. All experiments have shown that the stochastic LBFGS

algorithm is the method that has the best generalization property based on the approach of taking the information of multiple batches into account in order to compute a correction pair. To improve the LBFGS method Gower et al. proposed a stochastic block BFGS method [GGR16] that uses more curvature information than the standard BFGS method. This could improve both rate of convergence and stability of the algorithm with respect to larger problems. However, it consumes more memory since more than one Hessian-vector products are used which is why we considered it as less relevant in practice as stochastic LBFGS.

We implemented the SCG method following [Møl93] and [Nab02]. To adapt it to the large training dataset we applied the minibatch concept and run the algorithm only for a few number of iterations on each minibatch. The SCG method proves to be the best second order optimization method for large problem classes. It outperforms both LBFGS and Hessian-free, as well as SGD, significantly in the experiments on the supermarket data. Like we have seen for the Hessian-free method, choosing the Gauss-Newton matrix instead of the Hessian to compute curvature information, ends up in better results and makes the algorithm more robust. A flaw of SCG is that it tends to overfit the training data, which is a consequence of performing several iterations on one single minibatch. However, the generalization error is usually still small. One possibility to overcome the overfitting could be to increase the regularization parameter for weight decay, as we experimented in Sections 7.4.1 and 7.4.2.

Considering the run time of all algorithms, the first order method SGD is clearly superior to the second order methods. In particular the Hessian-free method takes up a lot of time without giving better results. The stochastic LBFGS method is however not much slower than SGD. Besides for shallow networks it outperforms SGD, and is therefore preferable. SCG is already significantly slower than both SGD and LBFGS. Since it can achieve better results than SGD and LBFGS, without much tweaking of hyperparameters it is according to our experiments the best choice of algorithm, though.

The outcomes of the Bayesian hyperparameter optimization suggest that the second order solvers SCG and stochastic LBFGS are rather insensitive to modifications of the hyperparameters when applied to a small network such as LeNet-5. In particular the LBFGS method is very robust, as long as the initial learning rate is not too large. However, it can be chosen defensively small and the method can still converge to an acceptable minimum. The fact that for both SCG and LBFGS some parameter configurations decrease the loss much faster and farther, can be an indication that a deep neural network has some local minima that have a very small objective value but are hard to find. Baldassi et al. [BBC$^+$16] give theoretical and experimental evidence that the geometry of network loss functions shows rare, but very dense regions of local minima with a good generalization error. These regions are rather flat compared to the otherwise very rough landscape of the graph. It is however very challenging to find these local minima. We conclude from the results in Section 7.4.2 that for deeper networks the geometry of the network loss function is increasingly complex where the regions of well generalizing local minima are excessively difficult to be found.

As we have seen in the experiments the stochastic LBFGS algorithm does the best job to find well generalizing local minima because of averaging techniques. A similar approach is the idea of stochastic variance reduced gradients (SVRG) presented by Johnson and Zhang [JZ13] which computes an averaged gradient on all training data. Moritz et al. [MNJ15] combine SVRG with the stochastic LBFGS method, which we consider in this work, proposed by Byrd et al. [BHNS16]. The authors could prove a linear convergence rate of the resulting algorithm for strongly convex and smooth objective functions.

In conclusion we found that the second order methods are superior to SGD especially in shallow networks. In these cases the usage of curvature information can give better results. With growing depth of the network, however, the objective function becomes more complex and presumably nonconvex in larger domains which is why the second order information is not as useful in these cases. Thus, future research topics would include to find algorithms that can handle the nonconvexity and stochastic regime of the optimization problem in a robust way by guaranteeing convergence rates for this problem class.

List of Algorithms

List of Figures

List of Tables

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[BBC+16] Carlo Baldassi, Christian Borgs, Jennifer T Chayes, Alessandro Ingrosso, Carlo Lucibello, Luca Saglietti, and Riccardo Zecchina. Unreasonable effectiveness of learning neural networks: From accessible states and robust ensembles to basic algorithmic schemes. *Proceedings of the National Academy of Sciences*, 113(48):E7655–E7662, 2016.

[BCNN11] Richard H Byrd, Gillian M Chin, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[BEHW87] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

[BHNS16] Richard H Byrd, SL Hansen, Jorge Nocedal, and Yoram Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[Bis06] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[Bou14] Jake Bouvrie. Notes on convolutional neural networks. Technical report, 2014.

[BPRS15] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

[CHH02] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[CS16] Nadav Cohen and Amnon Shashua. Convolutional rectifier networks as generalized tensor decompositions. *arXiv preprint arXiv:1603.00162*, 2016.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

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

[DKM+95] Rodney J Douglas, Christof Koch, Misha Mahowald, Kevan AC Martin, and Humbert H Suarez. Recurrent excitation in neocortical circuits. *Science*, 269(5226):981, 1995.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS*, pages 315–323, 2011.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[GGR16] Robert M Gower, Donald Goldfarb, and Peter Richtárik. Stochastic block bfgs: Squeezing more curvature out of data. *arXiv preprint arXiv:1603.09649*, 2016.

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

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

[HV15] Benjamin D Haeffele and René Vidal. Global optimality in tensor factorization, deep learning, and beyond. *arXiv preprint arXiv:1506.07540*, 2015.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[JZ13] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KH91] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems 4*, pages 950–957, 1991.

[Kri07] David Kriesel. A brief introduction on neural networks, 2007.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBOM12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[Mar10] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

[Mar16] James Martens. *Second-order Optimization for Neural Networks*. PhD thesis, University of Toronto, 2016.

[MNJ15] Philipp Moritz, Robert Nishihara, and Michael I Jordan. A linearly-convergent stochastic l-bfgs algorithm. *arXiv preprint arXiv:1508.02087*, 2015.

[Møl93] Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

[MYSM+90] Shlomo Mor-Yosef, Arnon Samueloff, Baruch Modan, Daniel Navot, and Joseph G Schenker. Ranking the risk factors for cesarean: logistic regression analysis of a nationwide study. *Obstetrics & Gynecology*, 75(6):944–947, 1990.

[Nab02] Ian Nabney. *NETLAB: algorithms for pattern recognition*. Springer, 2002.

[Nas82] Stephen G Nash. Truncated-newton methods. Technical report, DTIC Document, 1982.

[Nel16] Rick Nelson. From hephaestus' automatons to openai's deep learning. *EE-Evaluation Engineering*, 55(2):2–3, 2016.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[NW06] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer, 2006.

[NWC+11] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, 2011.

[Pea94] Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1):147–160, 1994.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[RM99] Ian H Robertson and Jaap MJ Murre. Rehabilitation of brain damage: brain plasticity and principles of guided recovery. *Psychological bulletin*, 125(5):544, 1999.

[Sch02] Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[SHK+14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[SMDH13] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning. *ICML (3)*, 28:1139–1147, 2013.

[SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[SYG+07] Nicol N Schraudolph, Jin Yu, Simon Günter, et al. A stochastic quasi-newton method for online convex optimization. In *AISTATS*, volume 7, pages 436–443, 2007.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[WZZ+13] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[XWW08] Yunhai Xiao, Zengxin Wei, and Zhiguo Wang. A limited memory bfgs-type method for large-scale unconstrained optimization. *Computers & Mathematics with Applications*, 56(4):1001–1009, 2008.

[XWW+16] Lingxi Xie, Jingdong Wang, Zhen Wei, Meng Wang, and Qi Tian. Disturblabel: Regularizing cnn on the loss layer. *arXiv preprint arXiv:1605.00055*, 2016.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.