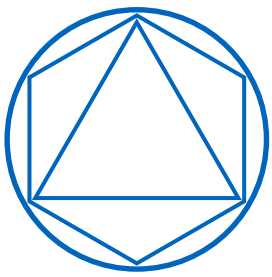
Mathematics in Science and Engineering

Technical University of Munich

Master's Thesis

Application of second-order optimisation for large-scale deep learning

Julian Suk



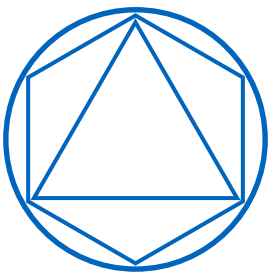
Mathematics in Science and Engineering

Technical University of Munich

Master's Thesis

Application of second-order optimisation for large-scale deep learning

Author:	Julian Suk
1st examiner:	Prof. Dr. Hans-Joachim Bungartz
Assistant advisor:	M. Sc. Severin Reiz
Submission Date:	April 15th, 2020



I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

April 15th, 2020 Julian Suk

Acknowledgments

This thesis is dedicated to my grandmother Hedwig Suk and my father Thomas Suk, both of whom passed away last year.

"Vor lauter Globalisierung und Computerisierung dürfen die schönen Dinge des Lebens wie Kartoffeln oder Eintopf kochen nicht zu kurz kommen."

- Angela Merkel

Abstract

Deep neural networks have become some of the most prominent models in machine learning due to their flexibility and therefore, their broad applicability. The training of large-scale deep neural networks requires vast computational resources. Stochastic gradient descent methods still enjoy great popularity but Hessian-based optimisation techniques are on the rise. While computing the second derivative of the loss function is still computationally expensive, a possibly much faster convergence rate justifies the consideration of such methods. Gradient descent is inherently sequential and cannot take full advantage of highly parallelised computing architectures. This motivates the exploration of second-order optimisation methods also in the context of high performance computing. This thesis aims to provide an overview of numerical challenges, their solutions and stochastic details regarding the application of Hessian-based optimisation to the training of large-scale deep neural networks. It lays emphasis on a strong theoretical foundation, which is crucial for the less heuristic second-order methods. The potential of a quasi-Newton method is showcased by outperforming gradient descent in optimisation of the loss function corresponding to ResNet.

Contents

Part I.

Introduction

Background and significance

The promise of artificial intelligence (AI) through machine learning has been one of the greatest scientific endeavours of the decade. Machine learning is comprised of mathematical and statistical methods to find an underlying, latent representation in a set of data. In particular the field of deep learning has been growing rapidly. The key idea is to construct a mathematical model, inspired by neuron connections in the human brain. It is for this reason that these models are referred to as neural networks. The term "deep" is to contrast with earlier shallow architectures that have gradually been replaced by multi-layer ones. With this advancement come issues regarding the training of these models: the deeper the architecture, the higher the number of training parameters. A popular example is ResNet (residual network) [15], that won the 2015 ImageNet large-scale visual recognition competition (ILSVRC 2015) [36] and boasts a total of $23,944,392$ trainable parameters (50-layer layout).

Despite this challenge, deep neural networks are heavily requested in fields like computer vision, biomedicine and optimal control. They are used to e.g. detect cancer metastases [21], predict lung cancer [51] or in superhuman chess AIs [39]. This demand establishes the base for rapidly evolving research in mathematical optimisation: Deep neural networks are trained by minimising a cost function that quantifies the prediction error of their underlying mathematical model. The large scale of those networks creates a lot of challenges in optimisation. In particular, the loss landscape is generally non-convex and thus, finding a global optimum is difficult or impossible.

The go-to training algorithm for deep neural networks is the fairly heuristic stochastic gradient descent (SGD), which sacrifices optimisation performance for the sake of straightforward applicability and little computational work. However, in order to better understand these previously black-box models, curvature information about the loss landscape is beneficial. Applications include convergence analysis and robustness experiments. As computational power increases, demand for optimisation strategies with higher convergence rates grows. This establishes the foundation for research in second-order optimisation making use of Hessian information w.r.t. the cost function.

Outline of this work

The purpose of this work is to provide a guideline for the successful implementation of second-order optimisation suitable for large-scale deep learning based on strong theoretical foundations. Therefore, connections are emphasised to the relevant mathematical fields of numerical analysis and stochastic wherever possible. This thesis aims to answer the following questions:

- Which role does mathematical optimisation play in neural network training and which elements helped and are necessary for the successful application to large-scale systems? (part II)

- Which assumptions and approximations are necessary in the high-dimensional domain and how does an exemplary implementation look like? (part III)

Furthermore, this thesis provides results from numerical experiments to substantiate the established theory (part IV).

Part II.

Basics of neural network training

1. Deep neural networks

Deep neural networks are a field that has become the centre of attention for scientists and engineers across many fields like computer vision, control theory and natural language processing. The main workhorse of this subsection of artificial intelligence is mathematical optimisation. In the following chapter, deep neural networks will be described in a way that lends itself to mathematical optimisation.

Layers and activation

Neural networks can be regarded as a non-linear relation, that maps a data sample $x \in S$, where S denotes the sample space, onto a prediction p. For the popular field of image classification, x is an image and S may be the space of all 28×28 pixels greyscale images $S = \{x_{i,j} \in \mathbb{N} | 0 \leq x_{i,j} \leq 255; i = 1, 2 \ldots, 28; j = 1, 2, \ldots, 28\}$. A prediction usually assigns values to each of l classes between which the neural network is supposed to distinguish. In practice, this is done in vector form $p \in \mathbb{R}^n$. A neural network is comprised of so-called layers that each perform some sort of mathematical manipulation. "Deep" in deep neural networks stands for stacking of multiple layers. The most basic example is a fully connected layer, which performs a right multiplication of a vectorised input signal $x^{\text{in}} = (x_1, x_2, \ldots, x_a)$ with filter matrix $W \in \mathbb{R}^{a \times b}$. In order to restrict the entries of the resulting product to moderate values, it is run through an activation function $\phi : \mathbb{R}^b \to \mathbb{R}^b$. One choice is the entry-wise processing through the popular sigmoid function

$$\hat{\phi}_{\text{sigm}}(t) = \frac{1}{2}\left(1 + \tanh\left(\frac{t}{2}\right)\right)$$

which is given in its scalar form here. The output x^{out} of the fully-connected layer is then

$$x^{\text{out}} = \phi(x^{\text{in}} W) \tag{1.1}$$

In recent years, convolutional layers rose to popularity due to their unparalleled performance in image classification. They consist of the convolution of a two-dimensional (or more for multichannel formats) input signal with a filter matrix (or multidimensional tensor). Writing down the mathematical manipulation in symbolic form like (1.1) is not practical in this case, since it involves defining a new integration measure or extensive entry-wise summations and offers little compared to a black-box or graph-based approach. In classification, a neural network is supposed to output a prediction based on the input data,

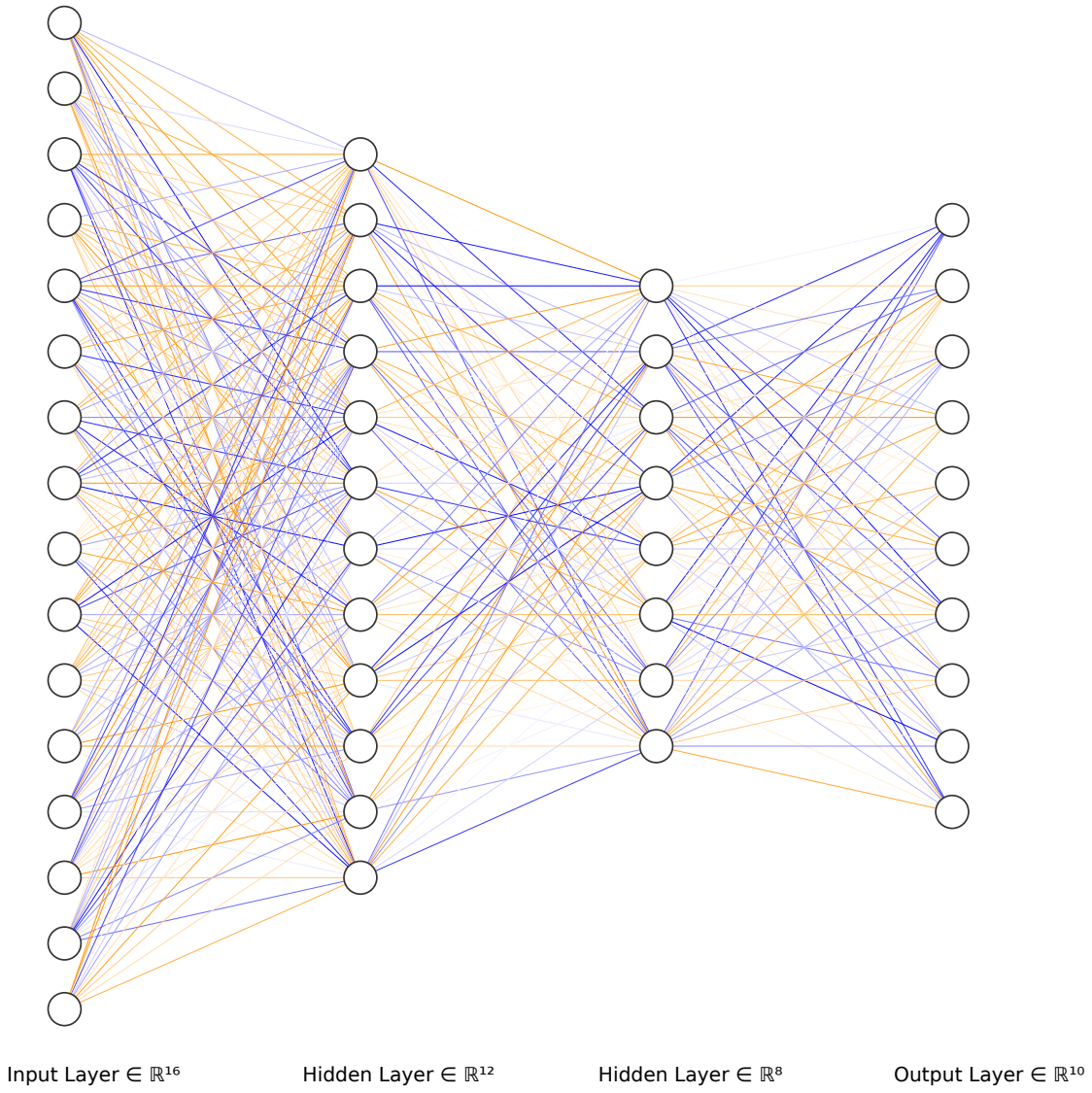


Figure 1.1.: Graph-based visualisation of a fully connected neural network with ten classes, 16 input neurons and two hidden layers. Coloured edges illustrate a hypothetical weight configuration (blue for negative, orange for positive, opacity proportional to magnitude)

i.e. provide probabilities for each class. This is achieved by using softmax activation

$$\phi_{\text{softmax}}(x^{\text{in}})_i = \frac{\mathrm{e}^{(x^{\text{in}})_i}}{\sum_j \mathrm{e}^{(x^{\text{in}})_j}}$$

after the last layer right before the network output. The result will be a vector of probabilities, i.e. values in the interval $(0, 1)$ that sum to one. Note that softmax activation is a smooth ($\mathcal{C}^\infty$) approximation to the $\arg\max$ function.

For the purpose of this work, it makes sense to add a level of abstraction and consider the neural network with all its layers as a non-linear function $f : S \times \mathbb{R}^n \to \mathbb{R}^l$ that takes a sample $x \in S$ and a configuration of n filter entries, referred to as weights $w \in \mathbb{R}^n$ and outputs a prediction p:

$$f(x, w) = p$$

The input data x is usually fixed and the degrees of freedom for model fitting are the entries of vector w.

Loss function

The goal of neural network training is to find a weight configuration w that enables the model to accomplish a certain task, e.g. correct classification of an image. To this end, an objective has to be formulated in form of a loss function. Popular examples are the mean squared error (MSE) between the model output and a reference y

$$c_{\text{MSE}}(x, y, w) = \frac{1}{l}(f(x, w) - y)^{\mathsf{T}}(f(x, w) - y) = \frac{1}{l}\sum_{i=1}^{l}(f_i - y_i)^2$$

or categorical cross-entropy

$$c_{\text{entr}}(x, y, w) = -\sum_{i=1}^{l} y_i \log(f_i)$$

The objective of neural network training is to minimise this loss. In the following, $y \in L$ is referred to as a label and L as the label space. In the context of image classification, it is usually a one-hot encoded vector $y = (y_1, y_2, \ldots, y_l)$ where $y_i = 1$ to indicate the correct class and $y_i = 0$ otherwise.

For the most part of this work, it suffices to add another level of abstraction and "hide" the neural network in the training cost $c(x, y, w)$. Building upon this implicit representation, it might make sense in the context of network training to input more than one sample $x \in S$, but rather $X \in S^d$, i.e. d samples at once (with corresponding labels $Y \in L^d$). As a consequence, the neural network has to produce d predictions instead of one. While this is cumbersome to write down symbolically, implementation is typically trivial, because the aforementioned loss functions contain a reduce-type sum and can thus just process the additional model outputs additively.

2. Optimisation

Consider a loss function $c : S^d \times L^d \times \mathbb{R}^n \to \mathbb{R}$ assigned to a neural network. This function takes d samples from sample space S with corresponding labels from label space L and maps them to a cost value (hence c) that depends on a number of n weights. If the cost attains a minimum, optimal prediction performance is expected. This leads to the following formal problem statement:

$$\min_{w \in \mathbb{R}^n} c(X, Y, w)$$

with X and Y fixed. It is an unconstrained minimisation problem. For the optimisation, the first two arguments of $c(X, Y, w)$ play no role, thus they will be dropped in the further discussion for readability. The reader is referred to [46][30] for a complete introduction to mathematical optimisation which inspired the following.

2.1. Optimality conditions

For the application in neural networks, $c(w)$ can be constructed (continuously) differentiable, e.g. by only using the building blocks discussed in the last chapter. Since they work well for many modern deep learning problems, it is fair to assume $c(w) \in \mathcal{C}^\infty(\mathbb{R}^n)$ for the sake of the following discussion. A condition is now given that characterises any local minimum.

Theorem 2.1 (Necessary optimality condition)**.** *Let $c(w)$ be differentiable on $\mathbb{R}^n$ and $w^* \in K$ a local minimiser of $c(w)$ on an open set $K \subset \mathbb{R}^n$. Then it holds*

$$\nabla_w c(w^*) = 0$$

Proof. For arbitrary $d \in \mathbb{R}^n$ and sufficiently small $t > 0$, the differential quotient

$$\frac{c(w^* + td) - c(w^*)}{t} \geq 0$$

Letting $t \to 0$ yields

$$\nabla_w c(w^*)^\mathsf{T} d \geq 0$$

By choosing $d = -\nabla_w c(w^*)$ follows the claim. □

Conversely, it can be useful to establish a condition that implies a local minimum.

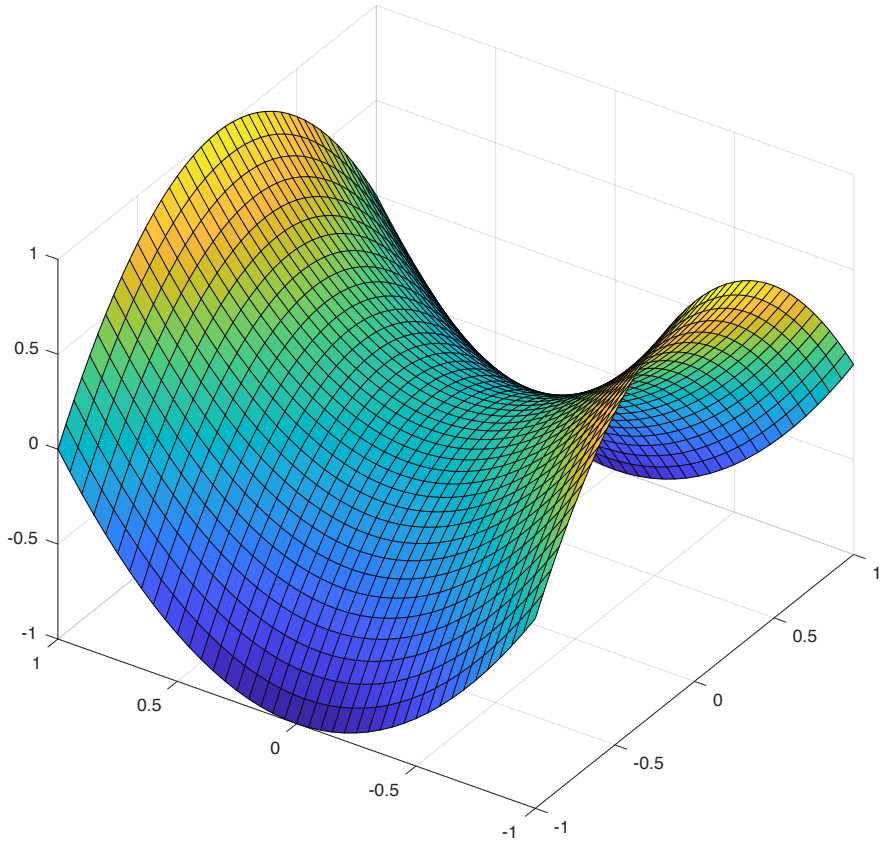


Figure 2.1.: Example for optimisation objective with indefinite Hessian: saddle point in $\mathbb{R}^2$

Theorem 2.2 (Sufficient optimality condition)**.** *Let $c(w) \in \mathcal{C}^2(\mathbb{R}^n)$, i.e. twice continuously differentiable and $\mathrm{H}_c(w)$ denote the Hessian of $c(w)$. Further, let $w^* \in K \subset \mathbb{R}^n$ open and*

- $\nabla_w c(w^*) = 0$
- $d^\mathsf{T} \mathrm{H}_c(w^*) d > 0 \; \forall d \in \mathbb{R}^n \backslash \{0\}$

Then w^* is a local minimiser of $c(w)$.

Proof. Since $\mathrm{H}_c(w^*)$ is positive definite and continuous, there exist $\varepsilon > 0$ so that for $d \in \mathbb{R}^n$ and any $z \in \mathcal{B}_\varepsilon(w^*)$ (the n-dimensional open ball around w^* with radius ε)

$$d^\mathsf{T} \mathrm{H}_c(z) d > 0$$

Choosing any d with $0 < \|d\|_2 < \varepsilon$ and applying Taylor's theorem yields

$$c(w^* + d) = c(w^*) + \nabla_w c(w^*)^\mathsf{T} d + \frac{1}{2} d^\mathsf{T} \mathrm{H}_c(w^* + td) d = c(w^*) + \frac{1}{2} d^\mathsf{T} \mathrm{H}_c(w^* + td) d > c(w^*)$$

for some $t \in [0, 1]$. □

2.2. Gradient descent

It can be shown that the direction of steepest descent w.r.t. the **Euclidean norm**, i.e. solution of

$$\min_{\|d\|_2 = 1} \nabla_w c(w)^\mathsf{T} d$$

is the negative, normalised gradient direction [46]. This property is utilised in the gradient descent method. Note that choosing a direction where the cost value decreases is usually

Data: number of iterations, initial w^0
for k *in* 0 *to number of iterations* **do**
  if $\nabla_w c(w^k) = 0$ **then**
    break
  end
  $s^k \leftarrow -\nabla_w c(w^k)$
  calculate σ_k so that $c(w^k + \sigma_k s^k) < c(w^k)$
  $w^{k+1} \leftarrow w^k + \sigma_k s^k$
end
return w^k

Algorithm 1: Gradient descent

possible for the above assumptions until a minimum is reached. The only limitation is convergence to a point where $\nabla_w c(w) = 0$ but which is not a minimum (saddle point).

Proposition 2.3. *For $c(w) \in \mathcal{C}(\mathbb{R}^n)$ either $\nabla_w c(w) = 0$ or there exists a $\sigma \in \mathbb{R}$ so that with $s = -\nabla_w c(w)$*

$$c(w + \sigma s) < c(w)$$

Proof. Suppose $\nabla_w c(w) \neq 0$. The claim follows from continuity of $\nabla_w c(w)$. □

In the general case, no convergence result for algorithm 1 can be given. Further assumptions are necessary to assess the convergence rate: assume $c(w) \in \mathcal{C}^2(\mathbb{R}^n)$ is quadratic and its Hessian is positive definite. If the best possible step length is chosen in each iteration, the error converges Q-linearly [23]:

$$c(w^{k+1}) - c(w^*) \leq \rho \cdot (c(w^k) - c(w^*))$$

where w^* denotes the local minimiser and $\rho \in [0, 1)$.

2.3. Newton method

The Newton method for the solution of non-linear systems of equations is the archetype for second-order optimisation. There are two ways to derive the connection. One is slightly shorter and is thus reserved for a brief motivation of context in a later chapter. In this section, the focus will be on the second derivation, which involves quadratic approximation of the objective function $c(w + s)$ around w: using Taylor series, the quadratic form reads

$$q(s) = c(w) + \nabla_w c(w)^\mathsf{T} s + \frac{1}{2} s^\mathsf{T} \mathrm{H}_c(w) s$$

If there exists a minimum of $q(s)$, the necessary optimality condition has to hold.

$$\nabla_s q(s) = \nabla_w c(w) + \mathrm{H}_c(w) s \stackrel{!}{=} 0$$

This leads to the so-called Newton equation

$$\boxed{\mathrm{H}_c(w) s = -\nabla_w c(w)}$$

which can be applied successively until convergence. This is called Newton method (algorithm 2). Note that this algorithm, unlike gradient descent, does not incorporate a step size calculation. Since the Newton method is based on quadratic approximation of the objective, curvature information is available. This makes step size adaption obsolete. More importantly, fast local convergence can be shown for a reasonably applicable case. Assume $c(w) \in \mathcal{C}^2(\mathbb{R}^n)$ and its Hessian is positive definite. Denote the local minimiser of $c(w)$ by w^*. Then there is a $\delta > 0$ so that for all $w^0 \in \mathcal{B}_\delta(w^*)$ algorithm 2 either terminates with $w^k = w^*$ or converges Q-superlinearly [46]:

$$\|w^{k+1} - w^*\| = o(\|w^k - w^*\|)$$

Data: number of iterations, initial w^0
for k *in* 0 *to number of iterations* **do**
  if $\nabla_w c(w^k) = 0$ **then**
    break
  end
  solve $\mathrm{H}_c(w^k)s^k = -\nabla_w c(w^k)$
  $w^{k+1} \leftarrow w^k + s^k$
end
return w^k

Algorithm 2: Newton method

The convergence result only applies if the starting point is sufficiently close to the minimum. Indeed, algorithm 2 is not globally convergent for all starting points w^0. To illustrate this, consider $c : \mathbb{R} \to \mathbb{R}$, $c(w) = \sqrt{(w)^2 + 1}$. Then

$$\nabla_w c(w) = \frac{w}{\sqrt{(w)^2 + 1}}, \ \mathrm{H}_c(w) = \frac{1}{((w)^2 + 1)^{\frac{3}{2}}} > 0$$

Substituting this in the Newton equation yields

$$s^k = -w^k((w^k)^2 + 1)$$

and thus the weight update becomes

$$w^{k+1} = -(w^k)^3$$

It is evident that any $|w^0| > 1$ leads to divergence $|w^k| \to \infty$. In order to make algorithm 2 globally convergent, it has to be modified in a way that it

- checks if the solution of the Newton equation is a descent direction and if not, employs an appropriate fall-back (e.g. steepest descent)
- includes step size adaption to avoid instability and account for the fall-back method.

3. Algorithmic differentiation

One of the main reasons deep neural networks can be as successful as they are, is the discovery of backpropagation. The term was originally introduced in [35], incorporated in a form of (stochastic) gradient descent. Since then, the technique has become the backbone of neural network training. The underlying idea is to arrange the model into a computational graph and recursively calculate the gradient using input-dependent derivatives equipped to each node. This is known as algorithmic differentiation (AD). The name "backpropagation" relates to the manner in which the gradients are calculated: Let $f(x, w)$ be a neural network characterised by a succession of k layers each implied in their activation function $\phi^i(\phi^{i-1}),\ i = 2, \dots, k$ and $\phi^1(x)$ as

$$f(x, w) = \left(\phi^k \circ \phi^{k-1} \circ \cdots \circ \phi^1\right)(x)$$

where each $\phi^i(\cdot)$ depends on a sub-vector of w in a way that every weight only appears in one layer. This assumption is motivated by common practice. Consider as an example the partial derivative of the i-th component of $f(x, w)$ w.r.t. the j-th weight w_j that appears in layer $\phi^\nu(\cdot)$. Applying the chain rule yields

$$\frac{\partial f_i}{\partial w_j} = \left(\frac{\partial \phi^k}{\partial \phi^{k-1}} \circ \phi^{k-1} \circ \cdots \circ \phi^1\right) \cdot \left(\frac{\partial \phi^{k-1}}{\partial \phi^{k-2}} \circ \phi^{k-2} \circ \cdots \circ \phi^1\right) \cdots \left(\frac{\mathrm{d}\phi^\nu}{\mathrm{d}w_j} \circ \phi^{\nu-1} \circ \cdots \circ \phi^1\right)$$

which can be interpreted as "passing backwards" through the graph. Neural networks are well suited for a graph-based implementation due to their modular composition of layers, activation and loss. For further information-theoretic details of the implementation of algorithmic differentiation, refer to [13]. Note that AD calculates exact derivatives, as opposed to approximated ones (e.g. by finite differences), while keeping the complexity close to a regular function evaluation ("forward pass"). The main drawback of this method regarding network design is the limitation to a finite library of building blocks with well-defined derivatives. There are several AD packages for Python, including Autograd, CasADi [3] and TensorFlow last of which is designed for neural network use.

4. Hessian product

For application of the second-order optimisation algorithm proposed in section 2.3, the Newton equation

$$\mathrm{H}_c(w)s = -\nabla_w c(w)$$

has to be solved in every iteration. Explicit evaluation of the Hessian, however, may be infeasible for state-of-the-art large-scale deep neural networks, as will be discussed later in this work. Therefore, the efficient Hessian-vector multiplication presented in the following chapter has become a key component of modern second-order training frameworks.

4.1. Gateaux derivative

Pearlmutter proposes in [32] a fast and exact way to evaluate the product of a vector $s \in \mathbb{R}^n$ with the Hessian $\mathrm{H}_c(w)$. The right-hand side of

$$\mathrm{H}_c(w)s = \left(\frac{\partial}{\partial r}\nabla_w c(w + rs)\right)\bigg|_{r=0} \tag{4.1}$$

is known as the Gateaux derivative of $\nabla_w c(w)$ at w in direction s. In the context of algorithmic differentiation (AD), it allows a product with the Hessian to be computed in two backpropagations instead of n.

4.2. Relation to directional derivative

Since this representation was created with different computational frameworks in mind than are available today, a modern form that is linked to the directional derivative will be derived in the following.

Proposition 4.1. *Let $c \in \mathcal{C}^2(\mathbb{R}^n)$, i.e. twice continuously differentiable. Then*

$$\left(\frac{\partial}{\partial r}\nabla_w c(w + rs)\right)\bigg|_{r=0} = \nabla_w \left(\nabla_w c(w) \cdot s\right)$$

Proof. Assume $c \in \mathcal{C}^2(\mathbb{R}^n)$. The Hessian matrix is then defined as

$$\mathrm{H}_c(w) = \begin{pmatrix} \frac{\partial^2}{\partial w_1 \partial w_1} c(w) & \frac{\partial^2}{\partial w_1 \partial w_2} c(w) & \dots & \frac{\partial^2}{\partial w_1 \partial w_n} c(w) \\ \frac{\partial^2}{\partial w_2 \partial w_1} c(w) & \frac{\partial^2}{\partial w_2 \partial w_2} c(w) & \dots & \frac{\partial^2}{\partial w_2 \partial w_n} c(w) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial w_n \partial w_1} c(w) & \frac{\partial^2}{\partial w_n \partial w_2} c(w) & \dots & \frac{\partial^2}{\partial w_n \partial w_n} c(w) \end{pmatrix}$$

Consider the left-hand side of equation (4.1). With the definition of the Hessian, it can be written as

$$\mathrm{H}_c(w)s = \begin{pmatrix} \sum_{i=1}^{n} s_i \frac{\partial^2}{\partial w_1 \partial w_i} c(w) \\ \sum_{i=1}^{n} s_i \frac{\partial^2}{\partial w_2 \partial w_i} c(w) \\ \vdots \\ \sum_{i=1}^{n} s_i \frac{\partial^2}{\partial w_n \partial w_i} c(w) \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial w_1} \sum_{i=1}^{n} s_i \frac{\partial}{\partial w_i} c(w) \\ \frac{\partial}{\partial w_2} \sum_{i=1}^{n} s_i \frac{\partial}{\partial w_i} c(w) \\ \vdots \\ \frac{\partial}{\partial w_n} \sum_{i=1}^{n} s_i \frac{\partial}{\partial w_i} c(w) \end{pmatrix}$$

since s is constant and $\frac{\partial}{\partial \cdot}$ is a linear operator. Applying the definition of the gradient yields

$$\begin{pmatrix} \frac{\partial}{\partial w_1} \sum_{i=1}^{n} s_i \frac{\partial}{\partial w_i} c(w) \\ \frac{\partial}{\partial w_2} \sum_{i=1}^{n} s_i \frac{\partial}{\partial w_i} c(w) \\ \vdots \\ \frac{\partial}{\partial w_n} \sum_{i=1}^{n} s_i \frac{\partial}{\partial w_i} c(w) \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial w_1} \left(\nabla_w c(w) \cdot s \right) \\ \frac{\partial}{\partial w_2} \left(\nabla_w c(w) \cdot s \right) \\ \vdots \\ \frac{\partial}{\partial w_n} \left(\nabla_w c(w) \cdot s \right) \end{pmatrix} = \nabla_w \left(\nabla_w c(w) \cdot s \right)$$

and thus it is equivalent to the Gateaux derivative from earlier. □

Note that $\nabla_w c(w) \cdot s = \nabla_w c(w)^\mathsf{T} s$ is the directional derivative of $c(w)$ at w in direction s. Consequently, the product of the Hessian with an arbitrary vector s can be seen as the gradient of the directional derivative w.r.t. s. These calculations are motivated by the fact that this new representation is easy to implement, e.g. in the popular AD framework TensorFlow.

Remark 4.2. If one would need the vector-matrix product $s^\mathsf{T}\mathrm{H}_c(w)$, by symmetry of the Hessian (Schwarz's theorem)

$$s^\mathsf{T}\mathrm{H}_c(w) = (\mathrm{H}_c(w)s)^\mathsf{T}$$

5. LSE solvers

The solution of the Newton equation is an important element of second-order optimisation. Consider the cost function $c(X, Y, w)$ with X, Y fixed. With its Hessian $\mathrm{H}_c(w) \in \mathbb{R}^{n \times n}$, the equation reads

$$\mathrm{H}_c(w)s = -\nabla_w c(X, Y, w)$$

For fixed w, it has to be solved for s and is therefore a linear system of equations (LSE). Numerical mathematics provide a plethora of solution methods for LSEs. Note that for $c(X, Y, w) \in \mathcal{C}^2(\mathbb{R}^n)$ the Hessian is symmetric by definition and Schwarz's theorem.

Direct methods

Originally, LSE solvers were developed with small, well-defined systems in mind. It is for that reason that many of these algorithms lead to an exact solution in a finite number of steps. While achieving this, they operate on an explicit and dense system matrix. Popular examples are Gaussian elimination, QR decomposition and Cholesky factorisation. To be successful in finding a solution, they usually require a certain regularity of the system matrix: For Gaussian elimination and QR decomposition it has to be invertible and for Cholesky factorisation even positive definite. There are two main reasons direct methods are not suitable for large-scale deep learning.

- **Regularity.** In practice, the Hessian cannot be guaranteed to be non-singular. This can be due to numerical noise or certain behaviour of the loss landscape, as will be explained in section 7.1. Only for carefully chosen cost functions $c(X, Y, w)$ some sort of regularity can be established, which sacrifices flexibility and makes them unsuitable for modern machine learning tasks.
- **Explicit Hessian.** For state-of-the-art deep neural networks, computation and storage of the explicit Hessian may not be feasible. It is due to the extremely high dimension n of the weight space, because both computation work and storage scale with $\mathcal{O}(n^2)$. This will be elaborated on later in this work.

Iterative methods

Addressing the aforementioned concern of regularity, predominantly iterative methods are employed in neural network training. Instead of solving the LSE in one go, they usually increase the quality of the solution in each iteration. Stopping after a number of iterations creates an approximation to the correct solution. Therefore, when the Newton

equation is solved in this way, it is called inexact Newton method. Iterative methods are relevant in practice, because they allow even numerically noisy problems that are ill-posed as a consequence to be approximately solved. This is achieved by creating a stopping criterion based on the average per-iteration improvement or simply by stopping after a set number of iterations. Additionally, computational resources can be used efficiently, since the numerical precision is limited anyway and iterative methods usually incorporate a tolerance for a sufficient solution. With the first problem of direct methods taken care of, only the explicit Hessian evaluation has to be addressed to find a method fit for deep learning applications.

Conjugate gradient method

In chapter 4, a trick was proposed to avoid the explicit evaluation of the Hessian. It turns out that the product of a vector with the Hessian can be exactly evaluated in a numerical efficient way. This leaves one to find a LSE solver that incorporates the Hessian exclusively in a product with an arbitrary vector. Fortunately, a well-established and straight-forward algorithm with this trait is given in the conjugate gradient (CG) method [16]. Given a symmetric and positive definite Hessian $\mathrm{H}_c \in \mathbb{R}^{n \times n}$ it even finds the exact solution to the Newton equation in at most n steps. Therefore, it becomes a direct method at n iterations. What is more, CG also functions very well as an iterative method: it can be shown that

Data: system matrix A, right-hand side b, initial guess ξ^0, tolerance
Result: approximate solution of $A\xi = b$
$r^0 \leftarrow b - A\xi^0$
$p^0 \leftarrow r^0$
$k \leftarrow 0$
while $\|r^k\| > \mathit{tolerance}$ **do**
  $\alpha^k \leftarrow \frac{(r^k)^\mathsf{T} r^k}{(p^k)^\mathsf{T} A p^k}$
  $\xi^{k+1} \leftarrow \xi^k + \alpha^k p^k$
  $r^{k+1} \leftarrow r^k - \alpha^k A p^k$
  $\beta^k \leftarrow \frac{(r^{k+1})^\mathsf{T} r^{k+1}}{(r^k)^\mathsf{T} r^k}$
  $p^{k+1} \leftarrow r^{k+1} + \beta^k p^k$
  $k \leftarrow k + 1$
end
return ξ^{k+1}

Algorithm 3: Conjugate gradient (CG) method

it converges R-linearly, dependant on the condition number of the system matrix [25]. Consider algorithm 3 and assume A is symmetric positive definite. With the ratio of the

maximum and minimum eigenvalue of A

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

the rate of convergence to the exact solution $\xi^* = A^{-1}b$ is

$$\|\xi^k - \xi^*\|_A \leq \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|\xi^0 - \xi^*\|_A$$

where $\|\cdot\|_A = \sqrt{(\cdot)^\mathsf{T} A(\cdot)}$, the energy norm of A. This implies that the algorithm produces monotonically improving iterates. Due to the dependence of the convergence rate on the condition number, preconditioning may be crucial in practice.

6. A different view on neural network training

Consider a neural network that is characterised by its model output $f : S \times \mathbb{R}^n \to \mathbb{R}^l$, i.e. takes a sample $x \in S$ and for a given weight configuration $w \in \mathbb{R}^n$ outputs a prediction assigning a value from $[0, 1]$ to a number of l classes. Given a label vector y in one-hot encoding (one for the correct class, zero otherwise) the mean squared error $c(x, y, w)$ of the model output is

$$c(x, y, w) = \frac{1}{l}(f(x, w) - y)^\mathsf{T}(f(x, w) - y) = \frac{1}{l}(f(x, w)^\mathsf{T} f(x, w) - 2f(x, w)^\mathsf{T} y + y^\mathsf{T} y)$$

In order to train the network on the given sample, the cost function is minimised. For the necessary optimality condition, the gradient has to be equal to zero. To this end, reformulate the equation

$$\frac{1}{l}(f(x, w)^\mathsf{T} f(x, w) - 2f(x, w)^\mathsf{T} y + y^\mathsf{T} y) = \frac{1}{l}(\sum_{i=1}^{l} f_i^2 - 2\sum_{i=1}^{l} f_i y_i + y^\mathsf{T} y)$$

It becomes evident that the gradient calculates to

$$\nabla_w c(x, y, w) = \frac{1}{l}(2\sum_{i=1}^{l} f_i \nabla_w f_i - 2\sum_{i=1}^{l} y_i \nabla_w f_i) = \frac{2}{l}(\mathrm{J}_f(w)^\mathsf{T}(f(x, w) - y))$$

and the necessary optimality condition is

$$\nabla_w c(x, y, w) = 0 \Leftrightarrow \mathrm{J}_f(w)^\mathsf{T}(f(x, w) - y) = 0 \tag{6.1}$$

In the following, fix $x \in S$. If now the model $f(x, w)$ is linear or linearly approximated around w, i.e.

$$f(x, w + s) = f(x, w) + \mathrm{J}_f(w)s$$

the optimality condition (6.1) becomes

$$\mathrm{J}_f(w)^\mathsf{T}(f(x, w) + \mathrm{J}_f(w)s - y) = 0 \Leftrightarrow \mathrm{J}_f(w)^\mathsf{T}\mathrm{J}_f(w)s = \mathrm{J}_f(w)^\mathsf{T}(y - f(x, w)) \tag{6.2}$$

since

$$f(x, w) = b(x) + A(x)w \Rightarrow \mathrm{J}_f(w) = A(x)$$

The linear system of equations (6.2) can be solved by an appropriate solver, for example the conjugate gradient method (CG). The applicability of the resulting solution will depend on how close to linearly the neural network behaves locally. A drawback of the presented approach is that deriving second order optimisation algorithms in logical succession to the previous calculations is tedious. While it can be done theoretically, modern algorithmic-differentiation frameworks handle the occurring calculations efficiently and in a black-box manner. Thus, abstraction to the level of only considering a non-linear cost function $c(x, y, w)$ is practical.

Part III.

Methods and implementation

7. Newton equation

7.1. Non-convex optimisation

Relation of strict convexity and positive definiteness

In the training of deep neural networks for image classification, the objective is to minimise a cost (or loss) function $c : S^d \times L^d \times \mathbb{R}^n \to \mathbb{R}$, where S is the sample space, L is the label space, n denotes the number of neuron connection weights and d denotes the number of available data samples:

$$\min_{w \in \mathbb{R}^n} c(X, Y, w) \tag{7.1}$$

with $X \in S^d$ and $Y \in L^d$ fixed. This is an unconstrained optimisation problem. In the following, the notion of convexity will be discussed. The first two arguments of c will be dropped for simplicity.

Definition 7.1. Let $K \subset \mathbb{R}^n$ be a convex set. A function $c : \mathbb{R}^n \to \mathbb{R}$ is called strictly convex on K if for all $a, b \in K$, $a \neq b$ and $\lambda \in (0, 1)$

$$c((1-\lambda)a + \lambda b) < (1-\lambda)c(a) + \lambda c(b)$$

In the context of neural network training, the cost function c can be chosen continuously differentiable on the weight space. This requires the appropriate building blocks in the form of activation and loss functions, as seen before. For continuously differentiable functions, strict convexity can be further concretised [46].

Theorem 7.2. *Let $c \in \mathcal{C}^2(\mathbb{R}^n)$, i.e. twice continuously differentiable on $\mathbb{R}^n$ and $K \subset \mathbb{R}^n$ be convex. The function $c(w)$ is strictly convex on K if its Hessian $\mathrm{H}_c(w)$ is positive definite for all $w \in K$, i.e.*

$$\forall d \in \mathbb{R}^n \backslash \{0\} : d^\mathsf{T} \mathrm{H}_c(w) d > 0$$

Proof. With arbitrary $a, b \in K$, Taylor's theorem yields

$$c(b) = c(a) + \nabla c(a)^\mathsf{T}(b-a) + (b-a)^\mathsf{T} \mathrm{H}_c(\xi_\lambda)(b-a)$$

for $\xi_\lambda = a + \lambda(b-a)$ and some fixed $\lambda \in [0, 1]$. This is to say ξ_λ lies on the line segment between a and b. By the positive definiteness of $\mathrm{H}_c(w)$,

$$c(b) - c(a) > \nabla c(a)^\mathsf{T}(b-a) \tag{7.2}$$

In order to arrive at the definition of convexity, consider for general $\lambda \in [0, 1]$

$$(1-\lambda)c(a) + \lambda c(b) - c(\xi_\lambda) = (1-\lambda)(c(a) - c(\xi_\lambda)) + \lambda(c(b) - c(\xi_\lambda))$$

From (7.2) follows

$$(1-\lambda)(c(a) - c(\xi_\lambda)) + \lambda(c(b) - c(\xi_\lambda)) > (1-\lambda)\nabla c(\xi_\lambda)^\mathsf{T}(a - \xi_\lambda) + \lambda \nabla c(\xi_\lambda)(b - \xi_\lambda)$$

and the right-hand side simplifies to

$$\nabla c(\xi_\lambda)^\mathsf{T}((1-\lambda)a + \lambda b - \xi_\lambda) = 0$$

Consequently,

$$(1-\lambda)c(a) + \lambda c(b) - c(\xi_\lambda) > 0$$

which is the definition of convexity. □

Remark 7.3. It can be shown that the converse of theorem 7.2 does not hold. In other words: Even if $c(w)$ is strictly convex, it is not guaranteed that its Hessian is positive definite. Consider $f(x) = x^4$ as a counterexample on $\mathbb{R}$: This function is strictly convex on $[-1, 1]$ but $f''(0) = 0$.

While the cost function c can be chosen $\mathcal{C}^2$, it is generally not strictly convex on the whole weight space. This leads to the Hessian not being positive definite on some subsets.

Corollary 7.4. *Let $c \in \mathcal{C}^2(\mathbb{R}^n)$ be not strictly convex on K. Then the Hessian H_c is not positive definite everywhere on K.*

Proof. Assume for a proof by contradiction that $\mathrm{H}_c(w)$ is positive definite for all $w \in K$. Then c is strictly convex on K by theorem 7.2. □

Consequently, if c is not strictly convex on $\mathbb{R}^n$, its Hessian is not positive definite everywhere.

Implications for optimisation: pre-training

Consider now the minimisation problem (7.1). In optimisation, a necessary (but not sufficient) condition for a minimum is

$$\nabla_w c(w + s) = 0 \tag{7.3}$$

On a set where H_c is positive definite, this zero of the gradient can be found by solvers for non-linear systems of equations, e.g. the Newton method. To this end, Taylor-approximate equation (7.3) linearly around w:

$$\nabla_w c(w) + \mathrm{H}_c(w)s + \rho(s) = 0$$

where $\|\rho(s)\| = o(\|s\|)$ and thus it can be neglected for small s. This leads to the Newton equation

$$\mathrm{H}_c(w)s = -\nabla_w c(w) \tag{7.4}$$

and the idea is to solve it for s and update w iteratively until the loss attains a satisfactory tolerance.

Although (7.4) is a linear system of equations, solving it by explicit methods is not feasible in practice. It would require computation and storage of the full Hessian with n^2 entries, taking n backpropagations in an algorithmic-differentiation framework. It is much more efficient to leverage the tools discussed in chapter 4 in the context of an appropriate LSE solver, e.g. the conjugate gradient method. Recall that many solvers, including CG, require the system of equations to have a positive definite matrix [46]. However, from corollary 7.4 follows that this is not the case everywhere. Even in the convex neighbourhood $w \in K$ of a minimum, where $c(w)$ is strictly convex, its Hessian is not necessarily positive definite, as shown in remark 7.3.

Nonetheless, in order to make sure that the Hessian **can** be positive definite everywhere, by corollary 7.4, a set K has to be found, where c is strictly convex. This is important because the Newton equation represents a quadratic approximation to the cost function (see section 2.3) for which minimisation is only meaningful if the objective is convex. Additionally, CG can generally not be used to solve the Newton equation (7.4) outside of such a neighbourhood. If the Hessian is positive definite almost everywhere in K, there is a high chance that CG performs well. In neural network training, naïve methods (e.g. gradient descent) can be employed during a pre-training period to iterate towards such a neighbourhood.

7.2. Tikhonov regularisation

Due to the large scale of state-of-the-art deep neural networks, the linear system of equations resulting from application of the Newton method to the optimisation problem (7.1) may often be ill-conditioned, i.e. with the maximum and minimum eigenvalue of H_c

$$\frac{\sigma_{\max}(\mathrm{H}_c)}{\sigma_{\min}(\mathrm{H}_c)} \gg 1$$

The computational solution of such a system is prone to numerical instability, since the matrix is close to singular. This can also leads to slow convergence of the CG method, as was mentioned in chapter 5.

Quasi-Newton method

To address this issue, some sort of regularisation can be applied. A straight-forward and easy-to-implement method is the so-called Tikhonov regularisation [44][43][45]. It was

independently developed for many different mathematical fields, e.g. statistics, where it is known as ridge regression. Given a linear system of equations

$$\mathrm{H}_c s = -\nabla_w c$$

the key idea is to add an appropriate $\Gamma \in \mathbb{R}^{n \times n}$ to the system matrix H_c in order to enforce certain desired properties. For example, this matrix can be chosen $\Gamma = \alpha \mathrm{I}$, with a small $\alpha \in \mathbb{R}$, which is known as L_2-regularisation [28]:

$$(\mathrm{H}_c + \alpha \mathrm{I}) s = -\nabla_w c \tag{7.5}$$

To incorporate the previously discussed efficient Hessian product, by linearity

$$\mathrm{H}_c s + \alpha s = -\nabla_w c$$

Note that by changing the system matrix used in the Newton method, a so-called Newton-like method is introduced. Together with inexact Newton methods, this class is usually summarised under the term quasi-Newton methods. These have been extensively studied [9][29] with the most prominent algorithm for machine learning being the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method [20][5].

Drawbacks

The discussed regularisation improves the conditioning of the problem and thereby facilitates the numerical solution. However, it comes at the cost of distorting the loss landscape: Adding noise to the exact Hessian can be interpreted as altering the curvature of the original cost function in (7.1). This ultimately changes the minimisation objective and leads to imprecise solutions. Therefore, a trade-off has to be found between numerical efficiency and precision of the objective. To this end, the factor α in (7.5) is typically chosen to be slightly above machine precision.

8. Stochastic approximation

8.1. Stochastic gradient descent

Consider the optimisation problem (7.1). The previously discussed gradient descent method can be applied as a naïve algorithm for its solution. To this end, the weight vector receives an update in each iteration that is computed with the gradient of the cost function and a step size σ_k as

$$w^{k+1} = w^k - \sigma_k \nabla_w c(X, Y, w^k) \tag{8.1}$$

where $X \in S^d$, i.e. from the d-dimensional sample space and $Y \in L^d$, from the d-dimensional label space. In other words, d sample-label pairs are used in the computation of a gradient.

Limitations and stochastic approximation

Using a suitable framework for algorithmic differentiation, this requires $\mathcal{O}(d)$ work, i.e. scales linearly with the number of samples. In practice, training data sets are typically large. In the domain of image classification, prominent examples are the MNIST handwritten digits data set with $60,000$ samples in its training set and ImageNet, a collection of images depicting different objects, people and animals, with around 1.2 million samples in its 2012 version respectively. Since computation of the gradient becomes impractical for such large d, the optimisation is usually restricted to a subset X_i of $b \ll d$ samples.

$$X_i \in S^b,\ Y_i \in L^b$$

This gives rise to a stochastic approximation of the original optimisation problem (7.1) which becomes exact as $b \to d$:

$$\min_{w \in \mathbb{R}^n} c_i(X_i, Y_i, w)$$

with $c_i : S^b \times L^b \times \mathbb{R}^n \to \mathbb{R}$. The so-called batch size b can be chosen arbitrarily small, up until $b = 1$. In the case where computing with all available samples at once is not possible due to limited computational resources, sequential optimisation on a selection of subsets $X_i,\ i = 1, \ldots, m$ may be used as a compromise. Typically, the union of all subsets comprises the whole available data.

$$\bigcup_{i=1}^{m} X_i = X$$

One cycle in which all samples are used in sub-optimisations is often referred to as an epoch. Stochastic gradient descent (SGD) in the narrow sense is the method of randomly sampling a batch of size $b = 1$ in each of multiple sequential sub-optimisations and repeating over several epochs. An example of an algorithm working with larger mini-batches of input data is the prominent Adam method [18].

SGD vs. full-batch approach

For large data sets, SGD in the narrow sense might have desirable properties and thus be favoured over a full-batch approach. To illustrate this, fix $X \in S^d$ and $Y \in L^d$ and assume $c(X, Y, w) \in \mathcal{C}^2(\mathbb{R}^n)$ has a positive definite Hessian on a set K and attains a minimum on K which is denoted by c^*. To find said minimum, first employ the full-batch gradient descent method (8.1). With a carefully selected step size and applied to a quadratic objective, the algorithm will exhibit R-linear convergence [7], i.e. there exists $\rho \in (0, 1)$ so that after k iterations

$$c(X, Y, w^k) - c^* \leq \mathcal{O}(\rho^k)$$

In order to arrive at a desired tolerance $\epsilon > 0$, it follows that $\mathcal{O}(\log(1/\epsilon))$ iterations are needed:

$$\epsilon \leq \mathcal{O}(\rho^k) \Leftrightarrow \log(\epsilon) \leq \mathcal{O}(k \log(\rho))$$

and since $\rho \in (0, 1)$, $\log(\rho)$ is negative and non-zero

$$\mathcal{O}\left(\frac{\log(\epsilon)}{-1|\log(\rho)|}\right) \geq k \Rightarrow \mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\right) \geq k$$

Recall that the computation of each full-batch gradient requires $\mathcal{O}(d)$ work. Multiplied by the number of iterations this yields $\mathcal{O}(d \log(1/\epsilon))$. SGD can be analysed in a similar manner. Let $X_i \in S,\ i = 1, \ldots, d$ be uniformly sampled from the whole data set X. Under certain regularity assumptions about $c(w)$ and with a certain step size, the algorithm converges sublinearly [6]: After k iterations

$$\mathbb{E}\left[c(X, Y, w^k) - c^*\right] = \mathcal{O}\left(\frac{1}{k}\right)$$

i.e. the expected value of the training error decreases proportionally to $1/k$. Consequently, the tolerance ϵ is attained after $\mathcal{O}(1/\epsilon)$ iterations. Note that this convergence rate is independent of the data set size d and directly translates into the required work, because no large gradients have to be calculated. As d is usually large, the convergence rate may favour SGD over the full-batch approach in practice.

Implications for classification and regression

Considering the context of image classification, (7.1) can be interpreted as a mathematical translation of the ambiguous formulation ”assign the correct labels to arbitrary images”.

If the initial task is to classify samples that are not contained in the training data, one might expect that not all d samples have to be used anyway. Moreover, evidence suggests that predictions might even be more robust when the network is sequentially trained on smaller subsets of data. Yao et al. state in [50] that large-batch training leads deep neural networks to be weak to adversarial attacks, i.e. perturbation of inputs with small but carefully-chosen noise to provoke classification errors. A well-received explanation is that the network cannot rely on the training set as heavily in that case.

This, in turn, means that for regression problems, where the task is to assign the right value to certain samples in a preferably precise way, such a stochastic approach exhibits low chances of success. Instead of flexibility, regression problems call for the precise mapping of the training data. The stochastic approach adds unwanted variance to the objective and thus seems unfit. With respect to the aforementioned convergence rates, SGD also converges too slowly to guarantee a precise solution in a feasible amount of time.

8.2. Stochastic Newton method

Motivation for second-order methods

Yao et al. argue in [50] that large-batch training leads the optimisation algorithm to converge to areas with high curvature in the loss landscape. Such areas are weak to adversarial attacks, since slight perturbations highly influence the cost value and therefore exhibit weak generalisation properties. In contrast, a stochastic approach introduces approximation noise which is believed to enable the iteration to skip past those minima. Depending on the application however, a stochastic approximation with high variance might be unwanted. Examples are regression problems and classification problems with (infinitely) large training sets, which are still supposed to be efficiently mapped. In those cases, a larger mini-batch size is preferred [4].

When leaving the regime of high-variance approximation, second-order optimisation methods become attractive. Since the gradient descent method is inherently sequential, it cannot fully take advantage of modern computational architectures. For this reason, second-order optimisation methods have been rising to prominence. These exhibit fast convergence to a local optimum due to the incorporation of curvature information. When computing power is no longer the bottleneck, their extensive computational cost seems justified.

To summarise, large mini-batch sizes favour second-order methods. Conversely, large batches are needed in order to construct a quadratic, stochastic approximation of the objective function in a meaningful way, avoiding variance. Advances have been made to develop stochastic variants of the BFGS method which are scalable to high-dimensional regimes [38].

Stochastic Newton method

The Newton method is prototypical for second-order optimisation. Most other methods either approximate the curvature matrix (Newton-like) or the solution of the Newton equation (inexact) and thereby create the family of quasi-Newton methods. Consider the Newton equation for the solution of (7.1),

$$\mathrm{H}_c(w^k)s = -\nabla_w c(X, Y, w^k)$$

where X and Y again denote the whole data. After it is solved for s, an update is applied to the weights according to

$$w^{k+1} = w^k + s$$

Note that no step-size adaption is applied here. In a theoretical framework where certain properties of the cost function $c(X, Y, w)$ are assumed, the Newton method autonomously produces an appropriate step size. It is implied in s. For this reason, it is commonly believed that the inclusion of curvature information leads to less hyper parameters in those methods. In an applied context, however, step size control may be needed to balance out numerical instabilities and ill-conditioning.

Analogously to the first-order case, restricting the training data so a subset of the available samples introduces a stochastic approximation

$$\min_{w\in\mathbb{R}^n} c_i(X_i, Y_i, w)$$

to the minimisation problem.

Convergence analysis

Estimating the required computational work of this iteration is less trivial. Consider the corresponding stochastic extension to the Newton equation

$$\mathrm{H}_{c_i}(w^k)s = -\nabla_w c_i(X_i, Y_i, w^k)$$

In order to make meaningful statements, assume it can be solved in $\mathcal{O}(\psi(b))$ work, where b is again the mini-batch size. Function $\psi(\cdot)$ is determined by the method of numerical solution. This assumption is motivated by the algorithm discussed in the next chapter. Fix $X \in S^d$ and $Y \in L^d$ and assume $c(X, Y, w) \in \mathcal{C}^2(\mathbb{R}^n)$ has a positive definite Hessian on a set K. Denote its minimizer on K by w^*. The convergence analysis of the full-batch problem follows the one in section 2.3:

$$\|w^{k+1} - w^*\| = o(\|w^k - w^*\|)$$

i.e. the iteration converges Q-superlinearly. In order to achieve a desired tolerance $\epsilon > 0$ the iterate w^k only has to be sufficiently close to the minimum while each iteration has the

complexity $\mathcal{O}(\psi(d))$. Regarding the stochastic approach, where in each iteration, a mini-batch of size $b = 1$ is uniformly sampled from the available data, Agarwal et al. showed that the expected convergence rate cannot be better than sublinear [1].

$$\mathbb{E}\left[c(X, Y, w^k) - c^*\right] = \mathcal{O}\left(\frac{1}{k}\right)$$

Here, the total required work to achieve expected ϵ-optimality multiplies to $\mathcal{O}(\psi(1)(1/\epsilon))$. Consequently, a stochastic Newton method with batch size $b = 1$ would have little advantage over SGD, if any. In order to leverage the benefits of second-order optimisation, larger mini-batches should be aimed for, as is once again illustrated. The transition from stochastic to full-batch approach is an interesting subject of research. Advances have been made to impose R-linear convergence on a stochastic second-order method using online batch size adaption [5]. To this end, the batch size is progressively increased during the optimisation.

9. Algorithm

The following chapter will provide an overview over the implementation details of large-scale second-order machine learning. For simplicity, suppose a regression problem is to be solved. In order to achieve a sufficiently accurate solution, a full-batch approach is taken:

$$\min_{w \in \mathbb{R}^n} c(X, Y, w)$$

with $X \in S^d$ and $Y \in L^d$ fixed. Regarding the implementation details, this is no different from solving a stochastic sub-problem in the context of a classification task:

$$\min_{w \in \mathbb{R}^n} c_i(X_i, Y_i, w)$$

with $X_i \in S^b$ and $Y_i \in L^b$. The latter can be extended to a state-of-the-art training algorithm by cumulating the sub-problems in an appropriate outer loop. For ease of readability, the first two arguments as well as the index of $c_i(X_i, Y_i, w)$ are dropped in the following. This is motivated by the fact that modern deep learning frameworks like TensorFlow can handle inputs in a vectorised manner with no need to redefine the cost function.

It should be stated that the following optimisation routine is in no way meant to be competitive with complete state-of-the-art stochastic training methods like SGD. The goal is merely to provide a template for second-order optimisation including all necessary components. For this reason, where there is exemplary code, it is provided in Python using TensorFlow (fast prototyping). Given the template, novel ideas and approaches can easily be implemented. To showcase the potential of second-order optimisation in large-scale machine learning, the performance of naïve implementations of first- and second-order methods will be compared in the next part.

9.1. Hessian product

The main bottleneck of large-scale second-order optimisation is the solution of the Newton equation

$$\mathrm{H}_c s = -\nabla_w c(w) \tag{9.1}$$

with $w \in \mathbb{R}^n$. All the variants are based on this equation. Replacing H_c with a different matrix, possibly imposing positive definiteness or improving conditioning, leads to a Newton-like method. Approximately solving (9.1) via an iterative method yields an inexact Newton iteration.

Comparison to full Hessian

Explicitly computing the Hessian is infeasible in many modern applications with respect to computational work and memory. Fortunately, the efficient Hessian product described in chapter 4 enables computation of the left-hand side of (9.1) in two backpropagation steps. Multiplying with the input batch size, here denoted by b, leads to $\mathcal{O}(2bn)$ work. However, the solution of the classic Newton equation requires an LSE solver, e.g. the conjugate gradient method. Since each CG step is powered by one left-hand-side evaluation, the total work sums up to $\mathcal{O}(2mbn)$, where m is the number of CG iterations needed to attain a satisfactory tolerance. It is well known that for a well-posed system, CG converges to the exact solution in at most n iterations and thus

$$m \leq n$$

Given that a full computation of the Hessian via backpropagation requires $\mathcal{O}(n^2 b)$ work - one backward pass times n for the second partial derivatives - it is evident that the approach outclasses explicit computation in almost every case. It is better, precisely if

$$2m < n$$

where m can be controlled via a tolerance argument passed to the CG algorithm.

Implementation details

The efficient Hessian product can easily be implemented using modern deep learning frameworks. Here, an exemplary code snippet in Python 3 with TensorFlow 2.0 is given.

```
1  import tensorflow as tf
2
3  # Hessian product
4  def Hs(samples, labels, weights, s):
5      with tf.GradientTape() as g:
6          g.watch(weights)
7          with tf.GradientTape() as gg:
8              gg.watch(weights)
9              c = cost(samples, labels, weights)
10         grad = gg.gradient(c, weights)
11         grad_s = tf.matmul(grad, s, transpose_a=True)
12     return g.gradient(grad_s, weights)
```

Line 11 contains the directional derivative of $c(w)$ w.r.t s

$$\nabla_w c(w)^\mathsf{T} s$$

For the sake of completeness, it will also be explained how to compute the full Hessian in $\mathcal{O}(n^2 b)$ work using TensorFlow. To this end, recall that taking the Jacobian of a gradient equals the Hessian.

$$\mathrm{J}_{\nabla_w c} = \mathrm{H}_c$$

Consequently, the full Hessian of the cost function can be computed by applying nested `tf.GradientTape()` calls.

```
import tensorflow as tf

# full Hessian
def H(samples, labels, weights):
    with tf.GradientTape(persistent=True) as g:
        g.watch(weights)
        with tf.GradientTape() as gg:
            gg.watch(weights)
            c = cost(samples, labels, weights)
        grad = gg.gradient(c, weights)
    return g.jacobian(grad, weights, experimental_use_pfor=False)
```

However, previous TensorFlow versions included a significantly more efficient function for direct computation: `tf.Hessians()`.

9.2. Optimisation

Algorithm

The Newton method for optimisation is not globally convergent for every starting point w^0. In particular, a strictly convex objective does not imply convergence of the method, as the counterexample in section 2.3 shows. The solution of the Newton equation may yield an infeasible search direction for which, assuming $c(w)$ is continuous, no minimisation of the loss can be guaranteed. In this case, an optimisation algorithm should fall back to some safe method. Furthermore, pre-training is required to set the LSE solver up for a high probability of success, as was elaborated on in section 7.1.

The optimisation routine is sketched in the following. The Newton equation is usually solved implicitly with CG using the Hessian product discussed in the previous section. Although, different approaches are possible. Since the Newton equation is prone to numerical ill-conditioning, even explicit solutions favour iterative solvers, e.g. least-squares approximation. Afterwards, the search direction s^k is checked against a threshold $T < 0$. If

$$\nabla_w c(w^k)^\mathsf{T} s^k > T$$

i.e. the directional derivative w.r.t s^k is non-negative, feasibility cannot be guaranteed and the algorithm falls back to a gradient descent iteration. This causes the need for a step size adaption. In the case when s^k is a feasible descent direction, the step size could be chosen $\sigma_k = 1$ because the Newton equation implies a valid step due to curvature information. In practice however, this might not be the case due to an ill-posed (locally not strictly convex) objective or numerical instabilities. It is therefore safer to apply the step size adaption even for feasible s^k.

Data: carefully initialised weights w^0
Result: local minimum of $c(w)$
for *number of pre-training iterations* **do**
  apply gradient descent
end
for k *in* 0 *to number of training iterations* **do**
  compute s^k by solving $\mathrm{H}_c(w^k)s^k = -\nabla_w c(w^k)$
  if $\nabla_w c(w^k)^\mathsf{T} s^k > T$ **then**
    $s^k \leftarrow -\nabla_w c(w^k)$
  end
  determine step size σ_k
  $w^{k+1} \leftarrow w^k + \sigma_k s^k$
end

Algorithm 4: Globalised Newton method for optimisation

Step size adaption

In the following, two popular step size adaptions will be explained.

- **Armijo step size adaption.** Line-search methods offer an intuitive approach to finding suitable step lengths σ_k. Their key idea is to impose certain constraints on a hypothetical step and iteratively lower σ_k until the constraints are met. Ideally, this is done while avoiding additional gradient evaluations. The so-called Armijo step size adaption operates as follows [46]: Given $\beta \in (0, 1)$ and $\gamma \in (0, 1)$, determine

  $$\sigma_k \in \{1, \beta, \beta^2, \dots\}$$

  so that

  $$c(w^k + \sigma_k s^k) - c(w^k) \leq \sigma_k \gamma \nabla_w c(w^k)^\mathsf{T} s^k$$

  A geometric interpretation of the method is shown in figure 9.1.

- **Learning rate.** For some applications the additional loss evaluations introduced by a line search algorithm might be problematic. This is typically the case for large-scale networks or certain code frameworks where the cost is not explicitly implemented as a function of the weights. In those cases, one can resort to using a learning rate. The step size is set to a predefined value that is tested to yield acceptable results. Learning rates can be adapted during the optimisation exclusively using first-order information [52] or incorporating second-order information [37][48][8].

Note that by adding a step size adaption, hyperparameters are introduced for the second-order method.

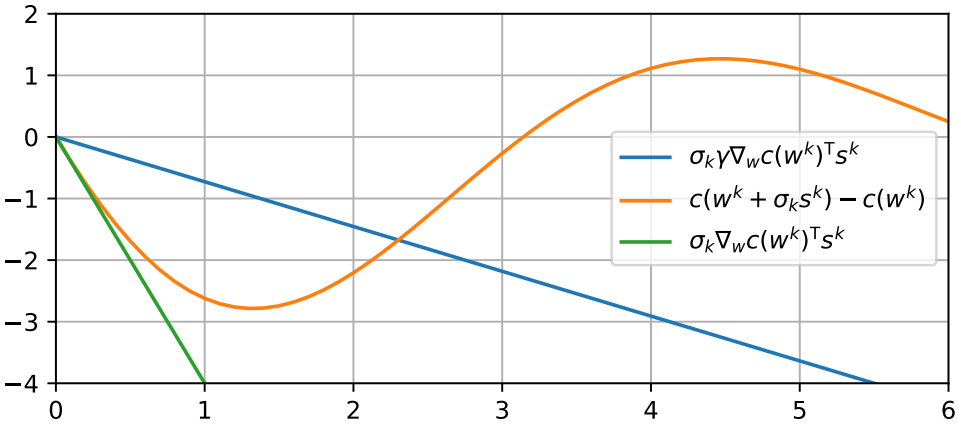


Figure 9.1.: Geometric interpretation of Armijo step size adaption on $\mathbb{R}$. Letting $\gamma \to 1$ leads to greedier step selection but may cause stagnation

Curse of dimensionality

In large-scale neural networks, the dimension n, i.e. number of weights is typically very high. Slight disturbances or numerical noise across all dimensions have great impact on summation operations that transform from $\mathbb{R}^n$ to $\mathbb{R}$. To illustrate this, suppose $v \in \mathbb{R}^n$ is afflicted with a numerical error ϵ. Model this by adding a vector x of independent, uniformly distributed random variables

$$x_i \sim \mathcal{U}(-\epsilon, \epsilon), \; i = 1, \dots, n$$

Now, consider the summation over all elements. Taking the variance yields

$$\operatorname{Var}\left(\sum_{i=1}^{n}(v_i + x_i)\right) = \sum_{i=1}^{n}\operatorname{Var}(v_i + x_i) = \sum_{i=1}^{n}\operatorname{Var}(x_i)$$

and by the formula for the variance of uniformly distributed random variables

$$\sum_{i=1}^{n}\operatorname{Var}(x_i) = \sum_{i=1}^{n}\frac{1}{12}(\epsilon + \epsilon)^2 = n\frac{\epsilon^2}{3}$$

It is evident that for very large n, this operation exhibits somewhat significant numerical variance. To combat this effect, the machine precision has to be set appropriately. However, this is not always feasible in terms of memory. Therefore, the Euclidean norm

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

loses part of its significance. This can get problematic because iterative solvers usually rely on an error norm as stopping criteria. To deal with this issue, one can factor the dimension n into these norms, for example by multiplying the tolerance by $\sqrt{n}$. Another approach could be to account for the underlying global geometry of the input data and formulate a novel distance measure, compare [42].

Complexity

Classic gradient descent requires one gradient evaluation per iteration. With the aforementioned assumptions this amounts to $\mathcal{O}(bn)$ work. Since gradient descent acts as the fall-back method, the second-order techniques expand on this complexity. The computational bottleneck of algorithm 4 is the evaluation of the Hessian. In the proposed form this requires a mathematical framework to perform two backpropagations, thus scaling linearly with batch size b and number of weights n:

$$\mathcal{O}(2bn)$$

As previously proposed, choose the conjugate gradient method for the solution of the Newton equation (9.1). Given that a satisfactory solution can be attained in $m \leq n$ steps, the overall complexity is $\mathcal{O}(2mbn)$. In summary, algorithm 4 requires $\mathcal{O}(bn)$ work for the underlying gradient descent routine as well as additional $\mathcal{O}(2mbn)$ for the second-order component. Its computational cost is justified by a faster convergence rate as will be explained now. Choose $c(w)$ to be twice continuously differentiable. Assume $c(w)$ possesses a positive definite Hessian on a convex set $T \subset \mathbb{R}^n$ which is aimed to be reached after pre-training. Then, the loss sequence converges with Q-superlinear rate to the minimiser w^*, i.e.

$$\|w^{k+1} - w^*\| \leq o(\|w^k - w^*\|)$$

on the **current stochastic sub-problem**. It should be stated that in the context of an image classification task this does not imply the same convergence regarding the training error, unless the network is trained on the whole data, i.e. $b = d$. In comparison, (with the necessary assumptions) pure gradient descent converges R-linearly and stochastic gradient descent sublinearly. Throughout this work, the analysis of second-order methods was limited to merely showing superlinear convergence. Additional assumptions on the regularity of $\nabla_w c(w)$ for $w \in T$, namely Lipschitz continuity, would however lead to quadratic convergence. Such regularity could be imposed on the network by design with an appropriate layer architecture.

9.3. On parallelisation

The following section will deal with computational parallelisation and the difficulties therein regarding deep neural network training. To motivate those considerations, note that the implementation of loss of a neural network is typically batch-wise additive.

Definition 9.1. Given a number of functions $c_i : S^{b_i} \times L^{b_i} \times \mathbb{R}^n \to \mathbb{R},\ i = 1, \ldots, a$, the function

$$c : S^{\sum_{i=1}^{a} b_i} \times L^{\sum_{i=1}^{a} b_i} \times \mathbb{R}^n \to \mathbb{R}$$

is called batch-wise additive w.r.t c_i, if the identity

$$\sum_{i=1}^{a} c_i(X_i, Y_i, w) = c(X, Y, w)$$

holds for

$$\bigcup_{i=1}^{a} X_i = X,\ \bigcup_{i=1}^{a} Y_i = Y$$

Corollary 9.2. *The loss $c(X, Y, w)$ of a deep neural network is batch-wise additive w.r.t any number of stochastic approximations defined on a disjoint partition of the training data.*

Intuitively, this seems to promote parallel processing of an optimisation algorithm. Sadly, the sequential nature of weight updates severely limits possibilities. The prediction of a neural network depends on its weights, which in turn cannot be updated batch-wise. Advances in the field are thus confined to increasing the computational work required in each iteration to increase opportunities for parallelisation. However, there has been effort to break the sequential nature of neural network training altogether in the form of algebraic methods [24] that factor in the dependence between weight updates and in asynchronous techniques [53]. Ba et al. propose in [4] a method that computes quasi-Newton updates asynchronously from the rest of the optimisation and accept possible mismatch as additional stochastic variance. A downside of this approach is lacking reproducibility caused by dependence on computational capacities.

10. Experimental setup

10.1. Simple CNN

Network layout

For prototyping purposes a simple convolutional network (CNN) was created. Its architecture consists of one convolutional and two fully-connected layers. The precise layout is as follows: As input, the network takes a 28×28 pixels greyscale image, i.e. a single-track square tensor (as opposed to RGB images with three tracks). The signal is passed to a convolutional layer with a 4×4 pixels filter and stride one along both dimensions. Padding is added to prevent the signal from shrinking to a square tensor of dimensions

$$28 - (4 - 1) = 25$$

and to maintain the same output of 28×28 pixels. As activation function, ReLU was chosen. The signal is then vectorised to shape $(1, 784)$ and successively right-multiplied by the filters $W_1 \in \mathbb{R}^{784\times4}$ and $W_2 \in \mathbb{R}^{4\times10}$ followed by sigmoid and softmax activation functions respectively. To calculate the final loss, categorical cross entropy is employed. In total, this amounts to

$$4 \cdot 4 + 784 \cdot 4 + 4 \cdot 10 = 3192$$

trainable variables. Regarding the implementation, the images' pixel value range was scaled dividing by the maximum (white) value 255.

MNIST

The network was trained on batches of the MNIST hand-written digits data set [19]. The data consists of greyscale images of 28×28 pixels which are partitioned in a training set of size $50,000$, a validations set of size $10,000$ and a test set of size $10,000$. Each image shows a hand-written number from 0 to 9. The corresponding labels were one-hot encoded for the purpose of training.

Regularity

Much of the numerical analysis in this work assumes the loss function $c(w) \in \mathcal{C}^2(\mathbb{R}^n)$, twice continuously differentiable. It is an important prerequisite for guaranteeing a well-posed Newton equation and convergence of the employed numerical methods. However,

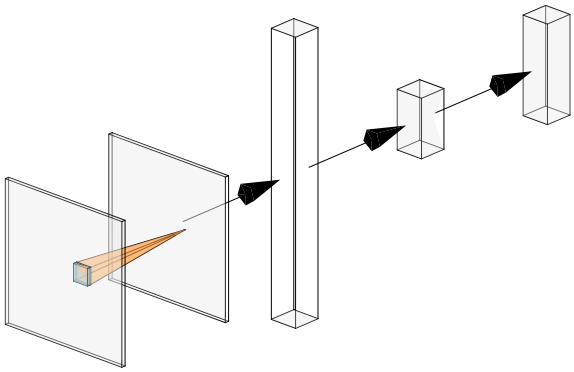


Figure 10.1.: Layout of the simple CNN. Convolutional layer followed by flattening and two fully-connected layers

Figure 10.2.: Sample images from MNIST training set (hand-written digits)

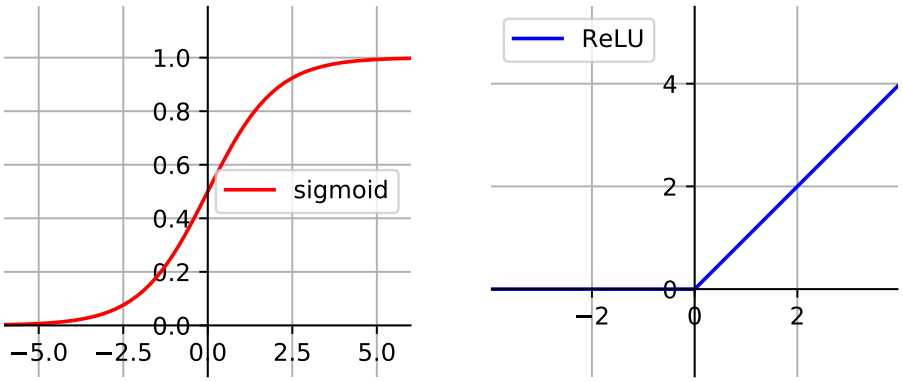


Figure 10.3.: Sigmoid (left) and ReLU (right) activation. While sigmoid is smooth, it suffers from vanishing gradients due to close-to-constant behaviour for small and large inputs. ReLU is not continuously differentiable at zero, yet standard for state-of-the-art deep learning architectures

this assumption is violated in this simple CNN, since ReLU activation is used after the convolutional layer. This is defined as

$$\phi_{\text{ReLU}}(t) = \max\{0, x\} = \begin{cases} 0 & \text{if } t < 0 \\ x & \text{if } t \geq 0 \end{cases}$$

for $x \in \mathbb{R}$ and thus only continuous, but not continuously differentiable. The reasons ReLU activation is usually chosen over the twice continuously differentiable sigmoid function

$$\phi_{\text{sigm}}(t) = \frac{1}{2}\left(1 + \tanh\left(\frac{t}{2}\right)\right)$$

are manifold. Experiments in the field of image classification have shown that it can be beneficial for a deep neural network to obtain a sparse representation of the data [41], for which the ReLU with its hard zero is naturally suited [11]. On the other hand, the non-linearity can block backpropagation through the corresponding neurons, giving rise to a problem called dying ReLU [22]. It is characterised by certain neurons becoming unresponsive, as their gradients are equal to zero. This can also happen for sigmoid activation both for very high and low x, because the function is nearly constant in those domains. The phenomenon is sometimes referred to as saturation. These effects are summarised under the term vanishing gradient. Experiments on the simple CNN have shown ReLU activation to be superior to sigmoid for this network, since more weights across all layers could be reached during training. While sigmoid activation is prone to causing vanishing

gradients, ReLU can additionally suffer from the opposite: exploding gradients [14] and thus weights. This is due to unboundedness of the linear component.

Certain methods were proposed to deal with this problem. Some argue that proper weight initialisation is the solution [10][26]. For the simple CNN, this was tested and satisfactory results were achieved. The details will be explained in the next chapter. However, second-order optimisation techniques usually benefit from their strong theoretical frameworks, that factor in the assumption of regularity. Additionally, unstable activation functions may negatively influence the conditioning of the Newton equation. Based on those conjectures, a different network needed to be considered for further experiments. There have been attempts to design continuously differentiable activation functions with possibly learnable parameters [34]. However, by far the most prominent ansatz to avoid the aforementioned problems is batch normalisation [17]: internal covariate shift, or saturated non-linearities, are relaxed by incorporating input normalisation into the network architecture. This implies including two learnable parameters to each activation, namely mean and variance. Since this architecture can possibly harmonise with the twice continuously differentiable sigmoid function, good synergy with second-order optimisation is expected. A popular network layout that incorporates batch normalisation is the so-called ResNet.

10.2. ResNet

The academic neural network described above was useful for prototyping of the optimisation routine, but possesses little technical significance. Its shallow layout allows for fast backpropagation making both evaluation and storage of its Hessian $H_c \in \mathbb{R}^{3192\times3192}$ possible on low-end processors. In the domain of deep neural networks, additional problems arise. As benchmark of technical relevance, the popular ResNet [15] was chosen. With around 20 million trainable variables in the 50-layer version, more thought has to be put into the optimisation algorithm. Assuming 32-bit (4 byte) single precision, the required storage space for a dense Hessian becomes extensive at approximately $1,600$ terabyte. Thus, the training algorithm has to be designed more carefully.

Network layout

In the experiments, a 50-layer ResNet architecture is considered. It is comprised of a plain network and residual connections. The plain network consists of convolutions with filters of 3×3 pixels where downsampling is performed by a stride of two. After each convolution, batch normalisation is adopted. At the end, global average pooling is employed followed by a 1000-way fully-connected layer with softmax loss. Based on the plain network, residual connections are incorporated. These are denoted by the additional errors on the right in figure 10.4. The dotted lines indicate a connection that increases the signal dimensions. In total, the network has 3.6 billion FLOPs (multiply-adds) and

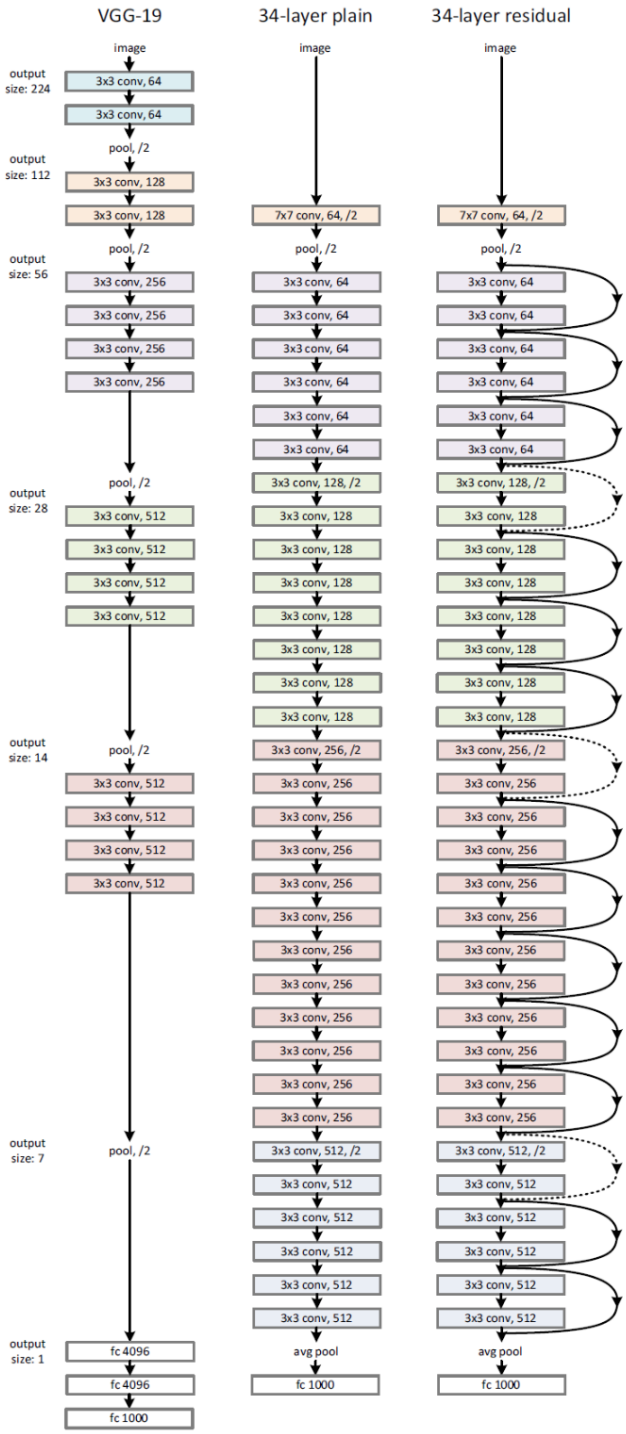


Figure 10.4.: ResNet visualisation [15]. VGG-19 [40] (left) for reference, plain network (middle) and residual layout (right). Dotted lines indicate an increase in dimensions.

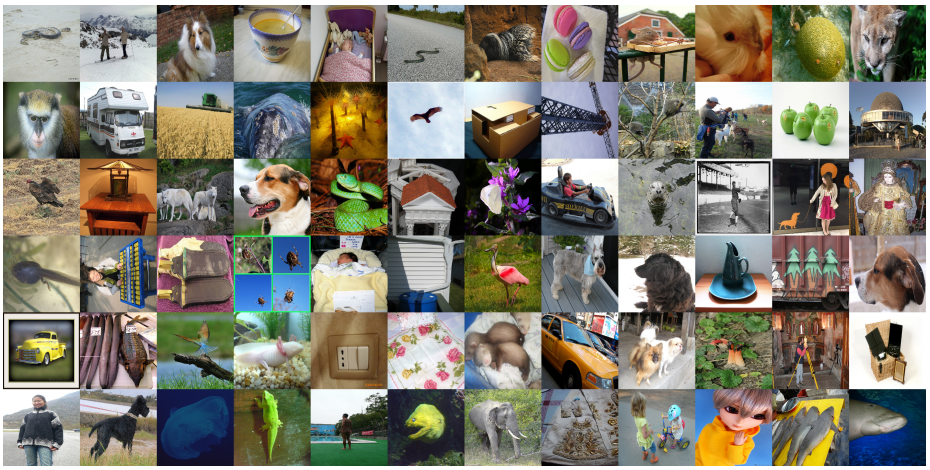


Figure 10.5.: Samples from the ImageNet validation set (ILSVR 2012)

contains $23,944,392$ trainable weights. Regarding implementation, the input images were resized to triple-track (RGB) 224×224 pixels tensors using OpenCV's `cv2.resize()`. ResNet is available in TensorFlow via Keras through the `tf.keras.applications.resnet.ResNet50()` class. A downside is that the model works with 32-bit single precision which cannot be adjusted without adapting the source code. Keras provides a set of weights that were pre-trained on ImageNet using images that were not normalised, i.e. with pixel values from $[0, 255]$.

ImageNet

As training input, the data set from 2012 ImageNet large-scale visual recognition (ILSVR 2012) challenge [36] was used. It consists of RGB images of various size depicting an animal, person or object from 1000 different classes. There are approximately 1.28 million images in the training set, $50,000$ validation and $100,000$ test images. The labels were one-hot encoded for training.

11. Weight initialisation

In respect of section 10.1, activation functions play an important role in the convergence of neural network training. There, the concepts of vanishing gradients, exploding gradients and dying ReLU were briefly described. These phenomenons distort the backpropagation by making large parts of the network inaccessible or scale them to enormous values. ReLU or sigmoid activation can get "stuck" and thereby lead to inefficient weight updates.

Example

For illustration, consider the first fully-connected layer in the simple CNN from section 10.1. It consists of the right-multiplication

$$(x_{\text{conv}})^{\mathsf{T}} W_1$$

where $x_{\text{conv}} \in \mathbb{R}^{784}$ is the vectorised signal passed from the convolutional layer (which maps from S to $\mathbb{R}^{784}$) and $W_1 \in \mathbb{R}^{784\times 4}$. If the filter W_1 was naïvely initialised by (e.g.) ones, it is easy to see how even moderate x_{conv} will blow up the output and saturate the following sigmoid activation. To this end, sample the entries of x_{conv} from a normal distribution. This is to represent the colour spectrum of an image-based signal. Since the images are scaled beforehand, assume $x_{\text{conv},i} \sim \mathcal{N}(0.5, 0.25),\ i \in [1, 784]$ independently. Then, the expected value of the four output entries each sums up to

$$\mathbb{E}\left[\sum_{i=1}^{784} x_{\text{conv},i}\right] = \sum_{i=1}^{784} \mathbb{E}\left[x_{\text{conv},i}\right] = 784 \cdot 0.5 = 392$$

which is high enough to saturate the sigmoid activation. This means driving the output into its near constant regime and thus hindering the gradient-based optimiser to make meaningful changes to corresponding weights.

Implication for non-convex optimisation

Some authors propose that proper weight initialisation, i.e. starting the training from a carefully chosen initial point, fixes the aforementioned issues [10][14][22][26]. Indeed, saturating, killing or blowing up neurons can be avoided by simply inputting moderate values. To set the optimisation algorithm up with the best possible chances of success, proper weight initialisation is crucial. Especially in non-convex optimisation, where convergence to a global minimum is highly unlikely, the local minimum found depends entirely on the

initial vector. Regarding the benefits of "killing" neurons and a sparse data representation causing an increase in performance [41][11], it should be stated that this is typically an increase in classification accuracy, but not in optimiser performance. Since the latter is subject of this work, thorough attention had to be paid to weight initialisation.

Xavier-inspired initialisation

For the fully-connected layer of the simple CNN above this means requiring the expected value of the outputs to be around zero. Additionally, the variance is scaled by the dimensions for numerical stability. The finally chosen initialisation scheme was

$$W_j \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{j, in}} + n_{\text{j, out}}}}\right), \; j \in [1, 3]$$

where W_j are the filter matrices, $n_{\text{j, in}}$ the input (fan-in) and $n_{\text{j, out}}$ the output dimension (fan-out). It is based on the so-called Xavier initialisation first proposed by Xavier Glorot et al. in [10]:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

Since then, many different initialisation schemes have been proposed, e.g. asymmetric methods to account for ReLU's asymmetry [22] and sequential ones that trace the signal through layers hence factoring in the network's depth [26].

12. Related research

Additional approaches to second-order training for deep neural networks are manifold. In-depth analysis is conducted regarding the loss landscape, using curvature information obtained through statistical algorithms [49][12]. This paves the way for well-understood training methods and potentially a stronger theoretical foundation. In the stochastic domain, Agarwal et al. apply a stochastically approximated Hessian inverse with linearly-scaling work in second-order optimisation [2]. Pilanci et al. propose in [33] a quasi-Newton method which in every step uses a randomly sub-sampled Hessian that achieves superlinear convergence. In the deterministic domain, promising ideas are cubic regularisation and trust-region methods [27][47] which are able to deal with negative curvature and thus non-convex domains. This is achieved by introducing additional constraints to the optimisation. Moving on to a larger-scale proof-of-concept, Osawa et al. applied K-FAC-powered second-order training to ResNet on ImageNet [31]. Another promising advance is the inclusion of second-order information obtained in an efficient way to first-order optimisation. Yao et al. make use of this in curvature-based batch size adaption [48] for SGD in order to escape regions prone to adversarial attacks.

Part IV.

Numerical experiments

13. Quality of numerical Hessians

An important precursor to numerical second-order optimisation is the availability of stable and sufficiently precise (quasi-)Hessians. Algorithmic differentiation supplies this need by recursively passing gradients through the network in an exact manner. Together with the tools provided in chapter 4 and section 9.1 this makes the exact Hessian or an Hessian-vector product readily available. This computational bottleneck of second-order training was implemented in the context of this work for the simple CNN from section 10.1 and ResNet. To confirm numerical stability, those implementations were thoroughly tested.

Taylor approximation error

A simple mathematical measure to check the quality of a numerically computed Hessian matrix is the Taylor-approximation error. Consider the mapping $c : \mathbb{R}^n \to \mathbb{R}$. Taylor approximation around $w = \bar{w}$ yields

$$c(w) = c(\bar{w}) + \nabla_w c(\bar{w})^\mathsf{T}(w - \bar{w}) + \frac{1}{2}(w - \bar{w})^\mathsf{T}\mathrm{H}_c(\bar{w})(w - \bar{w}) + o(\|w - \bar{w}\|^2) \tag{13.1}$$

Note that the error term depends quadratically on the norm of $(w - \bar{w})$. For comparison, the function is additionally approximated to the first order:

$$c(w) = c(\bar{w}) + \nabla_w c(\bar{w})^\mathsf{T}(w - \bar{w}) + o(\|w - \bar{w}\|)$$

where the error scales with $\|w-\bar{w}\|$. To examine the Hessian quality, the error term in (13.1) is now evaluated at certain distances from $\bar{w}$. If the numerical Hessian is precise, linear convergence is observed towards the approximation point. It should be noted that if $c(w)$ is affine, there will be virtually no difference between the first- and second-order Taylor approximation. This stems from the fact that in a domain with negligible curvature the Hessian H_c will be close to zero. However, in a neighbourhood $w \in K$ around a minimum w^* where $c(w)$ is positive definite and thus strictly convex, there will be a clearly visible difference in quality. It is thus fair to assume that the second-order approximation gets better near a minimum. From this, it can be inferred that second-order optimisation gains an advantage over first-order optimisation in the vicinity of a minimum.

Working precision

The standard numeric data type for deep neural network training is 32-bit single precision (`tf.float32` in TensorFlow). Even though numerical stability would favour higher precision,

as was made plausible for Euclidean norms in section 9.2, the extensive amount of required memory limits the possibilities. Especially in the distributed-computing setting, single precision is chosen over 64-bit double precision (`tf.float64` in TensorFlow), to get rid of computational overhead. In second-order training, some go as far as to reduce the data type to half precision [31] to be able to deal with explicitly evaluated (quasi-)Hessians in approximately 20 million dimensions. While the efficient Hessian-vector products takes away some of the complexity, still a large amount of RAM is required, especially with increasing training batch sizes.

TensorFlow

The recently released TensorFlow version 2.0 introduced significant changes compared to earlier versions. Most importantly, the session-based execution of a previously built graph was abolished in favour of eager execution to fit natively into the Python framework. While the efficient Hessian product can still be implemented without major difficulties, evaluating the Hessian explicitly is more demanding due to the removal of the efficient `tf.Hessians()` function. It can still be done easily, however less efficiently, by taking the Jacobian of the gradient, see section 9.1.

14. Simple CNN

The simple CNN with its moderate scale allowed for numerical experiments using the 64-bit double precision (`tf.float64` in TensorFlow) numeric data type. The switch from default was motivated by tests that showed unsatisfactory Hessian quality at single precision. For reproducibility, Experiments on the simple CNN were conducted on a home computer with a 2,7 GHz Intel Core i7 processor and 4 GB 1333 MHz DDR3 RAM. Python 3.6.8 was used along with TensorFlow 2.1.0

14.1. Explicit Hessian

For experiments with the fully evaluated Hessian, first a sanity check was performed. On a single-sample batch, an explicit Hessian was computed with the TensorFlow version two framework. This took on average 35.447 [s] across seven runs. For comparison TensorFlow version one was used which took 7.461 [s] on average and the difference between the two Hessians evaluated to $3.19\mathrm{e}{-16}$ in the Frobenius norm

$$\|A\|_{\mathrm{F}} \coloneqq \sqrt{\sum_i \sum_j |A_{ij}|^2}$$

which is zero to working precision. Next, the computation-time scaling regarding batch size was examined. TensorFlow version one was used. Table 14.1 summarises the findings. In conformity with the previous theoretical analysis, approximately linear scaling can be observed. The numerical experiments were concluded by the evaluation of the Taylor

batch size	1	10	100	1000
comp. time [s]	6.529	15.722	145.366	1256.274

Table 14.1.: Evaluation time of full Hessian. Expected batch size scaling is linear

approximation error. To this end, a batch size of $b = 100$ samples was chosen. In order to be sufficiently close to a minimum, $k = 150$ gradient descent iterations were performed. The test direction, in which the distance was measured, was naïvely chosen to be a uniform (unit) vector across all 3192 dimensions. The results are presented in figure 14.1. In the logarithmic scale, linear and quadratic convergence correspond to a slope of one and two, respectively. This can be shown by the simple calculation

$$|y| = |x|^s \Leftrightarrow \log(|y|) = s \log(|x|)$$

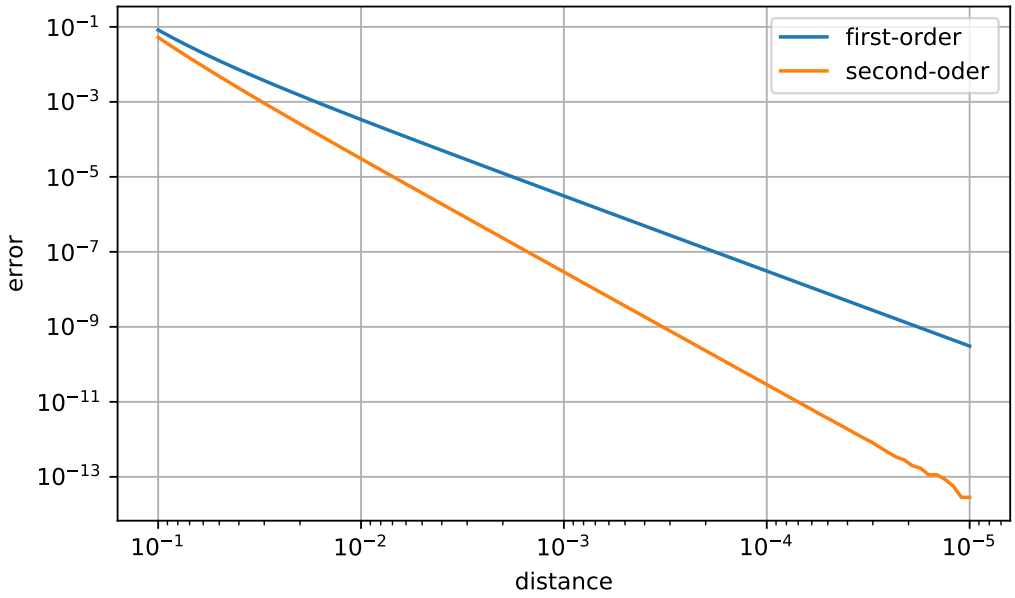


Figure 14.1.: First- and second-order Taylor approximation error for simple CNN with $b = 100$. Logarithmic scale. Linear and quadratic error convergence can be observed through an approximate slope of one and two, respectively

The expected convergence rates of the firs- and second-order Taylor approximation can be observed in the plot. It can therefore be concluded that the Hessian possesses sufficient numerical quality.

14.2. Hessian product

As a sanity check, the explicit Hessian was left-multiplied with a random direction s and compared with the Hessian product for a batch size $b = 100$ (and random weights). The 2-norm (or Euclidean norm) of the remainder was $2.40\text{e}{-13}$ which is sufficiently small in double precision, given that the high-dimensional Euclidean norm is prone to numerical variance (see section 9.2). Computation times of the efficient Hessian-vector product are given in table 14.2. Again, approximately linear scaling can be observed, as expected. With this technique at hand, it is possible to scale the batch size to the maximum and work with the full training set of $60,000$ samples (MNIST).

batch size	1	10	100	1000	10000
comp. time [s]	0.030	0.064	0.204	1.173	11.580

Table 14.2.: Evaluation time of Hessian-vector product. Expected batch-size scaling is linear

14.3. Second-order optimisation

Next up, the template algorithm explained in section 9.2 was compared against naïve gradient descent in a practical scenario. Leveraging the low computation times of the Hessian product, a full-batch approach with $b = d = 60,000$ was taken. For globalisation purposes, the quasi-Newton method operates with Armijo line search. For the gradient descent, line search proved impractical. This is due to the cost function being coercive, i.e. growing rapidly at the limits of $\mathbb{R}^n$. Thus, not for every cost function evaluation, a (numerically) finite value can be guaranteed, hindering the trial-and-error-based line search. Note that second-order methods do not suffer from this drawback, since they implicitly find a moderate step.

Consequently, an appropriate learning rate had to be found for the gradient descent in the context of hyperparameter tuning. In each iteration, the weights were updated a fixed distance in the direction of steepest descent. The results can be seen in figure 14.2. For convenience, the optimisation was done on 28 cores of a supercomputing cluster made up of several Intel Xeon E5-2690 v3 "Haswell" processors and with a supply of 64 GB of RAM (DDR4). However, it could have easily been done on the home system. On the supercomputer, one gradient evaluation clocked in at 1.523 [s] and one Hessian product was processed in 3.742 [s]. Factoring in overhead, this roughly confirms the previously

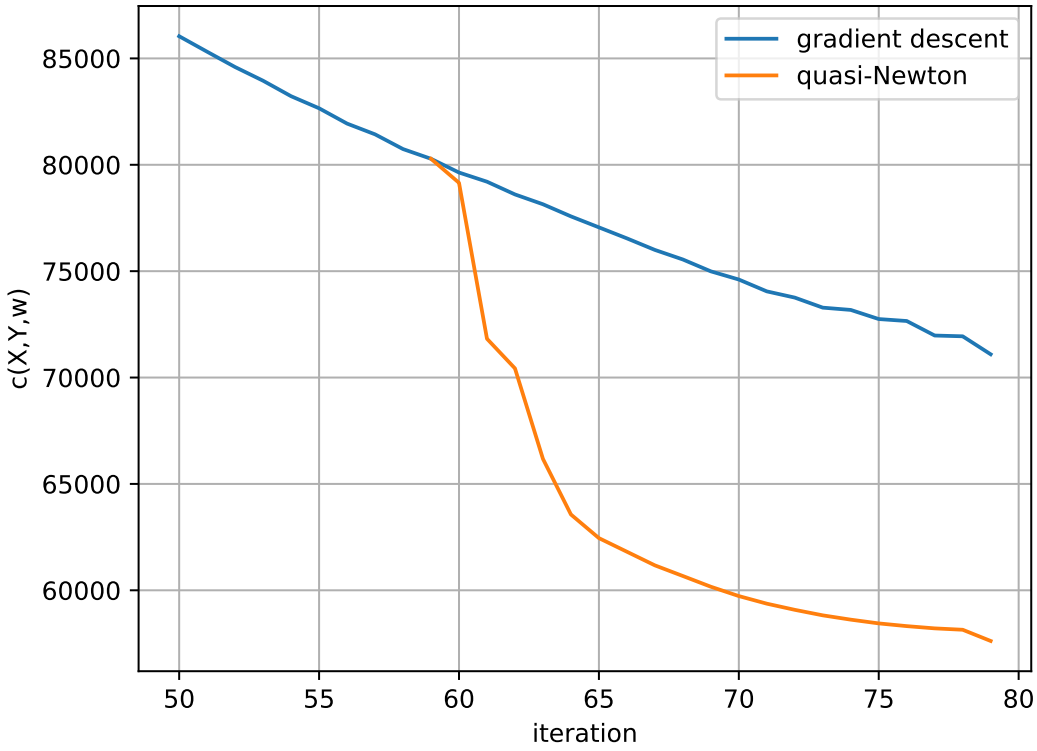


Figure 14.2.: Per-iteration cost of quasi-Newton method compared to gradient descent for simple CNN on MNIST (full training data $b = d = 60,000$). Comparison starts after pre-training

established theory of $\mathcal{O}(d)$ work and $\mathcal{O}(2d)$ work, respectively. In a pre-training period, 60 gradient descent iterations were executed, followed by 20 training iterations per method.

The conjugate gradient (CG) method was warm-started from the normalised weight vector

$$\frac{w^k}{\|w^k\|_2}$$

for faster convergence. On average, CG required $\bar{m} = 7.8$ iterations which each require one Hessian product. This means that the average quasi-Newton step took ca. 15.6 times the work of a gradient descent step as additional overhead. While the per-iteration cost alone cannot justify using the second-order method, the potential for HPC applications is evident.

15. ResNet

Computing the Hessian product posed more of a challenge for ResNet than for the simple CNN. Needless to say, evaluating the Hessian explicitly is infeasible in approximately 20 million dimensions. For this reason, computation was moved to a supercomputing cluster. There, 28 cores, made up of several Intel Xeon E5-2690 v3 "Haswell" processors were used, with a supply of 64 GB of RAM (DDR4). Even with this computing power at hand, the maximum practicable batch size from the ImageNet training data turned out to be $b = 16$. Reasonable larger-scale experiments would require parallelisation efforts or approval of extensive RAM consumption. This, in turn, hinders meaningful studies of the Hessian-product computation time. However, a simple proof-of-concept for the algorithm proposed in section 9.2 along with other investigations were possible.

15.1. Loss surface

The shape of the n-dimensional loss landscape plays an important role in optimisation. Especially second-order methods depend highly on the quality of their underlying approximation. As previously established, those methods get better in the neighbourhood of a minimum where the cost function is close to strictly convex. On the other hand, saddle points can hinder convergence, as the Hessian becomes indefinite which causes problems for CG. Second-order methods rely on the shape of the loss landscape more heavily than first-order methods, because they include curvature in addition to slope. It can thus be insightful to examine the loss landscape. This is usually done by computing principal eigenvectors of H_c, i.e. those with the highest absolute value, via a so-called power iteration. Given the principal eigenvector v^1, the largest absolute eigenvalue λ_1 can then be

Data: number of iterations
Result: approximation to principal eigenvector of H_c in unit length
sample $b^0 \sim \mathcal{N}(0, 1)$
normalise b^0 to unit length
for k *in* 0 *to number of iterations* **do**
  $b^{k+1} \leftarrow \mathrm{H}_c b^k$
  normalise b^{k+1} to unit length
end
return b^{k+1}

Algorithm 5: Power iteration

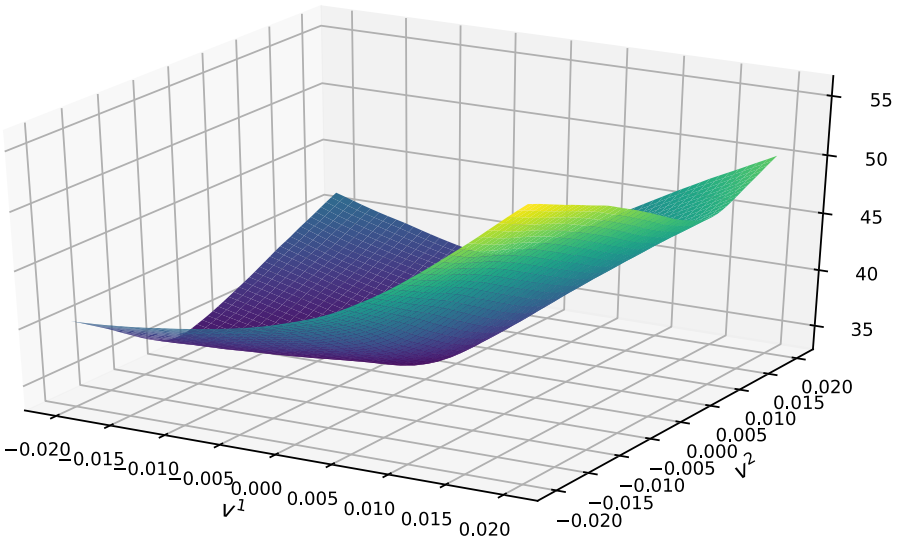


Figure 15.1.: Loss surface of ResNet (trained on ImageNet), plotted along the first two principal directions with eigenvalues $\lambda_1 = 131,942.86$ and $\lambda_2 = 101,590.70$

computed by calculating the norm $\|\mathrm{H}_c(v^1)\|_2$ since by the eigenvector equation

$$\mathrm{H}_c(v^1) = \lambda_1 v^1$$

Once the first eigenvalue is computed, the second eigenvector is obtained by repeating the procedure with

$$\bar{\mathrm{H}}_c = \mathrm{H}_c - \lambda_1 v^1 (v^1)^\mathsf{T}$$

To make this plausible, recall that

$$\mathrm{H}_c = \sum_{i=1}^{n} \lambda_i v^i (v^i)^\mathsf{T}$$

The principal directions of the Hessian can be interpreted as the directions of maximum curvature of the loss function $c(w)$ in the weight space. Neglecting all but the first two eigenvectors yields an approximate loss surface. This is illustrated in in figure 15.1 for the given implementation of ResNet, pre-trained on ImageNet. The corresponding approximate eigenvalues are $\lambda_1 = 131,942.86$ and $\lambda_2 = 101,590.70$, computed in 20 power iterations each. Since the network was initialised with pre-trained weights, a minimum can be observed along those (highest-curvature) directions.

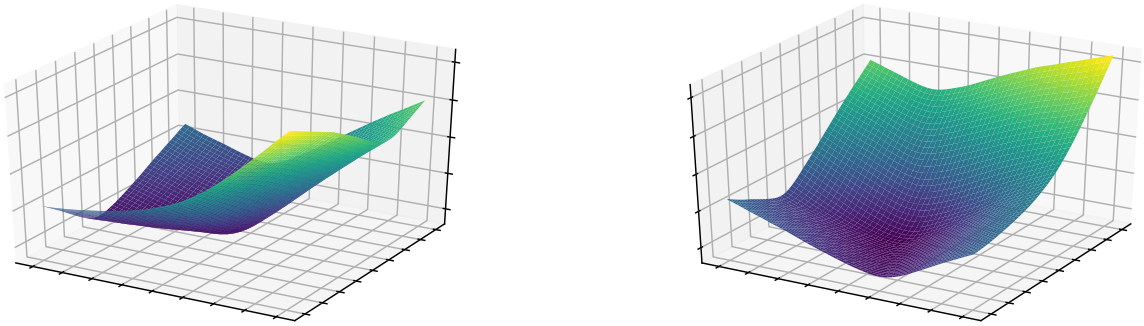


Figure 15.2.: Loss surface from figure 15.1 rotated 90° counter-clockwise

15.2. Hessian product

Similar to the simple CNN, the quality of numerical Hessian was evaluated through Taylor approximation error. For ResNet however, no explicit Hessian could be computed, so the efficient Hessian product was used. As stated before, in order to observe a difference in approximation error between first- and second-order approximation, the cost function must not be affine along the test direction. If a random vector is chosen in a high dimensional space, there is a chance the chosen direction is affine. To guarantee non-zero curvature, the Taylor approximation error was tested along the eigenvector corresponding to the largest eigenvalue of H_c. The error is given in table 15.1 Computing the Hessian product took

distance	0.1	0.01	0.001	1e−4	1e−5
err. (1st)	70.9687	3.37188	0.0773926	0.000835419	5.722 05e−5
err. (2nd)	588.7740	3.22557	0.0114250	0.000175476	4.959 11e−5

Table 15.1.: Error of first- and second-order Taylor approximation to the loss function of ResNet along the principal direction of H_c. Linear and quadratic convergence can be observed. Convergence plateaus in the end due to implementation-forced (TensorFlow) single precision limiting the range for analysis

20.577 [s] on average across four evaluations on the supercomputer. While quadratic convergence of the second-order approximation can be observed in the first four steps, the error plateaus in the end. This happens because the robustness against numerical variance is "saturated" for the extremely high-dimensional problem near machine precision, compare section 9.1. This imposes a limit on the range where meaningful analysis is possible. Due to the coerciveness of $c(w)$, further distances lead to numerically infinite values. It can be concluded that, to unlock the full potential of second-order approximations and therefore quasi-Newton methods, double precision is necessary. Unfortunately, this would have required re-implementing ResNet which exceeded the scope of this work.

15.3. Second-order optimisation

Concluding the numerical experiments, the template second-order optimisation routine from section 9.2 was applied to the large-scale ResNet in a proof-of-concept. To this end, a mini-batch of size $b = 16$ was used from ImageNet, as stated before. Scaling the problem up to larger batch sizes is possible on state-of-the-art supercomputers (e.g. the aforementioned cluster) but to be done properly, requires parallelisation of the per-iteration computational work. Additionally, code optimisation in terms of overhead is necessary.

To create the experimental setup, the pre-trained minimum was perturbed by multiplying the weight vector w with a n-dimensional random vector

$$p \sim \mathcal{N}(1, 0.01)$$

eliminating the need for pre-training. As comparison and fall-back method, naïve gradient descent was implemented, using a learning rate for step size regulation. Six sample runs were performed and the results can be seen in figure 15.3. The conjugate gradient (CG) method required $\bar{m} = 4.4$ iterations on average. It can be observed that the second-order method consistently out-performs the first-order method in terms of convergence. Further increase of this advantage is expected when changing from single to double precision.

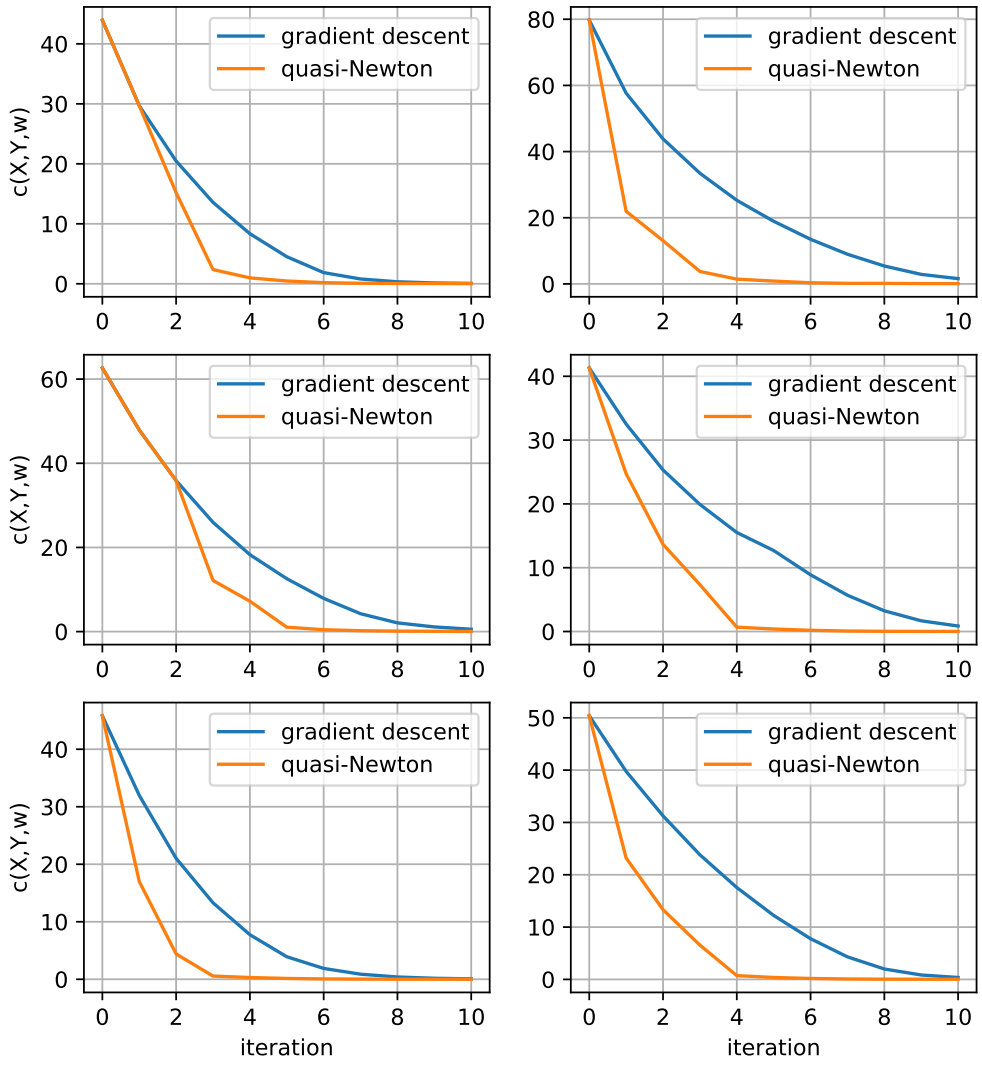


Figure 15.3.: Proof-of-concept of second-order optimisation for ResNet (on ImageNet with batch size $b = 16$). Start from randomly perturbed minimum (pre-trained). Algorithm 4 is compared against gradient descent

Part V.

Conclusion

Summary

In this work, a template for second-order optimisation tailored to application in the training of large-scale deep neural networks was motivated, established and analysed. To substantiate the theoretical claims, the algorithm was tested in a practical environment alongside other investigations. Through application to the popular ResNet on ImageNet, the potential of second-order optimisation in large-scale deep learning could be demonstrated. A short recap of the thesis work including the most important findings will be given in the following.

Deep neural networks were briefly introduced and the connection to mathematical optimisation was established. It was found that the associated cost function can be constructed smooth and thus suitable for second-order optimisation. In batch-wise input handling, a heuristic implementation detail was discussed and brought into conformity with rigorous mathematical analysis. Afterwards, the optimisation problem was formally stated and the optimality conditions were derived. Based on the preliminaries, gradient descent and the Newton method were introduced. In particular, the relation between Euclidean space and gradient descent was clarified and the importance of quadratic approximation for the Newton equation was indicated. Since the second-order method is not globally convergent for every starting point, globalisation measures were proposed. Algorithmic differentiation was briefly discussed and the term "backpropagation" was motivated. In anticipation of computational infeasibility regarding an explicit Hessian evaluation, a trick was presented to compute an implicit Hessian-vector product, reducing the complexity from n gradient calculations down to two. Subsequently, a brief overview was given over solution methods for the Newton equation in the context of large-scale deep learning. Conjugate gradient (CG) was found as a well-suited algorithm that can be used to exploit the aforementioned Hessian-vector trick. As an important detail, CG converges monotonically in exact arithmetic.

In a short digression, a first-order optimisation technique using CG was derived, based on a different formal problem statement, to contrast with the above.

Moving further towards practical application, the impact of strict convexity regarding the objective function on the solution of the Newton equation was formally examined. It was found that applicability is rather a question of positive definiteness and these two do not necessarily coincide. Heuristic measures to deal with ill-conditioning were introduced in pre-training and Tikhonov regularisation. The notion of stochastic approximation was explained and applied to both gradient descent and the Newton method, bridging the gap to common deep learning practice. Considering a stochastic sub-problem, gradient descent converges with R-linear rate under certain circumstances. Motivation was given for the use of second-order over first-order methods. In particular, they are better suited to take advantage of extensive computing power. A stochastic modification of the Newton method was introduced, with Q-superlinear convergence regarding the stochastic sub-problems.

Moving on to the implementation details, a template second-order optimisation routine

for large-scale deep learning was proposed. Fall-back steepest descent and line-search were introduced as globalisation measures. In total, the additional overhead compared to gradient descent regarding the stochastic sub-problem was calculated to be $\mathcal{O}(2mbn)$, where b is the batch size and m the number of CG iterations. Additionally, the impact of dimensionality on the Euclidean norm was made plausible in a stochastic sense and the challenges of parallelisation for optimisation were characterised. The experimental setup was discussed. Motivating the transition from an academic example to ResNet, the interplay between activation functions and optimisation was sketched, including dying ReLU, vanishing gradients and batch normalisation. Lastly, the importance of careful weight initialisation for neural network training was demonstrated and a Xavier-like scheme was proposed. To build on the template algorithm, a collection of relevant approaches to second-order optimisation for large-scale deep learning was given.

Concluding the work, above established theory was implemented in Python 3 using TensorFlow 2.0 and thoroughly tested. Adopting the efficient Hessian product in the optimisation routine made scaling up the batch size to $b = 60,000$ (MNIST training set) for the simple CNN possible. The proposed algorithm outperformed gradient descent consistently across all test cases (simple CNN and ResNet), demonstrating its potential. Additionally, the fast Hessian product inspired a loss landscape study powered by eigenvalue analysis. Even though the ResNet optimisation was computed on an HPC cluster, a technical ceiling was hit, overcoming which would require parallelisation of the per-iteration computational work.

Perspectives

As was the goal of this work, a template for the incorporation of new approaches is now readily available. The rapidly evolving field of deep neural network training offers many promising ideas. Given more time, cubic regularisation and trust-region methods [47] as well as preconditioning for conjugate gradient [8] would have been investigated to address the issues of lacking global convergence and ill-conditioning. Another interesting ansatz for the latter is to expand the second-order approximation and "approximate positive definitely" through the generalised Gauss-Newton matrix $(\mathrm{J}_c)^\mathsf{T}\mathrm{H}_c\mathrm{J}_c$. The counter-approach to developing increasingly robust optimisation methods is creating deep neural networks that are easily optimisable by design. Particularly in the field of optimal control, where sooner or later learning and decision making is supposed to be performed in real-time, this appears relevant. Regarding second-order optimisation, a potential starting point would be guaranteeing Lipschitz-continuity of the Hessian which leads to Q-quadratic convergence of the Newton method. What is more, a geometric approach could be taken to incorporate manifold learning into optimisation: Considering the dependence of gradient descent on the Euclidean space, methods could be created exploiting the underlying manifold of the training data.

Appendix

A. Convergence rates

Convergence rates quantify the convergence of a sequence to some limit. They are a common tool in optimisation and deep learning literature. It can be useful to put them in direct comparison. For convenience, the convergence rates used throughout this work are listed here:

- **Sublinear.**
$$c(w^k) - c(w^*) \leq \mathcal{O}\left(1/k\right)$$
- **R-linear.**
$$c(w^k) - c(w^*) \leq \mathcal{O}(\rho^k),\ \rho \in (0,1)$$
- **Q-linear.**
$$c(w^{k+1}) - c(w^*) \leq \rho \cdot (c(w^k) - c(w^*)),\ \rho \in (0,1)$$
- **Q-superlinear.**
$$\|w^{k+1} - w^*\| = o(\|w^k - w^*\|)$$

Bibliography

[1] Alekh Agarwal, Martin J. Wainwright, Peter L. Bartlett, and Pradeep K. Ravikumar. Information-theoretic lower bounds on the oracle complexity of convex optimization. In *Advances in Neural Information Processing Systems 22*, pages 1–9. Curran Associates, Inc., 2009.

[2] Naman Agarwal, Brian Bullins, and Elad Hazan. Second order stochastic optimization in linear time. *ArXiv*, abs/1602.03943, 2016.

[3] Joel A. E. Andersson, Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019.

[4] Jimmy Ba, Roger B. Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. In *ICLR*, 2017.

[5] Raghu Bollapragada, Dheevatsa Mudigere, Jorge Nocedal, Hao-Jun M. Shi, and Ping-Tak P. Tang. A progressive batching L-BFGS method for machine learning. *ArXiv*, abs/1802.05374, 2018.

[6] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60:223–311, 2016.

[7] Frank E. Curtis and Wei Guo. R-linear convergence of limited memory steepest descent. *IMA Journal of Numerical Analysis*, 38, 2016.

[8] Yann Dauphin, Harm de Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *NIPS*, 2015.

[9] Ron S. Dembo, Stanley C. Eisenstat, and Trond Steihaug. Inexact Newton methods. *SIAM Journal on Numerical Analysis*, 19(2):400–408, 1982.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13-15 May 2010. PMLR.

[11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence*

and Statistics, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11-13 Apr 2011. PMLR.

[12] Diego Granziol, Xingchen Wan, Timur Garipov, Dmitry P. Vetrov, and Stephen Roberts. MLRG deep curvature: An open-source package to analyse and visualise neural network curvature and loss surface. In *MLRG Deep Curvature*, 2019.

[13] Andreas Griewank. *Automatic Differentiation*, chapter VI.7, pages 749–752. Princeton University Press, 2014.

[14] Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In *NeurIPS*, 2018.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[16] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–436, 1952.

[17] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

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

[20] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

[21] Yun Liu, Krishna Gadepalli, Mohammad Norouzi, George E. Dahl, Timo Kohlberger, Aleksey Boyko, Subhashini Venugopalan, Aleksei Timofeev, Philip Q. Nelson, Gregory S. Corrado, Jason D. Hipp, Lily Peng, and Martin C. Stumpe. Detecting cancer metastases on gigapixel pathology images. *CoRR*, abs/1703.02442, 2017.

[22] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying ReLU and initialization: Theory and numerical examples. *ArXiv*, abs/1903.06733, 2019.

[23] David G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, 1973.

[24] Saeed Maleki, Madan Musuvathi, and Todd Mytkowicz. Parallel stochastic gradient descent with sound combiners. *ArXiv*, abs/1705.08030, 2017.

[25] Andreas Meister. *Numerik linearer Gleichungssysteme*. Wiesbaden: Springer Spektrum, 2015.

[26] Dmytro Mishkin and Juan E. Sala Matas. All you need is a good init. *CoRR*, abs/1511.06422, 2015.

[27] Yurii Nesterov and Boris Polyak. Cubic regularization of Newton method and its global performance. *Mathematical Programing*, 108:177–205, 2006.

[28] Andrew Y. Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 78. New York: Association for Computing Machinery, 2004.

[29] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[30] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[31] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12351–12359, 2018.

[32] Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 1993.

[33] Mert Pilanci and Martin J. Wainwright. Newton sketch: A linear-time optimization algorithm with linear-quadratic convergence. *SIAM Journal on Optimization*, 27:205–245, 2015.

[34] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[35] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, 2015.

[37] Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14:1723–38, 2002.

[38] Nicol N. Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-Newton method for online convex optimization. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 436–443, San Juan, Puerto Rico, 21-24 Mar 2007.

[39] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815, 2017.

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[41] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deeply learned face representations are sparse, selective, and robust. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2892–2900, 2014.

[42] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[43] Andrey N. Tikhonov. *Numerical methods for the solution of ill-posed problems*. Dordrecht: Kluwer Academic Publishers, 1995.

[44] Andrey N. Tikhonov and Vasiliy Y. Arsenin. *Solutions of ill-posed problems*. Washington: Winston, New York: Halsted Press, 1977.

[45] Andrey N. Tikhonov, Aleksandr S. Leonov, and Anatolij G. Yagola. *Nonlinear ill-posed problems*. London: Chapman & Hall, 1998.

[46] Michael Ulbrich and Stefan Ulbrich. *Nichtlineare Optimierung*. Mathematik Kompakt. Basel: Birkhäuser, 2012.

[47] Peng Xu, Farbod Roosta-Khorasani, and Michael W. Mahoney. Newton-type methods for non-convex optimization under inexact Hessian information. *Mathematical Programming*, pages 1–36, 2017.

[48] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W. Mahoney. Large batch size training of neural networks with adversarial training and second-order information. *ArXiv*, abs/1810.01021, 2018.

[49] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W. Mahoney. PyHessian: Neural networks through the lens of the Hessian. *ArXiv*, abs/1912.07145, 2019.

[50] Zhewei Yao, Amir Gholami, Qi Lei, Kurt Keutzer, and Michael W. Mahoney. Hessian-based analysis of large batch training and robustness to adversaries. In *NeurIPS*, 2018.

[51] Kun-Hsing Yu, Ce Zhang, Gerald Berry, Russ Altman, Christopher Ré, Daniel Rubin, and Michael Snyder. Predicting non-small cell lung cancer prognosis by fully automated microscopic pathology image features. *Nature Communications*, 7, 2016.

[52] Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[53] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. In *NIPS*, 2015.