SECOND ORDER METHODS FOR NEURAL NETWORKS OPTIMIZATION

JAN NICLAS HARDTKE

Geboren am 27.Oktober 1999 in Troisdorf

Bachelor Thesis Informatik

Betreuer: Dr. Moritz Wolter
Zweitgutachter: Prof. Dr. Christian Bauckhage

Institut für Informatik II – Visual Computing

Mathematisch-Naturwissenschaftliche Fakultät Rheinische
Friedrich-Wilhelms-Universität Bonn

May 2024

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

INTRODUCTION

Optimization techniques play a crucial role in science and engineering, as they enable the refinement of models and solutions by iteratively minimizing or maximizing objective functions. This process ensures that solutions are as effective and efficient as possible, impacting a wide range of applications, from designing engineering systems to refining algorithms in computational research.
In the field of numerical optimization, we differentiate between so-called *first-order* and *second-order* optimization methods, where the former refers to solely utilizing *first-order*, i.e., gradient information, for the task of optimization.

Second-order optimization methods utilize both the gradient and Hessian information, providing deeper insights into the nature of the loss landscape.
In traditional numerical optimization problems, second-order methods are essential for efficient algorithm design as they have provably better convergence properties than first-order methods. Techniques such as Newton's method utilize the Hessian for rapid convergence.
Algorithms in this family often use a preconditioning matrix to transform the gradient before applying each step. Classically, the preconditioner is the matrix of second-order derivatives (the Hessian) in the context of exact deterministic optimization.[**anil2021scalable**]
In machine learning, the direct application of second-order information is unfortunately limited due to the computational intensity and storage requirements of handling full Hessian matrices, particularly since today's models often involve billions of trainable parameters. While first-order methods like (stochastic) gradient descent (SGD) are preferred for the training of today's models because of their simplicity and reduced computational demands, they often fall short in convergence speed and sensitivity to hyperparameter settings.
Recent practice of training large models even suggests that the utility of common first-order methods is quickly reaching a plateau, as their time-per-step is already negligible. Consequently, the only way to accelerate training is by reducing the number of steps taken by the optimizers to reach a solution.[**anil2021scalable**]
Therefore integrating Hessian-based information can potentially improve optimization efficiency by drastically increasing training convergence. In this thesis, we will focus on two novel approaches for neural network optimization: *AdaHessian* [**yao2021adahessian**] and *Apollo*, as they are optimizers that incorporate second-order information by estimating diagonal Hessian elements to adjust learning rates. This work will provide an in-depth analysis of both AdaHessian and Apollo,

evaluating their performance in comparison to other optimizers, with focus on the quality of Hessian approximation. Additionally, we aim to make the following contributions:

1. Examining whether the potential benefits of these optimizers, in terms of faster convergence and improved generalization, justify the increased computational costs.
2. Investigating whether the claimed advantage of these optimizers in providing a better approximation of the Hessian diagonal holds up when compared to simpler approximations made by first-order methods.
3. Exploring whether other optimization techniques with similar characteristics exist that offer a better tradeoff between performance and resource consumption.

1

THEORETICAL FOUNDATIONS

1.1 FOUNDATIONS OF DIFFERENTIAL CALCULUS

In this section, we explore the fundamental concept of the derivative. In mathematics, a derivative measures the rate at which a function changes as its input changes. Simply put, it represents the slope of the tangent line to the function's graph at any given point. For a function $f(x)$ of a single variable x, the derivative is often written as $f'(x)$ or $\frac{df}{dx}$, and it quantifies how much the function changes with a small change in x.

1.1.1 *Derivatives for Functions of a Single Variable*

Let $f : \mathbb{R} \to \mathbb{R}$ be a single-variable function. The derivative of the function f at a point x denoted as $f'(x)$, is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \quad h \in \mathbb{R}. \tag{1.1}$$

1.1.2 *Partial Derivatives*

Consider a function $f : \mathbb{R}^n \to \mathbb{R}$. The partial derivative of f with respect to the variable x_i at a point $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined as

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}, \quad h \in \mathbb{R}. \tag{1.2}$$

Assuming this limit exists, this definition encapsulates how f responds to infinitesimal changes in x_i, while keeping its other variables fixed.

1.1.3 *The Gradient*

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function that is differentiable at a point $\mathbf{x}$. Than the gradient of f at $\mathbf{x} \in \mathbb{R}^n$, that is denoted as $\nabla f(\mathbf{x})$, is the vector of all its first partial derivatives

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right) \quad \mathbf{x} \in \mathbb{R}^n. \tag{1.3}$$

The gradient vector points in the direction of the greatest rate of increase of the function, and its magnitude represents the rate of change in that direction [**bachman2007**].

1.1.4 *The Jacobian Matrix*

Let $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ be a vector-valued function. The Jacobian matrix $\mathbf{J}_\mathbf{f}$ of $\mathbf{f}$ is an $m \times n$ matrix that contains all first-order partial derivatives of the component functions f_i with respect to the input variables $x_j \in \mathbb{R}$, where $1 \leq i \leq m$, $1 \leq j \leq n$. It is defined as follows

$$\mathbf{J}_\mathbf{f} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \tag{1.4}$$

Each element $\frac{\partial f_i}{\partial x_j}$ of the Jacobian matrix represents the partial derivative of the i-th component function f_i with respect to the j-th input variable x_j. Later, we will see that the Jacobian matrix plays a crucial role in the backpropagation algorithm used for optimizing neural networks.

1.1.5 *The Hessian Matrix*

Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is function that is an at least twice differentiable and takes as input a vector $\mathbf{x} \in \mathbb{R}^n$ and outputs a scalar $f(\mathbf{x}) \in \mathbb{R}$. Then the Hessian matrix $\mathbf{H}$ of f is given by

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \tag{1.5}$$

Schwarz's Theorem states states the following: Let $U \subseteq \mathbb{R}^n$ be an open set and $f : U \to \mathbb{R}$ be at least k-times partially differentiable. If all k-th partial derivatives in U are at least continuous, then f is k-times totally differentiable. In particular, the order of differentiation in all l-th partial derivatives with $l \leq k$ is irrelevant. **[arens2011mathematik]** We can therefore conclude that $\mathbf{H}$ is in fact a symmetric real-valued $n \times n$ matrix, meaning $\mathbf{H} = \mathbf{H}^T$.

1.1.5.1 *Eigenvalues of the Hessian*

Because $\mathbf{H}$ is symmetric, there exist n lineraly independent eigenvectors, such that $\mathbf{H}$ can be factorized as

$$\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1} \tag{1.6}$$

where $\mathbf{Q}$ is an n × n matrix whose ith column is the eigenvector q_i of $\mathbf{H}$, and $\mathbf{\Lambda}$ is a diagonal matrix whose elements are the corresponding eigenvalues [**kashyap2023survey**].
The eigenvalues of the Hessian matrix play a crucial role in numerical optimization as they provide unique insight into the curvature of the function at a given point.
For example, a given point is a local minimizer if all $\lambda_i > 0$ or a local maximizer if all $\lambda_i < 0$. If there exist eigenvalues of different sign, than a given point presents a saddle point which are particularly challenging in the context of machine learning as optimizers as SGD often times get stuck at these regions [**bottou2018optimization**].

1.1.5.2 *Matrix definiteness*

Let $\mathbf{M}$ be an $n \times n$ symmetric real matrix with $n \in \mathbb{R}$, than the following statements hold:

$$
\begin{aligned}
M \text{ is positive-definite} &\iff \mathbf{x}^{\mathrm{T}} M\mathbf{x} > 0 \text{ for all } \mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\} \\
M \text{ is positive semi-definite} &\iff \mathbf{x}^{\mathrm{T}} M\mathbf{x} \geq 0 \text{ for all } \mathbf{x} \in \mathbb{R}^n \\
M \text{ is negative-definite} &\iff \mathbf{x}^{\mathrm{T}} M\mathbf{x} < 0 \text{ for all } \mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\} \\
M \text{ is negative semi-definite} &\iff \mathbf{x}^{\mathrm{T}} M\mathbf{x} \leq 0 \text{ for all } \mathbf{x} \in \mathbb{R}^n
\end{aligned}
$$

Additionally we define the positive semi-definite order of matrices by $A \preceq B \iff x^T(B - A)x \geq 0 \quad \forall x \in \mathbb{R}^n$, were $A, B \in \mathbb{R}^{n \times n}$ are symmetric matrices.

1.1.5.3 *Singular Matrix*

Let $\mathbf{M}$ be an $n \times n$ symmetric matrix with $n \in \mathbb{R}$, then $\mathbf{M}$ is called *singular* if and only if it's determinant is 0. Therefore $\mathbf{M}^{-1}$ does not exist, meaning $\mathbf{M}$ does not have an inverse.

1.1.5.4 *Ill-conditioned Matrices*

A matrix $\mathbf{M} \in \mathcal{R}^{n \times m}$ is called *ill-conditioned* when its *condition number* is high. This implies that small changes in its input (or elements) result in disproportionately large changes in its output. The condition number of a matrix M, is defined as

$$\kappa(M) = \frac{\sigma_{\max}(M)}{\sigma_{\min}(M)}, \tag{1.7}$$

where $\sigma_{\max}$ and $\sigma_{\min}$ are the largest and smallest singular values (square roots of non-negative eigenvalues [**SZABO2015320**]) of M respectively. M is called *ill-conditioned* if $\kappa(M) \gg 1$. [**strang2022introduction**].

1.2 INTRODUCTION TO OPTIMIZATION

Optimization is a crucial tool used in nearly all areas of decision science, engineering, economics, machine learning, and physical sciences [**nocedal2006numerical**]. The process of optimization begins by identifying an objective, which is a measurable indicator of the performance of a model or system. Common objectives in optimization often involve maximizing or minimizing quantities like profit, cost, or time. These objectives depend on system characteristics known as variables or parameters. The goal of optimization is to identify the values of the set of variables or parameters that minimize or maximize a given objective. The parameters often face some constraints, that are necessary to arrive at a practical solution. Examples for this might be the non-negativity of the interest rate on a loan or the electron density in a molecule.[**nocedal2006numerical**] The process of building such an objective-(function), choosing its parameters and constraints is called *modelling*. Constructing an effective model is often the most important step in solving a given problem. If the model is too simple, it might not capture the core of the problem. On the other hand, if it's overly complex, it may lead to difficult-to-compute solutions and become prone to issues like *overfitting*, which will be covered in later sections.

Mathematically, a model and its constraints may be defined as follows *[**nocedal2006numerical**]*

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) \quad \text{subject to} \quad c_i(\mathbf{x}) = 0,\ i \in \mathcal{E}, \quad c_i(\mathbf{x}) \geq 0,\ i \in \mathcal{I}.$$

In this example model we want to minimize the scalar-valued objective function f given a vector of variables or parameters $\mathbf{x}$, where $\mathbf{x}$ has to satisfy the scalar-valued constraint functions c_i, with sets of indices $\mathcal{E}$ and $\mathcal{I}$.

The set of parameters that serve as optimal solutions to a given model are often denoted as $\mathbf{x}^*$. Depending on whether the problem involves maximization or minimization, these parameters are also referred to as maximizers or minimizers, respectively. Once the model has been formulated, an optimization algorithm can be used to find its solution, usually with the help of a computer [**nocedal2006numerical**]. The choice of an optimization algorithm heavily relies on the model's characteristics. Factors like linearity, differentiability, convexity, and dimensionality influence whether methods like simplex (for linear models) or gradient descent (for differentiable non-linear models) are suitable.[**sun2019survey**].

In this work, we focus on *unconstrained nonlinear* optimization problems, which involve finding the optimal values of a nonlinear objective function without explicit constraints on the variables. These types of problems are the most frequently encountered in modern machine learning models, particularly in the realm of deep learning [**Goodfellow-et-al-2016**]. For these nonlinear optimization prob-

lems, we often differentiate between *first-order* and *second-order* optimization techniques. First-order methods, like (stochastic) gradient descent, use only the gradient information of the loss landscape to guide their search for a local *minimizer* of the objective function [**kashyap2023survey**], while second-order methods, such as Newton's method, utilize both the gradient and the Hessian matrix to better approximate the curvature, leading to more efficient optimization but with increased computational cost [**yao2021adahessian**]. In the following sections, we explore various optimization algorithms within first- and second-order techniques, highlighting their strengths and weaknesses. We also provide a foundational overview of neural networks, focusing on how these optimization strategies are applied to train them.

1.3 FIRST-ORDER OPTIMIZATION ALGORITHMS

First-order methods are optimization techniques that rely on gradient information to guide the search for a function's minimum or maximum. They are termed "first-order" because they only use first derivatives 1.1.3 to update parameters. They are computationally efficient and work well with large-scale problems, especially in machine learning [**Goodfellow-et-al-2016**].

1.3.1 *Gradient Descent*

Gradient descent (GD) is one of the most established first-order optimization algorithms. GD iteratively adjusts model parameters to minimize a given loss function by following the negative gradient direction [**toussain2014gradient**]. Since the gradient is a vector indicating the direction of steepest ascent of the loss function, adjusting the parameters in the direction of the negative gradient will result in a decrease in the function's value [**toussain2014gradient**].
Formely we write:
Let $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$ be a differentiable function, such that **arg min** $\mathcal{L} \neq \emptyset$. Let $\theta_0 \in \mathbb{R}^d$ and $\gamma > 0$ be a step size. *The Gradient Descent (GD)* algorithm defines a sequence $(\theta_t)_{t \in \mathbb{N}}$ satisfying:

$$\theta_{t+1} = \theta_t - \gamma \nabla \mathcal{L}(x_t). \tag{1.8}$$

The step size, denoted as γ, is often called the *learning rate* and is a crucial hyperparameter when training a model. An incorrect choice of γ can yield significantly different outcomes: if too large, the step size may overshoot the minimum, resulting in oscillation or divergence; if too small, progress will be slow, requiring many iterations to reach convergence. [**wu2020wngrad**]

1.3.2 *Empirical Risk Minimization (ERM)*

[bottou2018optimization] In machine learning, optimization problems naturally arise due to the formulation of prediction models and the associated loss functions. These are typically used to evaluate measures like the expected and *empirical risk*, which practitioners aim to minimize [**bottou2018optimization**]. To formalize this, let's define a family of model functions for some given $f(\cdot;\cdot) : \mathbb{R}^{d_x} \times \mathbb{R}^d \to \mathbb{R}^{d_y}$ as follows:

$$\mathbb{F} := \{f(\cdot;\theta) : \theta \in \mathbb{R}^d\}. \tag{1.9}$$

[bottou2018optimization] The set $\mathbb{F}$ represents a collection of functions parameterized by a parameter vector w that determine the prediction functions in use. Now assume a loss function $l : \mathbb{R}^{d_y} \times \mathbb{R}^{d_y} \to \mathbb{R}$, which, given an input-output pair (x, y), yields the loss $l(f(x;w), y)$. The most gratifying behavior for such a prediction function is to minimize the expected loss between any input-output pair. For that, let's assume that losses are measured with respect to a probability distribution $P(x, y)$ with $P : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \to [0, 1]$, then we could write the expected loss as

$$R(\theta) = \int_{\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}} l(f(x;\theta), y) dP(x, y) = \mathbb{E}[l(f(x;\theta), y)]. \tag{1.10}$$

Where $R(\theta)$ is also called the *expected risk* of a model f given the parameter vector θ. Minimization of $R(\theta)$ would ensure that the expected loss for the resulting model $f(\cdot;w)$ over all possible (x, y) is minimal.

[bottou2018optimization]

In practice, however, this is unfeasible when one lacks complete information about P. Instead, one seeks to solve a problem by estimating R. In supervised learning, a set of $n \in \mathbb{N}$ independently drawn input-output samples $\{(x_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ is typically available. In machine learning specifically, we refer to this set of data as the training data. From these samples, the *empirical risk* function $R_n : \mathbb{R}^d \to \mathbb{R}$ is defined as:

$$R_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} l(f(x_i;\theta), y_i) \tag{1.11}$$

where l is the loss function that measures the discrepancy between the prediction $f(x_i;\theta)$ and the actual output y_i. In standard gradient descent, it is actually this *empirical risk* function we strive to minimize, leading to the following gradient update step:

$$\theta_{t+1} = \theta_t - \gamma \nabla R_n(\theta_t). \tag{1.12}$$

The gradient descent algorithm continues to iterate through this update step until a convergence criterion is met, such as a maximum number of iterations or an acceptable error tolerance.

1.3.3 *Stochastic Gradient Descent*

[stanfordSGD]

Since standard gradient descent requires the evaluation of the gradient over the whole set of training data, large training sets can quickly become computationally expensive. Another issue with standard gradient descent optimization methods is that they don't give an easy way to incorporate new data in an 'online' setting. Stochastic Gradient Descent (SGD) addresses both of these issues by following the negative gradient of the objective after seeing only a single or a few training examples. Therefore approximating the true gradient of $R_n(\theta_t)$ by the gradient of a single or a few data points.

$$\nabla R_b(\theta) = \frac{1}{|B|} \sum_{i \in B} \nabla l(f(x_i; \theta), y_i), \tag{1.13}$$

with $B \in \mathbb{N}$ beeing the sample size or *batch size*. Leading to the following update rule [**stanfordSGD**]

$$\theta_{t+1} = \theta_t - \gamma \nabla R_b(\theta). \tag{1.14}$$

It can be shown that, this estimate $\nabla R_b(\theta)$ provides an unbiased estimator of the true gradient, satisfying [garrigos2024handbook]

$$\mathbb{E}[\nabla R_b(\theta)] = \nabla R_n(\theta). \tag{1.15}$$

While this introduces noise into the gradient estimates, repeated updates over many mini-batches allow SGD to approximate the true gradient descent path.

1.3.4 *Momentum*

While Stochastic Gradient Descent (SGD) is a very effective algorithm for optimization, its convergence rates can often be slow. To tackle the problem of slow convergence, the method of momentum [**polyak1964some**] was introduced. Momentum has the benefit of leading to faster convergence even in settings with high curvature or noisy gradients [**Goodfellow-et-al-2016**]. The momentum algorithm introduces an additional variable v, called the *velocity*, which defines the direction and speed at which the parameters move through parameter space [**Goodfellow-et-al-2016**]. The update rule for SGD with momentum is given by [Goodfellow-et-al-2016]

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_\theta J(\theta_t),$$

$$\theta_{t+1} = \theta_t - \eta v_t.$$

where

- v_t is the velocity vector at iteration t,

- $\beta \in [0, 1)$ is the momentum term, typically set between 0.9 and 0.99,
- $\nabla_\theta J(\theta_t)$ is the gradient of the loss function at iteration t.

[Goodfellow-et-al-2016] The hyperparameter β determines how quickly the contributions of previous gradients exponentially decay. Therefore, the larger β is, the more previous gradients affect the next update direction. With this exponentially weighted summation, we can avoid the gradient successively changing sign and jumping around, because the moving average smooths out the updates by considering the influence of past gradients. This results in a more stable and consistent direction of the gradient descent [**Goodfellow-et-al-2016**].

1.3.5 *RMSProp*

[Goodfellow-et-al-2016] RMSProp (*Root Mean Square Propagation*) [**hintonrms**] is an adaptive learning rate optimization algorithm that adjusts the learning rate for each parameter based on the magnitude of recent gradients. It works by maintaining an exponential moving average of the past squared gradients to obtain curvature information. The scalar learning rate then gets divided by this moving average and the resulting vector is mulplied with the current gradient. This results in a parameter wise scaling of the learning rate, increasing it in low and decreasing in high curvature regions. This helps to stabilize the training process and reduce oscillations [**Goodfellow-et-al-2016**]. The RMSProp update rules are given by:

[Goodfellow-et-al-2016]

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_\theta J(\theta_t)^2,$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla_\theta J(\theta_t),$$

where

- v_t is the exponentially weighted moving average of the squared gradients at time step t,
- $\beta \in [0, 1)$ is the decay rate, typically set to around 0.9,
- η is the learning rate,
- $\nabla_\theta J(\theta_t)$ is the gradient of the loss function at iteration t,
- ϵ is a small constant (e.g., 10^{-8}) to prevent division by zero.

1.3.6 *Adam*

[d2l2023adam]

Adam (*Adaptive Momentum*) [**kingma2017adam**] is one of the most well known and popular optimization algorithms for neural networks

today. It builds upon the ideas of Momentum and adaptive scaling of the learning rate, as it combines both the ideas of Momentum and RMSProp. Adam calculates an exponential moving average not only from the gradients, but also the squared gradients. It therefore introduces two hyperparameters $\beta_1 \in [0, 1)$ and $\beta_2 \in [0, 1)$ which determine the influence of past (squared) gradients into the moving average

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta J(\theta_t), \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \nabla_\theta J(\theta_t)^2, \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t}, \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t,
\end{aligned}
$$

where

- m_t is the velocity vector at iteration t,
- v_t is the exponentially weighted moving average of the squared gradients at time step t,
- $\beta_1 \in [0, 1)$ is the decay rate, used for the first order moment,
- $\beta_2 \in [0, 1)$ is the decay rate, used for the second order moment,
- $\hat{m}_t$ is the bias corrected first order moment at t,
- $\hat{v}_t$ is the bias corrected second order moment at t,
- η is the learning rate,
- $\nabla_\theta J(\theta_t)$ is the gradient of the loss function at iteration t,
- ϵ is a small constant (e.g., 10^{-8}) to prevent division by zero.

As we can see, the Adam algorithm is very similar to RMSProp, with the exception that it uses a first-order moment estimate (m_t) in addition to the second-order moment estimate (v_t). Unlike RMSProp, which directly uses the current gradient $\nabla_\theta J(\theta_t)$, Adam incorporates bias-corrected estimates to improve the stability of the training process. Adam initializes $m_0 = 0$ and $v_0 = 0$. Consequently, m_t and v_t are biased towards zero, especially when the decay rates β_1 and β_2 are close to 1 **[d2l2023adam]**. This bias can lead to very large step sizes in the early stages of training.**[Goodfellow-et-al-2016]** To counteract this, Adam applies bias correction by dividing m_t and v_t by $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$, respectively, ensuring that the gradients at earlier steps are accurately represented **[d2l2023adam]**.

1.3.7 *AdaBelief*

AdaBelief [**zhuang2020adabeliefoptimizeradaptingstepsizes**] is a novel optimizer that builds upon the Adam algorithm. The core concept of AdaBelief is to adapt the step size based on the "belief" in the current gradient direction. This "belief" is derived from the proximity of the current gradient estimate to the exponential moving average of past gradients, denoted as m_t. If the current gradient estimate is close to m_t, we have a higher confidence in the gradient estimate and take a larger step in the proposed direction. Conversely, if the current gradient estimate significantly deviates from m_t, we take a much smaller step. [**zhuang2020adabe**

This adaptive mechanism is achieved by using the squared difference between the gradient and the exponential moving average, rather than the gradient itself, in the calculation of the second moment estimate v_t [**zhuang2020adabelief**]

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t), \\
s_t &= \beta_2 s_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t) - m_t)^2 + \epsilon, \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\
\hat{s}_t &= \frac{s_t}{1-\beta_2^t}, \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{s}_t + \epsilon}}\hat{m}_t,
\end{aligned}
$$

where

- m_t is the velocity vector at iteration t,
- s_t is the exponentially weighted moving average of the squared difference of the gradient and m_t at time step t,
- $\beta_1 \in [0, 1)$ is the decay rate, used for the first order moment,
- $\beta_2 \in [0, 1)$ is the decay rate, used for the second order moment,
- $\hat{m}_t$ is the bias-corrected first order moment at t,
- $\hat{s}_t$ is the bias-corrected second order moment at t,
- η is the learning rate,
- $\nabla_\theta J(\theta_t)$ is the gradient of the loss function at iteration t,
- ϵ is a small constant (e.g., 10^{-8}) to prevent division by zero.

Following this approach, AdaBelief achieves fast convergence, training stability, and good generalization results, comparable to Stochastic Gradient Descent (SGD) [**zhuang2020adabelief**].

1.4 SECOND-ORDER OPTIMIZATION ALGORITHMS

This section examines optimization algorithms utilizing second-order information, particularly the Hessian matrix (Equation 1.5). We begin by examining the classical *Newton method*, which serves as a foundation for understanding the motivation for approximating the Hessian matrix in subsequent algorithms. Our discussion then progresses to established second-order optimization methods, including the popular Broyden-Fletcher-Goldfarb-Shanno (*BFGS*[**BFGS**]) and Davidon-Fletcher-Powell (*DFP*[**Goodfellow-et-al-2016**]) quasi-Newton algorithms. We then shift our analysis to recent advancements in the field like *AdaHessian*[**yao2021adahessian**] and *Apollo*[**apollo**], which are algorithms for Hessian diagonal approximation, that are particularly useful in the context of neural network optimization. Given their central role in this work, we provide a more comprehensive examination of *AdaHessian* and *Apollo* compared to the other algorithms discussed.

1.4.1 *The Newton method*

Newton's method forms the basis of second-order optimization algorithms that aims to find the local minimum or maximum of a differentiable function by iteratively improving an initial estimate. The basic idea is to use a Taylor series expansion to approximate the function by a paraboloid at a given point. The algorithm then identifies the minimum of this paraboloid, which provides a direction vector that can guide subsequent algorithms, such as line search, towards a local minimum or maximum.In the following we will express this mathematically.
Let f be the function we wish to optimize. Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable and that $\Delta x \in \mathbb{R}^n$ then we have that

$$f(x + \Delta x) = f(x) + \nabla f(x)^T \Delta x + \frac{1}{2} \Delta x^T \nabla^2 f(x) \Delta x \tag{1.16}$$

This follows from the second-order Taylor expansion with $x \in \mathbb{R}^n$ choosen as the expansion point. [**nocedal2006numerical**]
As previously mentioned, this function approximates a paraboloid. To find the minimizer of this approximation we differentiate with respect to Δx and set the gradient to zero.

$$\begin{aligned} \tfrac{d}{d\Delta x} f(x + \Delta x) \quad &= \quad \nabla f(x) + \nabla^2 f(x) \Delta x = 0 \\ &\iff \quad \Delta x = -(\nabla^2 f(x))^{-1} \nabla f(x) \end{aligned} \tag{1.17}$$

From this, we can conclude that Δx represents the vector offset from the current position. This leads us to the following algorithm, which implements the Newton update rule

$$x_k + \Delta x = x_k - (\nabla^2 f(x))^{-1} \nabla f(x), \quad k \in \mathbb{N}. \tag{1.18}$$

Here, x_k denotes the current position, while $x_k + \Delta x$ denotes the next position in the algorithm's path. Therefore, we set $x_k + \Delta x = x_{k+1}$ and $\mathbf{H} = (\nabla^2 f(x))^{-1}$, which yields the Newton update step [**nocedal2006numerical**]

$$x_{k+1} = x_k - \mathbf{H}_f(x_k)^{-1} \nabla f(x), \quad k \in \mathbb{N}. \tag{1.19}$$

This iterative process ensures that each step moves in the direction that minimizes the function f based on its local curvature and gradient.

In practice however the pure Newton method does not necessarily converge [**nocedal2006numerical**].
There are several factors that contribute to this behavior, which can be categorized as follows:

- **Non-Convex Function:** The Newton method relies on a second-order Taylor series expansion to approximate the function near the current point as a paraboloid. In highly non-covex settings however this approximation might not reflect the function's true behavior in the neighbourhood of the expansion point. This way finding the minimum of the paraboloid might actually lead to a point that *increases* the function value.[**kashyap2023survey**]

- **Singular Hessian Matrix :** In order for Newton's method to lead to a local minimum, the Hessian matrix $\nabla^2 f(x)$ must be *positive-definite* at each step. If the Hessian has mixed eigenvalues or is not positive-definite, the steps may lead away from a minimum. *BFGS* (Broyden–Fletcher–Goldfarb–Shanno) is a popular *quasi-newton* optimization algorithm that solves this problem of non positive-definite Hessians, by iteratively approximating the inverse Hessian with a rank-2 update formula, to preserve the non singular property of the Hessian (see **??**) .[**nocedal2006numerical**]

- **Sensitive to Initial Point:** The choice of initial point x_0 can greatly affect the convergence and accuracy of Newton's method. A good initial value should be close to the actual minimizer. Choosing a poor initial value can lead to divergent or inaccurate results, as the newton method converges and diverges quadratically.[**nocedal2006numerical**]

1.4.1.1 *Proof of Convergence*

*[**yao2021adahessian**]* We conclude this section on the Newton method by a proof of its convergence in a strongly convex and strictly smooth setting. This proof serves as a reference for our subsequent discussion, where we argue that using only the diagonal of the Hessian inverse also yields a convergent algorithm under strongly convex and strictly smooth conditions. Although we refer to [**yao2021adahessian**] for this proof,

it was originally described by [**boyd2004convex**].

Theorem (Quadratic Convergence of Newton's Method). *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a twice continuously differentiable, strongly convex and strictly smooth function. Then, the Newton update (see 1.19), yields a quadratically converging algorithm with the following guarantee:*

$$f(\theta_{t+1}) - f(\theta_t) \leq -\frac{\alpha}{2\beta^2}\|\nabla f(\theta_t)\|^2, \quad \theta \in \mathbb{R}^d$$

where $\nabla f(\theta_t)$ denotes the gradient at θ_t.

Proof
As f is twice continuously differentiable, strongly convex, and strictly smooth, we can state:

$$\exists \alpha, \beta > 0 : \alpha I \preceq \nabla^2 f(\theta) \preceq \beta I, \quad \forall \theta \in \mathbb{R}^d, \tag{1.20}$$

where I is the identity matrix, α is the strong convexity parameter (satisfying $\exists \alpha > 0 : \alpha I \preceq \nabla^2 f(\theta), \quad \forall \theta \in \mathbb{R}^d$) [**convexity_smoothness**], and β is the strict smoothness parameter (satisfying $\exists \beta > 0 : \nabla^2 f(\theta) \preceq \beta I, \quad \forall \theta \in \mathbb{R}^d$) [**convexity_smoothness**]. While $\preceq$ denotes the positive semidefinite ordering of matrices (see 1.1.5.2).
Now define $\lambda\theta_t = \left(g_t^T \mathbf{H}_t^{-1} g_t\right)^{1/2}$ and $\Delta\theta = \mathbf{H}^{-1} g_t$. Given the β-smoothness property of f, we can infer that

$$\begin{aligned} f(\theta_t - \eta\Delta\theta_t) &\leq f(\theta_t) + g_t^T((\theta_t - \eta\Delta\theta_t) - \theta_t) \\ &\quad + \frac{\beta}{2}\|(\theta_t - \eta\Delta\theta_t) - \theta_t\|^2 \\ &= f(\theta_t) - \eta g_t^T \Delta\theta_t + \frac{\eta^2\beta}{2}\|\Delta\theta_t\|^2. \end{aligned} \tag{1.21}$$

Now $(\lambda\theta_t)^2 = g_t^T \mathbf{H}_t^{-1} g_t = \Delta\theta_t^T \mathbf{H}_t \Delta\theta_t$, and $g_t^T \Delta\theta_t = (\lambda\theta_t)^2$.
Because of the strong convexity of f (see 1.20) we get

$$\Delta\theta_t^T(\mathbf{H}_t - \alpha I)\Delta\theta_t \geq 0 \iff \Delta\theta_t^T \mathbf{H}_t \Delta\theta_t \geq \alpha\|\Delta\theta_t\|^2, \tag{1.22}$$

thus $\|\Delta\theta_t\|^2 \leq \frac{1}{\alpha}\Delta\theta_t^T \mathbf{H}_t \Delta\theta_t = \frac{1}{\alpha}(\lambda\theta_t)^2$. Now putting everything together

$$f(\theta_t - \eta\Delta\theta_t) \leq f(\theta_t) - \eta(\lambda\theta_t)^2 + \frac{\eta^2\beta}{2\alpha}(\lambda\theta_t)^2. \tag{1.23}$$

Setting the stepsize $\eta = \frac{a}{\beta}$ and expanding, it follows

$$f(\theta_t - \eta\Delta\theta_t) \leq f(\theta_t) - \frac{1}{2}\eta(\lambda\theta_t)^2. \tag{1.24}$$

We follow 1.22 and since $\frac{1}{\beta} I \preceq (\nabla^2 f(\theta))^{-1}$, we get

$$(\lambda\theta_t)^2 = g_t^T \mathbf{H}_t^{-1} g_t \geq \frac{1}{\beta}\|g_t\|^2, \tag{1.25}$$

with which we finally arrive at the claim

$$f(\theta_t - \eta\Delta\theta_t) - f(\theta_t) \leq -\frac{1}{2\beta}\eta\|g_t\|^2 = -\frac{a}{2\beta^2}\|g_t\|^2. \qquad \square \tag{1.26}$$

1.4.2 *DFP & BFGS*

Now that we know why the Hessian is a very useful quantity for optimization, we will take a look at how we can approximate it, as exact calculation is infeasible for most large-scale problems. The Davidon-Fletcher-Powell (*DFP*[**Goodfellow-et-al-2016**]) and Broyden-Fletcher-Goldfarb-Shanno (*BFGS*[**BFGS**]) algorithms are so-called quasi-Newton algorithms that use a positive definite approximation of the Hessian. In the following, we cover the main ideas of *DFP* and *BFGS*. We start by introducing the quasi-Newton update formula,

$$x_{k+1} = x_k - \mathbf{B}_k^{-1} \nabla f(x_k), \quad k \in \mathbb{N}, \tag{1.27}$$

where $\mathbf{B}_k$ is the Hessian approximation at timestep k [**nocedal2006numerical**]. From this, we can derive the *secant equation*,

$$\mathbf{B}_k s_k = y_k, \tag{1.28}$$

where $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ and $s_k = x_{k+1} - x_k$. We cover the derivation of these in full detail in 1.4.4. By multiplying the above with s_k^T, we can conclude, that if $s_k^T y_k > 0$, known as the *curvature condition*, holds, there exists a $\mathbf{B}_k$ with positive curvature along s_k, meaning $s_k^T \mathbf{B}_k s_k > 0$.

$$\mathbf{B}_{k+1} = \min_B \|B - \mathbf{B}_k\| \qquad \text{s.t. } B = B^T,\ Bs_k = y_k \tag{1.29}$$

When using the Frobenius norm for this optimization problem, we get a unique solution for $\mathbf{B}_{k+1}$ with,

$$\mathbf{B}_{k+1} = \left(I - \rho_k y_k s_k^T\right) \mathbf{B}_k \left(I - \rho_k s_k y_k^T\right) + \rho_k y_k y_k^T, \tag{1.30}$$

where $\rho_k = \frac{1}{y_k^T s_k}$. This update formula is usually referred to as the *DFP* updating formula [**DFP**]. As we need the inverse $\mathbf{C}_{k+1} := \mathbf{B}_{k+1}^{-1}$ for performing the Newton step (see 1.19), one can employ the *Sherman–Morrison–Woodbury* formula, defined as

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1} u v^T A^{-1}}{1 + v^T A^{-1} u} \, A \in \mathbb{R}^{n \times n},\ u, v \in \mathbb{R}^n, \tag{1.31}$$

to show by expanding and subsequently rearranging 1.30, that

$$\mathbf{C}_{k+1} = \mathbf{C}_k - \frac{\mathbf{C}_k y_k y_k^T \mathbf{C}k}{y_k^T \mathbf{C}k y_k} + \frac{s_k s_k^T}{y_k^T s_k}, \tag{1.32}$$

which is the update equation that is used in the *DFP* algorithm. The *BFGS* algorithm works very similarly, with the subtle difference that it imposes the above conditions on the inverses of the Hessian approximations, meaning we have

$$\mathbf{C}k + 1 = \min_C \|C - \mathbf{C}k\| \qquad \text{s.t. } C = C^T, \quad Cy_k = s_k \tag{1.33}$$

Again, *BFGS* uses the Frobenius norm, which leads to the following update formulation:

$$\mathbf{C}_{k+1} = (I - \rho_k s_k y_k^T)\mathbf{C}_k(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T, \tag{1.34}$$

with $\rho_k = \frac{1}{y_k^T s_k}$. As $\mathbf{C}_{k+1}$ is already an approximation of the Hessian inverse at x_k, we can directly use it for step calculation. Regarding the initial choice of $\mathbf{C}_0$, one often selects the identity matrix or approximates the Hessian inverse using finite differences on the gradient, when computationally feasible. As there is no universally effective initialization method for $\mathbf{C}_0$ across all optimization problems.

1.4.3 *AdaHessian*

AdaHessian, introduced in 2020 by Yao et al. [**yao2021adahessian**], is an adaptive second-order optimization algorithm. While being conceptually very similar to *Adam* (1.3.6), *AdaHessian* replaces the square of the gradients in *Adam's* second moment with the square of a Hessian diagonal approximation. To estimate the Hessian diagonal, AdaHessian employs two key techniques. First, it utilizes a Hessian-free method based on the Hessian-vector product [**pearlmutter1994fast**]. This approach allows for efficient computation without explicitly forming the full Hessian matrix. Second, AdaHessian implements a stochastic Hessian diagonal approximation based on [**martens20150optimizing**], which leverages the Hutchinson method [**hutchinson**], a technique for trace estimation of matrices. Now let f with $f : \mathbb{R}^n \to \mathbb{R}$ and $x \in \mathbb{R}^n$, be a neural network with subsequent loss calculation. We start by taking the scalar product of $g = \nabla_\theta f$ with z, where $z \in \mathbb{R}^n$ is a random vector which follows a Rademacher distribution. This results in a scalar. We then calculate the derivative of this scalar with respect to θ, such that we get

$$\frac{\partial g^T z}{\partial \theta} = \frac{\partial g^T}{\partial \theta} z + g^T \frac{\partial z}{\partial \theta} = \frac{\partial g^T}{\partial \theta} z = Hz. \tag{1.35}$$

This method is known as a Hessian-free approach,because by calculating this derivative, we obtain a Hessian-vector product without explicitly forming the Hessian matrix. Following the results from [**martens20150optimizing**], we get

$$D = \operatorname{diag}(H) = \mathbb{E}[z \odot (Hz)]. \tag{1.36}$$

The Hessian diagonal estimation in AdaHessian leverages an unbiased stochastic approximation technique. Specifically, the expression $z \odot (Hz)$, where $\odot$ denotes the Hadamard (element-wise) product, serves as an unbiased estimator for the diagonal elements of the Hessian matrix. Note that the authors of [**yao2021adahessian**] found that a single sampling of z is usually sufficient to lead to a reasonable

diagonal approximation. Next up, we demonstrate that employing this diagonal approximation of the Hessian in the update step yields convergence properties equivalent to those achieved when utilizing the full Hessian matrix.

Theorem (Convergence Rate of Hessian Diagonal Method)
[yao2021adahessian] *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a twice continuously differentiable, strongly convex and strictly smooth function. Then, the update rule given by*

$$\theta_{t+1} = \theta_t - \eta D_t^{-1} g_t,$$

where D_t is the diagonal of the Hessian $H_t = \nabla^2 f(\theta_t)$ and $g_t = \nabla f(\theta_t)$, yields a converging algorithm with the following guarantee

$$f(\theta_{t+1}) - f(\theta_t) \leq -\frac{\alpha}{2\beta^2}\|g_t\|^2, \quad \theta \in \mathbb{R}^d.$$

Proof
As f is strongly convex and strictly smooth function, we know from 1.4.1.1, that $\exists \alpha, \beta > 0 : \alpha I \preceq \nabla^2 f(\theta) \preceq \beta I, \quad \forall \theta \in \mathbb{R}^d$. To demonstrate that these bounds also apply to the diagonal matrix D, let's consider the standard basis vectors. For any e_i, where all elements are 0 except for the i-th one, which is 1, we can observe:

$$\alpha \leq e_i^T H e_i = e_i^T D e_i = D_{i,i} \quad \text{and} \quad \beta \geq e_i^T H e_i = e_i^T D e_i = D_{i,i} \tag{1.37}$$

This relationship implies that $D_{i,i} \in [\alpha, \beta], \quad \forall i \in \{1, \ldots, d\}$. Consequently, we can extend the matrix inequality to D, such that $\exists \alpha, \beta > 0 : \alpha I \preceq D \preceq \beta I, \quad \forall \theta \in \mathbb{R}^d$. Given this result, we can apply the same convergence analysis as in 1.4.1.1, thus proving the claim. □

To mitigate the inherent stochastic variance associated with this approximation, AdaHessian employs two key strategies. First, it maintains an Exponential Moving Average (EMA) of the diagonal estimates D. Second, AdaHessian implements a spatial averaging algorithm. Consider a Convolutional Neural Network (CNN) as an example. In a CNN, for a convolutional kernel with block size b (for instance, $b = 9$ for a 3×3 kernel), we perform spatial averaging among the kernel's parameters. This can be mathematically expressed as:

[yao2021adahessian]

$$D^{(s)}[ib + j] = \frac{1}{b}\sum_{k=1}^{b} D[ib + k], \quad \text{for } 1 \leq j \leq b,\ 0 \leq i \leq \left\lfloor \frac{d}{b} \right\rfloor - 1, \tag{1.38}$$

where d is the number of model parameters.After applying the spatial averaging, we can define the second momentum of AdaHessian with

$$\bar{D}_t = \beta_2 \bar{D}_{t-1} + (1 - \beta_2)(D^{(s)})^2. \tag{1.39}$$

Here, $\bar{D}_t$ represents the smoothed estimate of the squared Hessian diagonal at time step t, β_2 is the exponential decay rate for the second

moment estimate, and $D_{(s)}$ is the spatially averaged Hessian diagonal estimate. As mentioned earlier, the rest of AdaHessian functions exactly analogous to Adam (see 1.3.6), leading to the algorithm described in 1. [**yao2021adahessian**]

Algorithm 1 AdaHessian

Require: Initial parameter θ_0
Require: Learning rate η
Require: Exponential decay rates β_1, β_2
Require: Block size b
Require: Hessian power k
Initialize $m_0 = 0, v_0 = 0$
for $t = 1, 2, \ldots$ **do**
  $g_t \leftarrow$ current step gradient
  $D_t \leftarrow$ current step estimated diagonal Hessian
  Compute $D_t^{(s)}$ based on 1.38
  Update $\bar{D}_t$ based on 1.39
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(D_t^{(s)})^2$
  $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
  $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
  $\theta_t = \theta_{t-1} - \eta \frac{m_t}{v_t}$
end for

1.4.4 *Apollo*

Apollo [**apollo**] is a rather newly proposed quasi-Newton algorithm for non-convex stochastic optimization. It operates by calculating a non-singular diagonal approximation of the Hessian matrix, utilizing the weak secant equation. To elucidate the algorithm's mechanics, we first briefly revisit the theoretical foundations of the secant equation in general, and subsequently derive the algorithm from this basis. We recall from 1.19 that for a supposed neural network with subsequent loss calculation f with $f : \mathbb{R}^n \to \mathbb{R}$ and $x \in \mathbb{R}^n$, we consider the Newton update step as follows:

$$x_{k+t} = x_t - \mathbf{H}_f(x_t)^{-1} \nabla f(x_t), \quad k \in \mathbb{N}. \tag{1.40}$$

For simplicity, we write $\mathbf{H}_t$ instead of $\mathbf{H}_f(x_t)^{-1}$. With this, we can derive the general *quasi-Newton* update formula:

$$x_{k+1} = x_k - \mathbf{B}^{-1} \nabla f(x_t), \quad t \in \mathbb{N}, \tag{1.41}$$

where $\mathbf{B}$ is an approximation of the Hessian matrix at x_t. We can rewrite this as

$$\begin{aligned} x_{t+1} &= x_t - \mathbf{B}^{-1}\nabla f(x_t), \quad k \in \mathbb{N} \\ &\iff x_{t+1} - x_t = -\mathbf{B}_t^{-1}\nabla f(x_t) \\ &\iff \mathbf{B}(x_{t+1} - x_t) = -\nabla f(x_t) \\ &\iff \mathbf{B}(x_{t+1} - x_t) + \nabla f(x_t) = 0 \end{aligned} \tag{1.42}$$

From 1.4.1 we know that$\nabla_{\Delta x} f(x + \Delta x)$ should satisfy $\nabla_{\Delta x} f(x + \Delta x) = 0$. We can therefore conclude that $\mathbf{B}$ has to satisfy

$$\nabla_{\Delta x} f(x + \Delta x) = \mathbf{B}(x_{t+1} - x_t) + \nabla f(x_t). \tag{1.43}$$

This is equivalent to computing a second-order Taylor expansion (see 1.4.1), then taking the gradient with respect to Δx. We then proceed by defining

$$y_t = \nabla_{\Delta x} f(x + \Delta x) - \nabla f(x_t) \tag{1.44}$$

$$s_t = x_{t+1} - x_t, \tag{1.45}$$

such that the before formula is now

$$\mathbf{B}s_t = y_t, \tag{1.46}$$

which is also known as the *strong secant equation*. For the next approximation, we choose the closest matrix to $\mathbf{B}$ under the condition of the *strong secant equation*. Therefore, our algorithm for updating the Hessian approximation $\mathbf{B}$ will now be

$$\mathbf{B}_{t+1} = argmin_{\mathbf{B}} \|\mathbf{B} - \mathbf{B}_t\|, \qquad \text{s.t. } \mathbf{B}_{t+1}\mathbf{s}_t = \mathbf{y}_t. \tag{1.47}$$

This optimization problem forms the foundation for a family of quasi-Newton algorithms such as BFGS[**BFGS**], DFP[**DFP**], or SR1[**SR1**].[**apollo**]. *Apollo* employs a weakened form of the *secant equation*, known as the *weak secant equation*. The rationale behind this choice is as follows: While we have used a scalar-valued function f in our example, the difference of the gradients y_t is a vector-valued function. For such functions, the *mean value theorem*—which forms the basis of the standard secant equation in 1.4.4—generally does not hold[**apollo**]. Therefore, we apply the scalar product with s_t^T to weaken the condition. This relaxes the equality requirement to hold only in the direction of s_t.

$$\mathbf{B}_{t+1} = argmin_{\mathbf{B}} \|\mathbf{B} - \mathbf{B}_t\|, \qquad \text{s.t. } \mathbf{s}_t^T\mathbf{B}_{t+1}\mathbf{s}_t = \mathbf{s}_t^T\mathbf{y}_t. \tag{1.48}$$

This optimization problem can be solved by an approach first proposed in [**Zhu1999TheQR**], where the norm in 1.48 is interpreted as the Frobenius norm.

$$\Lambda = \mathbf{B}_{t+1} - \mathbf{B}_t = \frac{s_t^T y_t - s_t^T \mathbf{B}_t s_t}{\|s_t\|_4^4} \text{Diag}(s_t^2) \tag{1.49}$$

here s_t^2 is the element-wise square vector of s_t, and $\text{Diag}(s_t^2)$ is the diagonal matrix with diagonal elements from vector s_t^2, and $\|\cdot\|_4$ is the 4-norm of a vector [**apollo**]. To ensure that the Hessian update remains invariant to the chosen stepsize, the step direction s_t is normalized by η_t, leading to

$$\Lambda' = -\frac{d_t^T y_t + d_t^T \mathbf{B}_t d_t}{\|d_t\|_4^4} \text{Diag}(d_t^2), \tag{1.50}$$

where $d_t = -\frac{s_t}{\eta_t}$. The whole algorithm for *Apollo* is displayed in 1.4.4. Instead of working with gradients g_t directly, we choose the exponential moving average of them, in the same fashion as we do in the *Adam* optimizer 1.3.6. Because *Apollo* uses the newton-step 1.19 for its parameter update, we have to make sure that the approximation $\mathbf{B}$ is non singular. For that we define another diagonal matrix $\mathbf{D_t}$ for which we choose

$$\mathbf{D_t} = \text{rectify}(|\mathbf{B_t}|, \sigma) = \max(|\mathbf{B_t}|, \sigma), \quad |\mathbf{B_t}| = \sqrt{\mathbf{B_t}^T \mathbf{B_t}}, \tag{1.51}$$

with $\sigma \geq 0$. This approach achieves two key objectives: First, it ensures that small steps are taken in regions of very high curvature (sharp edges), as $|\mathbf{B_t}|$ becomes large in those cases, and that larger steps are taken when $|\mathbf{B_t}|$ is low, although not excessively large, since saddle points are common in the loss landscape. Secondly, it guarantees that $\mathbf{D_t}$ has no zero-valued diagonal elements, ensuring that it remains non-singular. In practice *Apollo* chooses $\sigma = 0.01$, although this can be tuned as a hyperparameter.**[apollo]** *[**apollo**]*

Algorithm 2 Apollo

Initial: $m_0, d_0, B_0 \leftarrow 0, 0, 0$ ▷ Initialize m_0, d_0, B_0 to zero
Good default settings are $\beta = 0.9$ and $\epsilon = 10^{-4}$
while $t \in \{0, \ldots, T\}$ **do**
  for $\theta \in \{\theta_1, \ldots, \theta_L\}$ **do**
    $g_{t+1} \leftarrow \nabla f_t(\theta_t)$ ▷ Calculate gradient at step t
    $m_{t+1} \leftarrow \frac{\beta(1-\beta^t)}{1-\beta^{t+1}} m_t + \frac{1-\beta}{1-\beta^{t+1}} g_{t+1}$ ▷ Bias-corrected EMA
    $\alpha \leftarrow \frac{d_t^T (m_{t+1} - m_t) + d_t^T B_t d_t}{(\|d_t\|_4 + \epsilon)^4}$ ▷ Calculate coefficient of B update
    $B_{t+1} \leftarrow B_t - \alpha \cdot \text{Diag}(d_t^2)$ ▷ Update diagonal Hessian
    $D_{t+1} \leftarrow \text{rectify}(B_{t+1}, 0.01)$ ▷ Handle nonconvexity
    $d_{t+1} \leftarrow D_{t+1}^{-1} m_{t+1}$ ▷ Calculate update direction
    $\theta_{t+1} \leftarrow \theta_t - \eta_{t+1} d_{t+1}$ ▷ Update parameters
  end for
end while
return θ_T