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

October 2024

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor, Moritz Wolter, for his consistent guidance and support throughout this project. He was always available to answer my questions and provided valuable feedback whenever I needed it. I would also like to express my gratitude to my friend Christoph Wolff, who always kept me updated with the latest AI news he discovered on Twitter.

LIST OF FIGURES

LIST OF TABLES

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[1].
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[1].
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* [43] 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. If the claim in 2 does not hold, investigate the reasons behind the optimizer's failure and propose a solution.

We begin by providing a general overview of the theoretical foundations of optimization and neural network training in Chapter 1. This includes a detailed explanation of well-known first-order optimizers, as well as Newton's Method, and the motivation behind *Apollo* and *AdaHessian*. The chapter concludes with an overview of the neural network training process, along with manual methods for calculating the Hessian matrix using forward and backward propagation. In Chapter 2, we evaluate each optimizer on multiple datasets, including tasks in both vision and translation. Additionally, we measure the resource footprint of each optimizer to assess their real-world applicability. In Chapter 3, we assess the quality of the Hessian approximation for both second-order and selected first-order optimizers using a small convolutional neural network. We conclude our work by introducing the *SApollo* optimizer, which aims to address issues related to the implementation of *Apollo*.

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 [3].

1.1.4 *The Jacobian Matrix*

Let $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ be a vector-valued function. The Jacobian matrix $\mathbf{J_f}$ 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

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

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 $f \colon U \to \mathbb{R}$ 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[2].
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 [21].
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 [7].

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:

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

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 [35]) of M respectively. M is called *ill-conditioned* if $\kappa(M) \gg 1$ [33].

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 [28]. 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[28]. 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.

As defined in [28]

Mathematically, a model and its constraints may be defined as follows

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

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 [28]. 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[34].
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 [17]. For these nonlinear optimization problems, 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 [21], 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 [43]. 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 [17].

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 [38]. 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 [38].
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 hyper-parameter 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. [42]

1.3.2 *Empirical Risk Minimization (ERM) [7]*

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 [7]. 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:

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

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) and a model $f \in \mathbb{F}$, 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 risk* as

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

Minimization of $R(\theta)$ would ensure that the expected loss for the resulting model $f(\cdot; w)$ over all possible (x, y) is minimal.

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 [39]*

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 $B \subseteq \{(x_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$.

$$\nabla R_b(\theta) = \frac{1}{|B|} \sum_{\{\hat{x}, \hat{y}\} \in B} \nabla l(f(\hat{x}; \theta), \hat{y}) \tag{1.13}$$

with $B \in \mathbb{N}$ beeing the sample size or *batch size*. Leading to the following update rule [39]

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

It can be shown that, this estimate $\nabla R_b(\theta)$ provides an unbiased estimator of the true gradient [15], satisfying

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

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 [17]*

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 [30] was introduced. Momentum has the benefit of leading to faster convergence even in settings with high curvature or noisy gradients [17]. 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 [17]. The update rule for SGD with momentum is given by

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

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

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.

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 [17].

1.3.5 *RMSProp [17]*

RMSProp (*Root Mean Square Propagation*) [16] 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 [17]. The RMSProp update rules are given by: *[17]*

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

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

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 [22]*

Adam (*Adaptive Momentum*) 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 [12]. This bias can lead to very large step sizes in the early stages of training [17]. 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 [12].

1.3.7 *AdaBelief [45]*

AdaBelief [45] 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.

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 [46]

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

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) [46].

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*[10]) and Davidon-Fletcher-Powell (*DFP*[17]) quasi-Newton algorithms. We then shift our analysis to recent advancements in the field like *AdaHessian*[43] and *Apollo*[25], 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 [28].
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.

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

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 [28]

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

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 [28]. 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.[21]
- **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 1.4.2) .[28]
- **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.[28]

1.4.1.1 *Proof of Convergence [43]*

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 [43] for this proof, it was originally described by [8].

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 $\exists \alpha > 0 : \alpha I \preceq \nabla^2 f(\theta), \quad \forall \theta \in \mathbb{R}^d$) [41], and β is the strict smoothness parameter (satisfying $\exists \beta > 0 : \nabla^2 f(\theta) \preceq \beta I, \quad \forall \theta \in \mathbb{R}^d$) [41]. While $\preceq$ denotes the positive semidefinite ordering of matrices (see 1.1.5.2).
Now define a function $\lambda(\theta_t) = \left(g_t^T \mathbf{H}_t^{-1} g_t\right)^{1/2}$ and $\Delta\theta = \mathbf{H}^{-1} g_t$. Given the β-smoothness property of f, we can infer that

$$\begin{aligned} f(\theta_t - \eta\Delta\theta_t) &\leq f(\theta_t) + g_t^T((\theta_t - \eta\Delta\theta_t) - \theta_t) \\ &\quad + \frac{\beta}{2}\|(\theta_t - \eta\Delta\theta_t) - \theta_t\|_2^2 \\ &= f(\theta_t) - \eta g_t^T \Delta\theta_t + \frac{\eta^2\beta}{2}\|\Delta\theta_t\|_2^2. \end{aligned} \tag{1.21}$$

Now $\lambda(\theta_t)^2 = g_t^T \mathbf{H}_t^{-1} g_t = \Delta\theta_t^T \mathbf{H}_t \Delta\theta_t$, as $\mathbf{H}_t$ is symmetric, 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 $\frac{1}{\beta} I \preceq \mathbf{H}_t^{-1}$, 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

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

1.4.2 *DFP & BFGS [28]*

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*[17]) and Broyden-Fletcher-Goldfarb-Shanno (*BFGS*[10]) 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 [28]. 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 $s_k^T \mathbf{B}_k s_k > 0$ (see 1.1.5.2).

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

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 [14]. 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

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

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 [43]*

AdaHessian, 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 [29]. This approach allows for efficient computation without explicitly forming the full Hessian matrix. Second, AdaHessian implements a stochastic Hessian diagonal approximation based on [4], which leverages the Hutchinson method [19], 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 [4], 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 [43] 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) [43]
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,

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

[43] 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 [43].

1.4.4 *Apollo [25]*

Apollo [25] 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:

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

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

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

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

This optimization problem forms the foundation for a family of quasi-Newton algorithms such as BFGS[10], DFP[14], or SR1[9].[25]. *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[25]. 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.

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

This optimization problem can be solved by an approach first proposed in [44], 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 [25]. 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

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

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 $\sigma = 0.01$.[25] *[25]*

1.5 INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

Biological neural networks are intricate systems of interconnected neurons that communicate with each other to process information. Each neuron, a specialized nerve cell, is designed to receive, process, and transmit signals through electrochemical processes. Around the year 1900, a groundbreaking realization emerged that

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
    $B_{t+1} \leftarrow B_t - \alpha \cdot \mathrm{Diag}(d_t^2)$ ▷ Update diagonal Hessian
    $D_{t+1} \leftarrow \mathrm{rectify}(B_{t+1}, 0.01)$ ▷ 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

these tiny physical building blocks of the brain — the nerve cells and their intricate connections — are responsible for perception, associations, thoughts, consciousness, and the ability to learn.[13] The idea of creating an artificial version of the brain to replicate its functions and achieve a synthetic form of intelligence has a long history.

The significant leap towards neural network-based artificial intelligence was made in 1943 by McCulloch and Pitts in their article "A Logical Calculus of the Ideas Immanent in Nervous Activity" [26]. They were the first to present a mathematical model of the neuron as a fundamental computational unit of the brain. This article laid the foundation for constructing artificial neural networks and, consequently, for this crucial subfield of AI.[13] With the advent of efficient optimization methods and improved computational power, researchers were able to develop and train neural networks that could genuinely learn and significantly enhance their performance. These advances in optimization algorithms, such as (stochastic) gradient descent, coupled with accelerating hardware like GPUs, enabled neural networks to process vast amounts of data. This breakthrough resulted in models that could recognize patterns, make predictions, and solve complex problems across various domains, effectively demonstrating their learning capabilities. [13] In this section, we discuss the architecture and some of the underlying theory behind neural networks, including the algorithms used for training. We will explore how neural networks are structured, the roles of different layers, and how information flows through them. Additionally, we will delve into the principles and methods behind training algorithms like backpropagation, which adjust the network's weights to improve its predictive accuracy.

1.5.1 *The artificial Neuron [13]*

A neuron in an artificial neural network is modeled as a mathematical function that processes input signals and produces an output. The basic structure is as follows:

$$y_j = f\left(\sum_{i=1}^{n} \theta_i x_i + b\right).$$

where:

- y_j is the output of the neuron,
- f is a nonlinear activation function applied to the weighted sum of the inputs,
- $\sum_{i=1}^{n}$ is the summation over all input values,
- θ_i are the weights associated with each input,
- x_i are the input values, and
- b is the bias term.

The inputs x_i of a neuron are each weighted with an individual weighting factor θ_i and summed up. These factors θ_i together with the *bias term* b, represent the trainable parameters of a neuron and are responsible for its performance on a given task. The sum is then fed into a non linear activation function f. This function is modeled on the idea that the activity of real biological neurons depends on a certain activation threshold[13]. Activation functions that are commonly used in practice are the **sigmoid-function** $f(z) = \frac{1}{1+e^{-z}}$ or **ReLU** (Rectified Linear Unit) $f(z) = \max(0, z)$[27]. The *bias term* b helps adjust the output by shifting the activation function horizontally. This allows the neuron to represent patterns that are not centered around the origin, making it able to better approximate a hyperplane[13].

1.5.2 *The Multi-Layer Perceptron (MLP)[17]*

A single neuron is limited in its ability to distinguish between linearly separable data points, as its mathematical formulation models a hyperplane. Consequently, it cannot represent the output of an XOR (exclusive OR) gate. This is because the XOR function is not linearly separable.[17]
To overcome this limitation, the Multilayer Perceptron (MLP) was introduced. An MLP consists of multiple stacked layers of single perceptrons (neurons with an activation function). Each layer takes an input, processes it through its perceptrons, and outputs a vector that can then be fed into the next perceptron layer. This structure enables the MLP to handle complex, non-linearly separable data, such as the XOR function [17]. The output of an MLP layer can be mathematically expressed as:

$$x^{(l)} = f\left(\theta^{(l)} x^{(l-1)} + b^{(l)}\right),$$

where:

- l is the current layer,
- $x^{(l)}$ is the output of layer l,
- f is the activation function applied to the layer's output,
- $\theta^{(l)}$ is the weight matrix of layer l,
- $x^{(l-1)}$ is the input vector to layer l (output of the previous layer),
- $b^{(l)}$ is the bias vector of layer l.

As we can see, the formulation follows the same principle as a single neuron but now in a vectorized form. $\theta^{(l)}$ is a weight matrix of size $n \times m$, where m is the number of inputs to a single neuron of the next layer and n is the number of neurons in a single layer. $\mathbf{x}^{(l-1)}$ is the output vector from the previous layer. From their matrix-vector product, we get the pre-activations for each layer in vectorized form. This vector is then fed into a non-linear activation function, whose output is then fed into the next layer.

It is easy to see why the activation function must be non-linear. If the activation functions were linear, the stacked MLP layers would collapse into a single layer, rendering the model unable to fit complex, highly non-linear data.[17]

The MLP falls into the category of feedforward neural networks, meaning that the flow of information is unidirectional, from one layer to the next. Each layer passes its information forward to the subsequent layer. In contrast, there are other network architectures, such as recurrent neural networks (RNNs), where the flow of information is not restricted to only consecutive layers. In RNNs, information can loop back to the same layer, allowing the network to maintain and utilize internal state information over time. In practice, a Multilayer Perceptron (MLP) typically consists of an input layer that takes the data and passes it to one or more hidden layers before reaching the output layer. In the output layer, the data is usually mapped to a probability distribution using a softmax function:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, m, \tag{1.52}$$

which can then be used to get the model's prediction depending on the task that should be performed.

1.5.3 *Training of Neural Networks [17]*

The goal of training is to find a set of parameters such that the neural network minimizes a particular cost function. The cost or loss function is a measure that determines the amount of error in the predictions $\hat{y}$ made by the neural network on a given dataset. In supervised training, we have a set of data points $x \in \mathbf{X}$ and the corresponding *ground truth* labels $y \in \mathbf{Y}$. We formulate our cost function such that it calculates the deviation between a ground truth label y and the prediction $\hat{y}$

of the network given the corresponding data point. One of the most popular loss functions is the Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2. \tag{1.53}$$

The MSE measures the Euclidean distance between the predictions $\hat{y}$ and the ground truth y. It is straightforward to show that minimizing the MSE is equivalent to maximizing the log likelihood:

$$\theta_{\text{ML}} = \arg\max_{\theta}\sum_{i=1}^{m}\log P(y^{(i)}|x^{(i)};\theta). \tag{1.54}$$

where θ_{ML} is the set of parameters that maximizes the probability that the labels $y^{(i)}$ correspond to the data points $x^{(i)}$.

Another common loss function, often used for classification tasks, is the Cross-Entropy Loss:

$$H(y,\hat{y}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c}). \tag{1.55}$$

where m and C are the number of samples and classes respectively. To understand the intuition behind this definition, consider a target or true probability distribution P and an approximate distribution Q. The cross-entropy between Q and P quantifies the additional number of bits needed to encode events from P using the distribution Q instead of the true distribution P. By minimizing the cross-entropy between Q and P, we improve the approximation of the true distribution P using the model distribution Q.[6]

To train the model, i.e., to tune its parameters such that the loss of the model's output is minimized, we use gradient-based optimization methods as described in 1.19. To calculate the gradient of the parameters, we employ the *backpropagation algorithm*. The backpropagation algorithm was first introduced by Rumelhart, Hinton, and Williams in 1986 [32]. Unless otherwise noted, all information in this section regarding the technique is taken from their work. It utilizes the chain rule to efficiently backpropagate gradient information from the last layer to the first.

Before applying the backpropagation algorithm, we first have to evaluate all the layers and activations in the network. This process is often called the forward pass. After computing the loss L, we calculate the gradient of L with respect to the pre-activations of the model output, which we denote as $\delta^M = \frac{\partial L}{\partial z^{(M)}} = \frac{\partial L}{\partial \hat{y}^{(M)}} \cdot \frac{\partial \hat{y}^{(M)}}{\partial z^{(M)}}$, where $z^{(M)} = \theta^{(M)}x^{(M-1)} + b^{(M)}$ and $\hat{y} = f(z^{(M)})$, with M being the last layer. We then loop back to the first layer with $0 \leq m < M$ and calculate the respective

gradients of the current layer using the gradient information from the previous layer. For that, we set

$$\delta^{(m)} = \frac{\partial L}{\partial z^{(m)}} \tag{1.56}$$

$$= \left[\frac{\partial \hat{y}^{(m+1)}}{\partial z^{(m)}} \frac{\partial L}{\partial z^{(m+1)}} \right] \odot \frac{\partial \hat{y}^{(m)}}{\partial z^{(m)}} \tag{1.57}$$

$$= \left[\frac{\partial \hat{y}^{(m+1)}}{\partial z^{(m)}} \delta^{(m+1)} \right] \odot \frac{\partial \hat{y}^{(m)}}{\partial z^{(m)}} \tag{1.58}$$

$$= \left[\theta^{(m+1)^T} \delta^{(m+1)} \right] \odot \nabla_{z^{(m)}} f. \tag{1.59}$$

Here, $\odot$ denotes the element-wise (Hadamard) product. To calculate the gradient information for the i-th the parameter of a neuron j in layer m, we set

$$\frac{\partial L}{\partial \theta_{j,i}^{(m)}} = \frac{\partial L}{\partial z_j^{(m)}} \cdot \frac{\partial z_j^{(m)}}{\partial \theta_{j,i}^{(m)}} = \delta_j^{(m)} \cdot \hat{y}_i^{(m-1)}, \tag{1.60}$$

$$\frac{\partial L}{\partial b_j^{(m)}} = \frac{\partial L}{\partial z_j^{(m)}} \cdot \frac{\partial z_j^{(m)}}{\partial b_j^{(m)}} = \delta_j^{(m)}. \tag{1.61}$$

We do this iteratively until we reach the first layer. After that, we employ the optimization step in which we tune the parameters to follow the negative gradient step as seen in 1.3.3. This step represents the crux of this work, which focuses on the efficient adjustment of network parameters by incorporating not only gradient information but also second-order information to achieve faster convergence. Algorithm 3 provides an overview of the backpropagation algorithm in pseudocode.

Taken from [17]

In addition to the gradient calculation and optimization steps, there are many other factors that determine the success of training a neural network. A crucial aspect of training a neural network is finding a good set of values for the hyperparameters, as most deep learning algorithms come with several hyperparameters that control various aspects of the algorithm's behavior. Key hyperparameters include the learning rate, the number of epochs, the batch size, the architecture of the network (number of layers and neurons per layer) and as weighting factors. Optimizing these hyperparameters is highly non-trivial and are either determined through manual or automatic selection. The goal is to find a set of hyperparameter such that the generalization error is as small as possible. To facilitate this optimization, the training data is typically divided into three distinct sets: the *training set*, the *validation set*, and the *test set*. The training set is used to train the model, while the test set is reserved for evaluating the model's final performance. The validation set is specifically used for tuning hyperparameters and monitoring the model's performance to avoid overfitting (i.e., achieving low training error but high generalization error), as the generalization error for many hyperparameters, such as the learning rate, often follows a U-shaped curve when plotted as a function of the hyperparameter, making careful selection of these hyperparameters essential. As we have already stated, overfitting and underfitting are common pitfalls during model training, resulting in the model either memorizing the training data (overfitting) or

Algorithm 3 Neural Network Backpropagation [32]

Require: Training data $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, learning rate α
Initialize weights W and biases b randomly
repeat
  for all training example $(x^{(i)}, y^{(i)})$ **do**
    Compute Loss:
    Compute loss $L(y, \hat{y}^{(M)})$
    Backward Pass:
    Compute $\delta^{M} = \frac{\partial L}{\partial z^{(M)}} = \frac{\partial L}{\partial \hat{y}^{(M)}} \odot \frac{\partial \hat{y}^{(M)}}{\partial z^{(M)}}$
    for $m = M - 1$ **to** 0 **do**
      Compute $\delta^{(m)} = [\theta^{(m+1)^T} \delta^{(m+1)}] \nabla_{z^{(m)}} f$
      Compute gradients:
      $\frac{\partial L}{\partial \theta_{i,j}^{(m)}} = \delta_j^{(m)} \odot \hat{y}_i^{(m-1)}$
      $\frac{\partial L}{\partial b_j^{(m)}} = \delta_j^{(m)}$
    end for
  end for
  for $l = 0$ **to** M **do**
    Update weights: $\theta^{(l+1)} \leftarrow \theta^{(l)} - \alpha \frac{\partial L}{\partial \theta^{(l)}}$
    Update biases: $b^{(l+1)} \leftarrow b^{(l)} - \alpha \frac{\partial L}{\partial b^{(l)}}$
  end for
until convergence

failing to learn the training data at all (underfitting). A common method to tackle this problem is *regularization*, which is often defined as "any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error" [17]. In theory, regularization is used to reduce the amount of variance in a model, at the cost of increasing the model's bias. This is often referred to as the *bias-variance tradeoff*. While the bias is a measures for the inherent inability of a model to capture the true relationship of the data, variance measures the variability of predictions across different sets of data. Thus reducing the variance of a model, results in a better generalization ability on unseen data[20]. There are various techniques to facilitate regularization, one of the most widely used regularization techniques is the norm penalty on the model's parameters. This introduces a penalty $\Omega(\theta)$, where Ω is a norm and θ represents the network's parameters. We denote the regularized objective function by $\tilde{J}$ such that we get:

As described in [17]

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \lambda \Omega(\theta), \tag{1.62}$$

where λ is a hyperparameter that controls the strength of the regularization and is often called the *weight decay*. In practice we only consider the *weights* of the network for regularization instead of all parameters θ [17]. The most widely used norms for Ω are the *L1* norm and the *L2* norm. The *L2* norm follows the definition of $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$. This formulation incentivizes the network to have *weights* that lie closer to the origin by pushing parameters whose directions, corresponding to eigenvectors of the Hessian of J with small eigenvalues (i.e., directions with small

curvature), closer to zero. The *L1* norm is defined as $\Omega(\theta) = \|\mathbf{w}\|_1 = \sum_i |\theta_i|$. It can be shown that *L1* regularization encourages sparsity in the solution, meaning that many parameters are driven to an optimal value of zero. This property of the *L1* regularizer is often used in a mechanism called *feature selection*, where the model learns which features are most important by driving many parameters to zero. Other important regularization techniques that do not influence the loss function are *dropout* and *early stopping*. *Dropout* is based on the idea of training an ensemble of subnetworks during the training of the larger model. This is achieved by multiplying the output of a unit by zero with a given probability, effectively removing the unit from the network. This way, a large number of subnetworks are trained to predict the correct output, even when there is "brain damage" present. In practice, this often leads to better generalization abilities of the entire model. *Early stopping* is another regularization technique that helps to prevent overfitting by monitoring the model's performance on the validation set during training. We stop the training onnce the model's performance on the validation step stops improving (or starts worsening), even if the performance on the training set continues to improve. Although simple, this form of regularization has been proven to be very effective. [17]

1.5.4 *Decoupled Weight Decay [23]*

Decoupled Weight Decay introduces a novel approach that reorders the application of the network parameters θ within the optimizer. The authors find that the traditional implementation of weight decay, which is the addition of $\lambda\theta$ to the gradient in the optimizer [23], is effectively equivalent to L_2 regularization for standard stochastic gradient descent (SGD). Given our *L2* regularized loss $\tilde{J}(\theta; X, y)$, the gradient of this regularized loss function is

$$\nabla_\theta \tilde{J}(\theta; X, y) = \nabla_\theta J(\theta; X, y) + \lambda\theta. \tag{1.63}$$

This yields the optimizer's update step

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \tilde{J}(\theta; X, y) = \theta_t - \eta(\nabla_\theta J(\theta; X, y) + \lambda\theta_t) \tag{1.64}$$

as described in [24].

This equivalence does not hold for adaptive gradient optimizers such as *Adam*. In adaptive optimizers, the weight decay must be decoupled from the gradient update to maintain effective regularization and optimization [23]. In the standard implementation of Adam, *weight decay* is incorporated into the gradient update step as follows:

$$\begin{aligned} g_t &= \nabla_\theta J(\theta_t) + \lambda\theta_t, \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \end{aligned}$$

Here, the weight decay term $\lambda\theta_t$ is added directly to the gradient of the loss function $J(\theta_t)$, which means the regularization is coupled with the gradient update.

In *decoupled weight decay*, however, the weight decay is applied simultaneously but separately from the gradient update step. This results in a new update rule:

$$\theta_{t+1} = \theta_t - \eta(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} + \lambda\theta_t). \tag{1.65}$$

In this formulation:

- θ_t represents the parameters at iteration t,
- λ is the weight decay coefficient.

The decoupled weight decay ensures that the weight decay term $\lambda\theta_t$ is applied separately from the gradient step, avoiding interference with the adaptive nature of the gradient updates, where weights with large gradient magnitudes are regularized by a smaller relative amount than other weights. [23].

1.5.5 *Exact Calculation of the Hessian Matrix for MLPs [5]*

In this section we will discuss how we can calculate the exact Hessian of a Loss using the backpropagation algorithm that we covered earlier. We see that the exact values of the Hessian can be computed using only a few forward and backward propagations. It should be noted that although an efficient form of calculation for the Hessian might be tempting, the storage requirements for such computations are infeasible for larger neural networks. Therefore, in practice, only approximations of the Hessian or its diagonal are considered. Unless otherwise specified, all information of this section are taken from [5]. We will start by considering a feed-forward network with the standard notion introduced in 1.5.1

$$z_i = \sum_j \theta_{ij} a_j + b_i \qquad a_i = f(z_i), \tag{1.66}$$

with z_i and a_i beeing an output of the previous layer and it's corresponding current activation. We now want to find the first and second derivatives of an error function E, which we model to consist of a sum of individual errors computed for each training instance

$$E = \sum_p E_p, \tag{1.67}$$

where p labels the data point. We now consider a simple feedforward architecture without skip or feedback connections. Without loss of generality, we assume that unit i is in the same layer as unit n, or in a lower layer. Due to the symmetry of the Hessian (see 1.1.5), the remaining terms do not have to be computed. Utilizing the chain rule, we can can formulate this as

$$\frac{\partial^2 E_p}{\partial\theta_{ij}\partial\theta_{nl}} = \frac{\partial z_i}{\partial\theta_{ij}}\frac{\partial}{\partial z_i}\left(\frac{\partial E_p}{\partial\theta_{nl}}\right) = a_j\frac{\partial}{\partial z_i}\left(\frac{\partial E_p}{\partial\theta_{nl}}\right). \tag{1.68}$$

We now introduce a set of quantities σ_n defined by

$$\sigma_n \equiv \frac{\partial E_p}{\partial z_n}. \tag{1.69}$$

Using this we can write the second derivative as

$$\frac{\partial^2 E_p}{\partial\theta_{ij}\partial\theta_{nl}} = a_j \frac{\partial}{\partial z_i}(\sigma_n a_l), \tag{1.70}$$

because of $\frac{\partial z_n}{\partial\theta_{nl}} = a_l$. Further we introduce some additional quantities

$$g_{li} \equiv \frac{\partial z_l}{\partial z_i} \qquad b_{ni} \equiv \frac{\partial \sigma_n}{\partial z_i}. \tag{1.71}$$

Utilizing the product rule, the second derivatives can now be written in the following form

$$\frac{\partial^2 E_p}{\partial\theta_{ij}\partial\theta_{nl}} = a_j \sigma_n \frac{\partial f}{\partial z_l} g_{li} + a_j a_l b_{ni}, \tag{1.72}$$

where $\frac{\partial f}{\partial z_l} = \frac{\partial a_l}{\partial z_l}$ and $\frac{\partial f}{\partial z_l} g_{li} = \frac{\partial a_l}{\partial z_i}$ Using the chain rule for partial derivatives we can evaluate the g_{li} as follows

$$g_{li} = \sum_r \frac{\partial z_l}{\partial z_r}\frac{\partial z_r}{\partial z_i}, \tag{1.73}$$

where the sum runs over all units r which send connections to unit l. The above equation can be recursivly defined as

$$g_{li} = \sum_r f'(z_r)\theta_{rl} g_{ri}, \tag{1.74}$$

where $g_{ri} = \frac{\partial z_r}{\partial z_i}$ and $\frac{\partial z_l}{\partial z_r} = \frac{\partial z_l}{\partial a_r}\frac{\partial a_r}{\partial z_r} = \theta_{rl}\frac{\partial f}{\partial z_r}$. To obtain the initial conditions, we set $g_{ii} = 1$ and $g_{li} = 0$ for all units $l \neq i$ in the same or lower layers (i.e., layers nearer to the output). All the other g_{li} will be determinded during forward propagation using the above recursive equation. We can obtain the values for σ_n in a very similar fashion

$$\sigma_n = \sum_r \frac{\partial E_p}{\partial z_r}\frac{\partial z_r}{\partial z_n}, \tag{1.75}$$

where the sum runs over all units r to which unit n sends connections. Therefore we get

$$\sigma_n = \sum_r \frac{\partial E_p}{\partial z_r}\frac{\partial z_r}{\partial z_n} = \sum_r \sigma_r \frac{\partial z_r}{\partial z_n} = \sum_r \sigma_r \theta_{rn}\frac{\partial a_n}{\partial z_n} = f'(z_n)\sum_r \sigma_r\theta_{rn}. \tag{1.76}$$

To obtain the initial condition for σ_m, where m is the label of an output unit, we derive

$$\sigma_m = \frac{\partial E_p}{\partial z_m} = \frac{\partial E_p}{\partial a_m}\frac{\partial a_m}{\partial z_m} = f'(z_m)\frac{\partial E_p}{\partial a_m}. \tag{1.77}$$

Next up, we derive a generalized back-propagation equation for the b_{ni}. Substituting the back-propagation formula of σ_n into the definition of b_{ni}, we get

$$b_{ni} = \frac{\partial}{\partial z_i}\left(f'(z_n)\sum_r \theta_{rn}\sigma_r\right). \tag{1.78}$$

calcuting the derivative gives

$$b_{ni} = f''(z_n) g_{ni} \sum_r \theta_{rn} \sigma_r + f'(z_n) \sum_r \theta_{rn} b_{ri}. \tag{1.79}$$

Using the above relationships, we can formulate the initial condition for b_{mi}

$$b_{mi} = \frac{\partial \sigma_m}{\partial z_i} = \frac{\partial}{\partial z_i} \left(\frac{\partial E_p}{\partial z_m} \right) = \frac{\partial^2 E_p}{\partial z_m^2} \frac{\partial z_m}{\partial z_i} = H_m g_{mi}, \tag{1.80}$$

with

$$H_m \equiv \frac{\partial^2 E_p}{\partial z_m^2} = f''(z_m) \frac{\partial E_p}{\partial a_m} + (f'(z_m))^2 \frac{\partial^2 E_p}{\partial a_m^2}. \tag{1.81}$$

Once we have obtained the initial values of all the b_{mi}, the b_{ni} of the remaining units, are determinded via back-propagation. To summarize this process, for each pattern or data point p:

1. Perform forward-propagation to calculate all a_n and g_{li} using their respective equations.
2. Execute back-propagation to determine σ_n and b_{ni}.
3. Finally, evaluate the value of $\frac{\partial^2 E_p}{\partial \theta_{ij} \partial \theta_{nl}}$ using the determinded values.

The total number of distinct forward and backward propagations required per training pattern is equal to twice the number of hidden and output units in the network. The number of operations for each propagation scales with N, where N is the total number of weights in the network[5].

2

NUMERICAL EVALUATIONS

In Chapter 1, we covered the mathematical foundations of second-order optimization and its application within the context of neural networks. We discussed both fundamental and cutting-edge techniques for first-order and second-order optimization, providing a comprehensive overview of the field. In this chapter, we examine our experimental results, by comparing the performance and convergence properties of each optimizer including second-order optimization methods, namely AdaHessian[43] and Apollo[25], against established first-order optimizers on common datasets across multiple domains. Our research will be conducted on two computer vision datasets, namely CIFAR and (Tiny)ImageNet, as well as WMT-14, a machine translation dataset, which will be used to train a transformer model. For each of these datasets, we will discuss in depth the hyperparameter settings and model selection process used in our experiments, and relating them to our results. This detailed documentation ensures that others can easily reproduce our results. Furthermore, we provide a thorough comparison of our findings with those reported in the original papers of AdaHessian and Apollo. We critically analyze these results, offering insights into the strengths and weaknesses of each method.

2.1 OVERVIEW OF SOFTWARE AND TOOLS USED

To efficiently evaluate the optimizers across various datasets with a range of hyperparameters, we developed a comprehensive benchmarking framework in Python 3.6.8. The code for this framework is publicly accessible on our GitHub repository[1]. and is available for anyone to use and modify. The core of our framework is built on *PyTorch*. *PyTorch* is a popular open-source machine learning framework known for its dynamic computation graph and easy to integrate GPU acceleration. For datasets and pre-built computer vision model architectures, we utilized the PyTorch-associated *torchvision* module. Additionally, *NumPy* is used in some of the utility functions and to prepare data for use with *matplotlib*. From the hardware side, most of the experiments, unless otherwise specified, were conducted on *Nvidia A40* GPUs. These GPUs are equipped with 48 *GB* of GDDR6 memory and deliver FP32 performance of up to 19.5 TFLOPS. For efficient multi-GPU training, we used the PyTorch *DataParallel* module, which splits the input across the specified GPU's by chunking along the batch dimension[31]. Because *AdaHessian* uses `torch.autograd.grad` to compute the second order derivatives, we were not able to utilize the more efficient `DDP` (Distributed Data Parallel) package, as DDP currently doesn't work with the construction of derivative graphs. For more information on this, see this issue[2].

1 https://github.com/Neural-Opt/second_order_optim_models
2 https://github.com/pytorch/pytorch/issues/63812

2.2 IMAGE CLASSIFICATION

For evaluation, we choose the most widely used first-order optimizers, as well as AdaHessian and Apollo for second-order optimizers. In addition to the regular metrics of *Training Accuracy*, *Training Loss*, *Test Accuracy*, and *Test Loss*, we also evaluate the speed of each optimization step and its memory usage during this step. This helps assess the real-world applicability of the second-order optimizers. Finally, we introduced a new metric called *Time till Convergence (TTC)*. *TTC* measures the time in epochs each optimizer needs for the *Training Loss* to converge. We compute the Time till Convergence (TTC) by following Algorithm 4, which takes as input the loss data, a threshold, and a window size. To prevent outliers from distorting the results, we rely on short-term intervals, from which we calculate the mean. For learning rate scheduling, we employ both *milestone decay* as well

Algorithm 4 Time till Convergence (TTC) Calculation

Input: Loss array `data`, treshold `T`$= \frac{\texttt{data}[0]}{100}$, window size $\sigma = 5$
Initialize: old_mean $\leftarrow$ *mean*(`data`$[|\texttt{data}| - \sigma :]$)
for $i = n - \sigma, n - \sigma, \ldots, 0$ **do**
    new_mean $\leftarrow$ *mean*(`data`$[i : i + \sigma]$)
    if $|\text{new\_mean} - \text{old\_mean}| \leq$ `T` **then**
        continue
    else
        `ttc` $\leftarrow i + \arg\min(\text{new\_mean})$
        break
    end if
end for

as *cosine annealing*. While *milestone decay* decays the learning rate by a factor of γ at specified milestones (certain epochs), *cosine annealing* gradually reduces the learning rate following a cosine function. For our experiments, we use *PyTorch's* implementation of `MultiStepLR` for milestone decay and `CosineAnnealingLR` for cosine annealing. In the next sections, we discuss the obtained results on the CIFAR-10 and TinyImageNet datasets, as well as the chosen hyperparameters for the optimizers and learning rate schedulers.

2.2.1 *CIFAR-10*

The CIFAR-10 dataset comprises 60,000 RGB images, each with dimensions of 32x32 pixels, categorized into 10 different classes, with 6,000 images per class and a training/test split of 50000/10000. For evaluation, we utilize the ResNet-110 architecture. Unlike ResNet-18, ResNet-110 is specifically designed to handle smaller images, such as those found in the CIFAR-10 dataset. ResNet-110 comprises 54 BasicBlocks, divided into 3 layer groups, each containing 18 blocks. Each block in ResNet-110 is two layers deep. The first layer consists of a 3x3 convolution, followed by batch normalization and an activation function. The second layer also applies a 3x3 convolution and includes a skip connection that directly adds the input of the

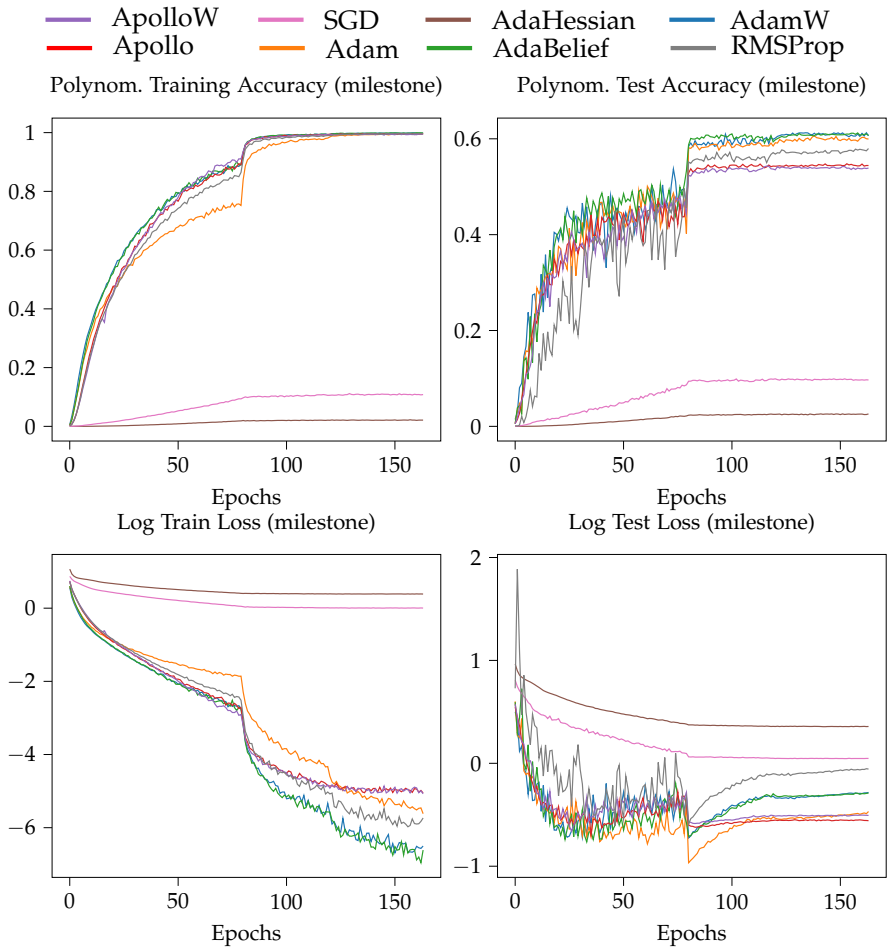

Figure 2.1: Evaluation of optimizers on CIFAR-10 using ResNet-110 with the *milestone* learning rate scheduler, where hyperparameters are held constant across all optimizers. For better visualization we applied a polynomial transformation, with $\hat{x} = x^{\alpha}$ and $\alpha = 5$, for every $x \in \mathcal{D}$ in the output data $\mathcal{D}$.

block to the output of the second layer. This is then followed by another application of an activation function (ReLU). After each layer group, the channel width doubles. This results in 16 channels for the first block group and 32 and 64 channels for the second and third block groups, respectively. Together with the fully connected classification layer of size 64x10, where 10 corresponds to the number of classes, this totals 110 layers [18]. Given Apollo's status as a recent advancement in the development of second-order optimizers, we closely adhere to their evaluation methodology. This allows us to effectively compare their results with our own in subsequent analyses. We therefore set our training batch size to 128. For the milestone lr-scheduler, we choose milestone epochs of 80 and 120 with $\gamma = 0.1$, meaning that the learning rate is decayed by a factor of 10^{-1} at epochs 80 and 120. For both the cosine-annealing and milestone learning rate scheduler, we train the model for 164 epochs. To ensure a fair comparison between first and second-order optimizers, we employed two key strategies. First, we addressed the potential for inherent advantages due to superior hyperparameter settings. To mitigate this, we

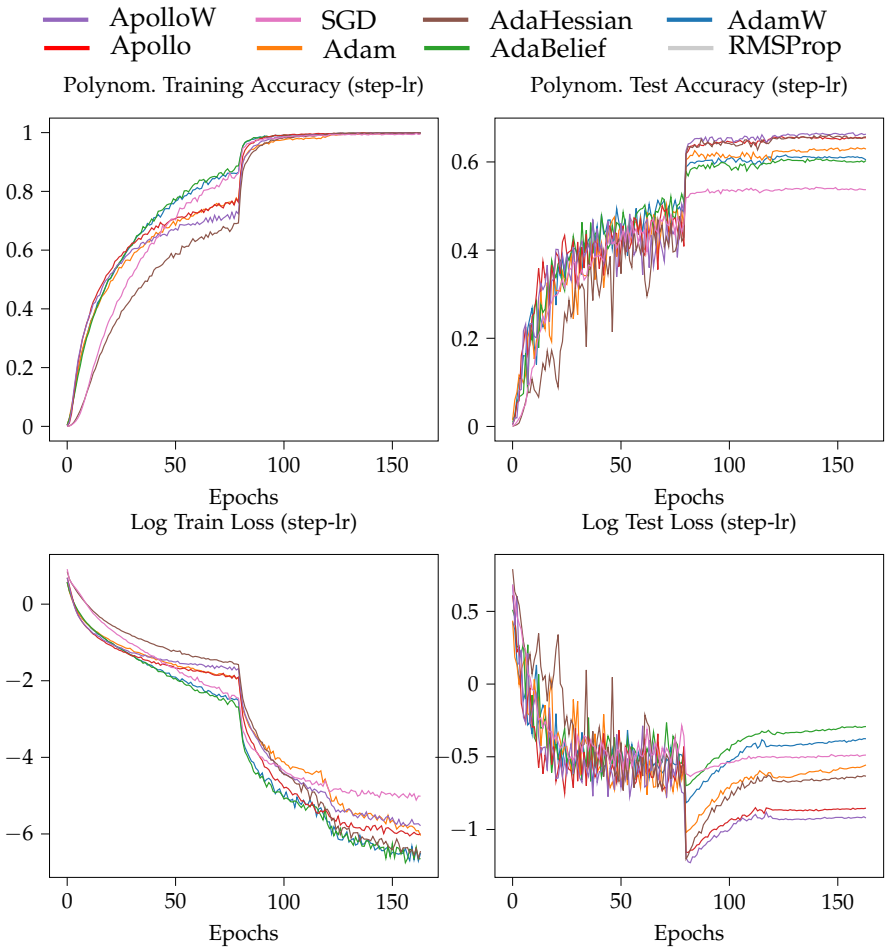

Figure 2.2: Evaluation of optimizers on CIFAR-10 using ResNet-110 with the *milestone* learning rate scheduler, where hyperparameters are choosen optimally across all optimizers.For better visualization we applied a polynomial transformation, with $\hat{x} = x^{\alpha}$ and $\alpha = 5$, for every $x \in \mathcal{D}$ in the output data $\mathcal{D}$.

applied the optimal hyperparameters of Adam or AdamW—widely regarded as industry standards—to the second-order optimizers. The authors of Apollo [25] note that both Apollo(W) and AdaHessian benefit significantly from learning rate warmup. However, our goal was to evaluate the real-world applicability of these second-order optimizers under basic conditions. Therefore, we opted for the most rudimentary settings to assess whether Apollo(W) and AdaHessian could perform well without such enhancements. This approach tests the optimizers' effectiveness in real-world conditions, where complex tuning is often impractical. Secondly, we test all optimizers with their respective optimal parameters as mentioned in [25]. As discussed in Section 1.5.4, *weight decay* introduces an additional constraint to the optimizer. This technique aids in regularization, though it can sometimes result in slower convergence. To ensure fairness in our comparison, we applied the same learning rate and weight decay across all optimizers, which are listed in **??**.

Examining the results presented in Figure 2.1, which utilizes milestone learning rate decay, we observe several key findings. *AdamW* and *AdaBelief* converge the

fastest with the best generalization. Although not optimally tuned, *Apollo(W)* shows reasonable convergence and generalization, while *AdaHessian* performs poorly with the chosen hyperparameters. When optimally tuned however, as shown in Figure 2.2, *Apollo(W)* and *AdaHessian* are capable of outperforming other optimizers by a small margin. This suggests that *AdaHessian* may require additional effort to tune its hyperparameters optimally, as commonly used settings from first-order methods have a much worse impact on performance compared to *Apollo(W)*.

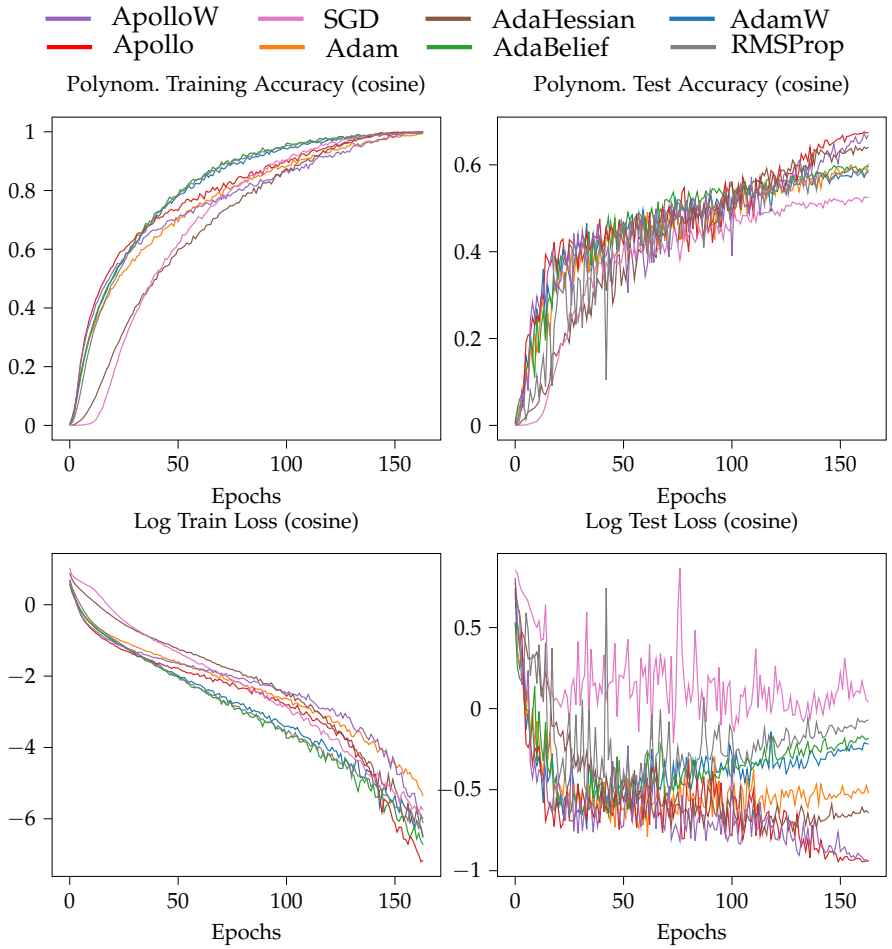

Figure 2.3: Evaluation of optimizers on CIFAR-10 using ResNet-110 with the *cosine annealing* learning rate scheduler, where hyperparameters are choosen optimally across all optimizers. For better visualization we applied a polynomial transformation, with $\hat{x} = x^{\alpha}$ and $\alpha = 5$, for every $x \in \mathcal{D}$ in the output data $\mathcal{D}$.

Using the cosine annealing learning rate scheduler in our first evaluation strategy, we observe the same ordering of optimizers in terms of generalization and convergence behavior (see Figure **??**). To expand on our second evaluation strategy, Figure 2.3 illustrates the performance of each optimizer when given its optimal set of hyperparameters, as specified in [25]. As previously mentioned, *AdaHessian*, *Apollo* and *ApolloW* demonstrate superior performance in terms of generalization error. While both *Apollo* and *ApolloW* are able to converge faster than *AdaHessian*, especially in the early epochs, *ApolloW* converges more slowly than its decou-

pled first-order counterparts, *AdamW* and *AdaBelief*. While *AdaHessian*, *Apollo* and *ApolloW* perform similiar in the setting of 2.3, *AdaHessian* exhorts much higher variability when compared to **??**. This observation lends credence to the claim in [25] that *Apollo* and *ApolloW* may be more practical than *AdaHessian* for real-world applications.
Expanding on *Apollo's* better generalization abilities, figure 2.3 shows that *Apollo* and *ApolloW* continue to decrease their test loss even at epoch 164, while *Adam* and *AdamW*, experience a steadily increasing test loss.

In Table 2.3, we can observe the maximum memory consumption of each optimizer and the duration of one optimization step realtive to *SGD*. Not surprisingly, *Apollo* and *AdaHessian* both take longer for each step and consume more memory (see Table 2.3). While the increase in computation time and GPU memory is rather modest for *Apollo*, we can see that *AdaHessian* takes significantly longer for computation compared to both *Apollo* and the first-order methods, as well as consumes much more memory. This is likely the result of the second backward pass *AdaHessian* uses to compute the second-order gradients, which also limits the use of *AdaHessian* in practice.

Table 2.1: Accuracy (%) of different optimizers across CIFAR-10 and TinyImageNet, evaluated on 3 runs (CIFAR-10)

	CIFAR-10		Tiny ImageNet	
	Milestone	**Cosine**	**Milestone**	**Cosine**
SGD	87.2 ± 0.86	88.17 ± 0.3	39.32	38.56
Adam	90.8 ± 0.39	90.32 ± 0.08	43.06	43.44
AdamW	90.35 ± 0.06	89.99 ± 0.26	42	41.71
AdaBelief	90.23 ± 0.18	90.08 ± 0.18	41.88	41.72
RMSProp	90.1 ± 0.12	89.5 ± 0.33	39.4	39.9
Apollo	91.88 ± 0.38	**92.18** ± 0.18	**44.96**	**44.32**
ApolloW	**92.03** ± 0.08	92.02 ± 0.17	43.06	43.96
AdaHessian	91.64 ± 0.26	91.81 ± 0.55	44.28	44.08

2.2.2 *Tiny ImageNet*

For the second image classification dataset, we use Tiny ImageNet [37]. Tiny ImageNet is a more manageable subset of the well-known ImageNet dataset. It consists of 100,000 RGB images, each sized 64x64 pixels, across 200 different classes. For training, we again follow the configuration outlined by *Apollo* [25] and utilize a ResNet-18 model with approximately 11.7 million parameters. We found that deeper ResNet architectures, like *ResNet-50*, lead to significantly more overfitting during training. The training is conducted with a batch size of 256. We implement milestone decay, adjusting the learning rate at epochs 40 and 80 by a factor of $\gamma = 0.1$. The model is trained for 120 epochs, employing both milestone decay and

Table 2.2: Time (epochs) until convergence (see 4) of the training loss across CIFAR-10 and TinyImageNet

	CIFAR-10		Tiny ImageNet	
	Milestone	**Cosine**	**Milestone**	**Cosine**
SGD	86 ± 5.0	113 ± 1.0	44	**52**
Adam	94 ± 1.0	132 ± 1.0	57	84
AdamW	81 ± 0.0	112 ± 3.0	**40**	59
AdaBelief	81 ± 0.0	**108** ± 4.0	**40**	60
RMSProp	**80** ± 0.0	**108** ± 1.0	**40**	58
Apollo	84 ± 1.0	125 ± 1.0	42	60
ApolloW	87 ± 1.0	137 ± 1.0	43	60
AdaHessian	88 ± 0.0	121 ± 2.0	53	72

cosine annealing strategies. Our strategy for hyperparameter evaluation remains consistent with those used for CIFAR-10. We first test the optimizers' performance by evaluating the second-order optimizers on the optimal values of the popular first-order optimizers. These settings are given in **??**

As we can see in 2.4 and 2.5, the performance of the second-order optimizers largely follows the same regime as in 2.1 and 2.2. *AdaHessian* and *SGD* still aren't able to perform well with their non-optimal hyperparameters. Although it shows that they both have a steadily decreasing test loss, meaning they are able to prevent overfitting much better than *AdamW* or *AdaBelief*. Referring to 2.6, we can see that *AdaHessian* still shows a high amount of performance variance between its optimal and non-optimal hyperparameter settings, while *Apollo* and *ApolloW* have less variance in their performance. As both second-order methods are able to reduce the amount of overfitting, especially in the later epochs (100-120), and therefore achieve better generalization results, we could conclude that they are able to find flatter minima in the loss landscape, that are generally associated with better generalization [17]. In terms of convergence behavior, we can see that although *Apollo* and *ApolloW* are able to converge faster than *AdaHessian*, they are still considerably slower than *AdamW* and *AdaBelief* (see 2.2). Therefore, we can conclude that, at least in its optimal setting, *Apollo* is able to offer a reasonably good trade-off between convergence time and generalization performance. Meanwhile, *AdaHessian* is severely limited in its practical applicability not only by its variability in performance, but also by its memory usage.

2.3 MACHINE TRANSLATION

For the task of machine translation, we utilize the MarianMT framework, specifically designed to train sequence-to-sequence translation models. The pre-built models in MarianMT are Transformer-based and share the same architecture as BART (Bidirectional and Auto-Regressive Transformers). In contrast to *PyTorch's*

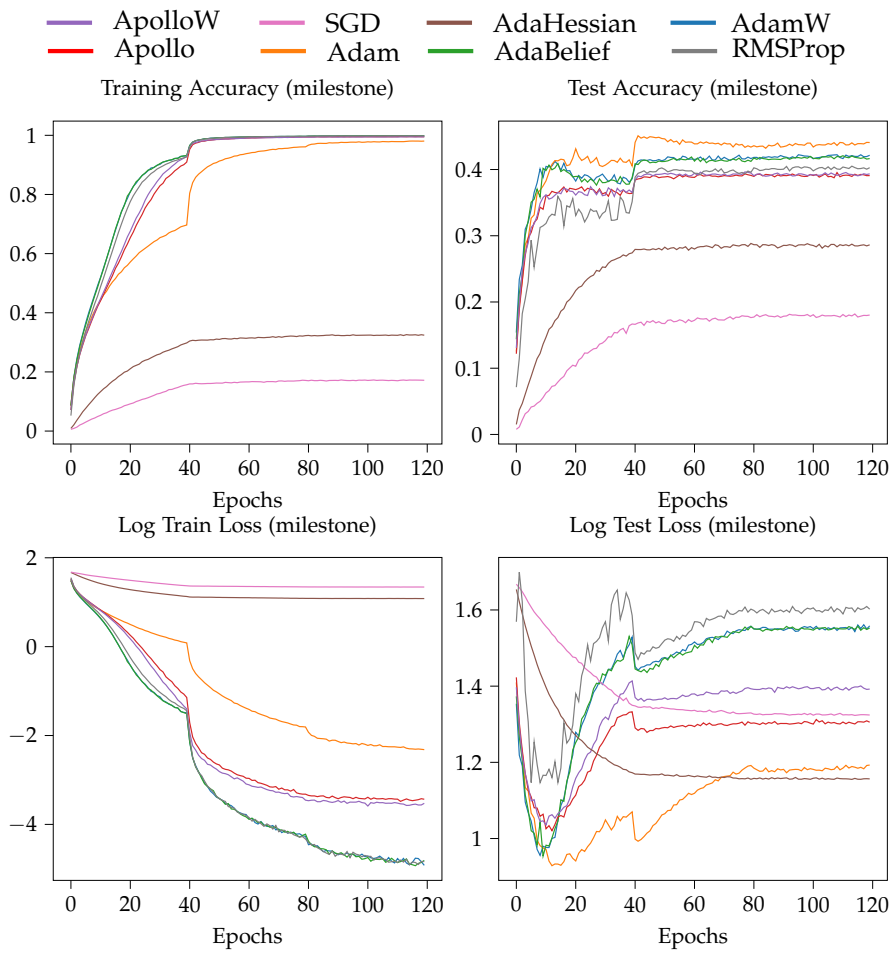

Figure 2.4: Evaluation of optimizers on TinyImageNet using ResNet-18 with the *milestone* learning rate scheduler, where hyperparameters are held constant across all optimizers

`nn.Transformer` module, MarianMT abstracts not only the Encoder and Decoder blocks but also the positional encoding and embedding layers. Which makes it much easier to work with. Additionally, MarianMT provides a pre build tokenizer, which we use for our input sequences.

2.3.1 *WMT-14*

The WMT-14 (Workshop on Machine Translation 2014) is a benchmark dataset for machine translation tasks. It offers parallel texts in various language pairs, with German-English being particularly prominent. In out translation task we utilizes the German-English portion of the WMT-14 dataset, which contains about 4.5 Mio. sentence pairs. We utilize embedding layers with a dimensionality of 512 for both input and output tokens. The Transformer consists of 3 Encoder and 3 Decoder blocks, each employing 8 attention heads. Both the Encoder and Decoder incorporate feed-forward networks (FFN) with a hidden dimension of 512. We evaluate our model's translation quality using the BLEU (Bilingual Evaluation Understudy)

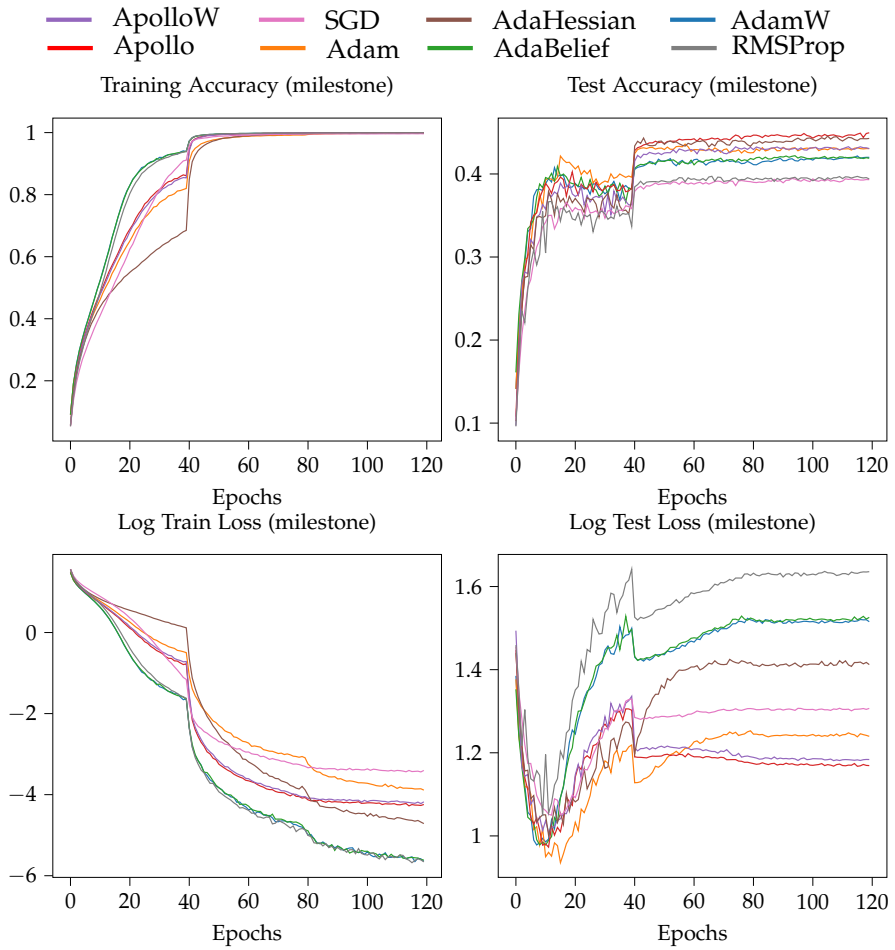

Figure 2.5: Evaluation of optimizers on TinyImageNet using ResNet-18 with the *milestone* learning rate scheduler, where hyperparameters where hyperparameters are choosen optimally across all optimizers.

score, calculated with the *sacrebleu* library. Our training procedure employs a batch size of 256 and limits input sentences to a maximum length of 128 tokens. For learning rate scheduling, we follow the original Transformer paper [40] and employ the `InverseSquareRootLR` function, which includes a warmup phase of 4,000 steps during which the learning rate increases linearly. Each optimizer was trained for 8 epochs which resulted in about 140k steps. The hyperparameters of each optimizer were carefully selected by evaluating the model on smaller datasets and comparing the performance of the optimizers. The final Hyperparameter settings are described in **??**. In Figure 2.7, we can see that both in terms of convergence and generalization performance, *AdaBelief* produces the best results from the tested first order methods. While *Apollo* and *ApolloW* convergence marginally faster than *Adam* and *AdamW*, *AdaBelief* is still performing slightly better in both domains. Only *AdaHessian* demonstrates an ability to converge about one epoch faster and shows significantly better performance in terms of the *BLEU score* when compared to *AdaBelief*. However, *AdaHessian* was about 2.5 times slower than *AdaBelief*, diminishing its convergence advantage in terms of wall clock time. *SGD* was able

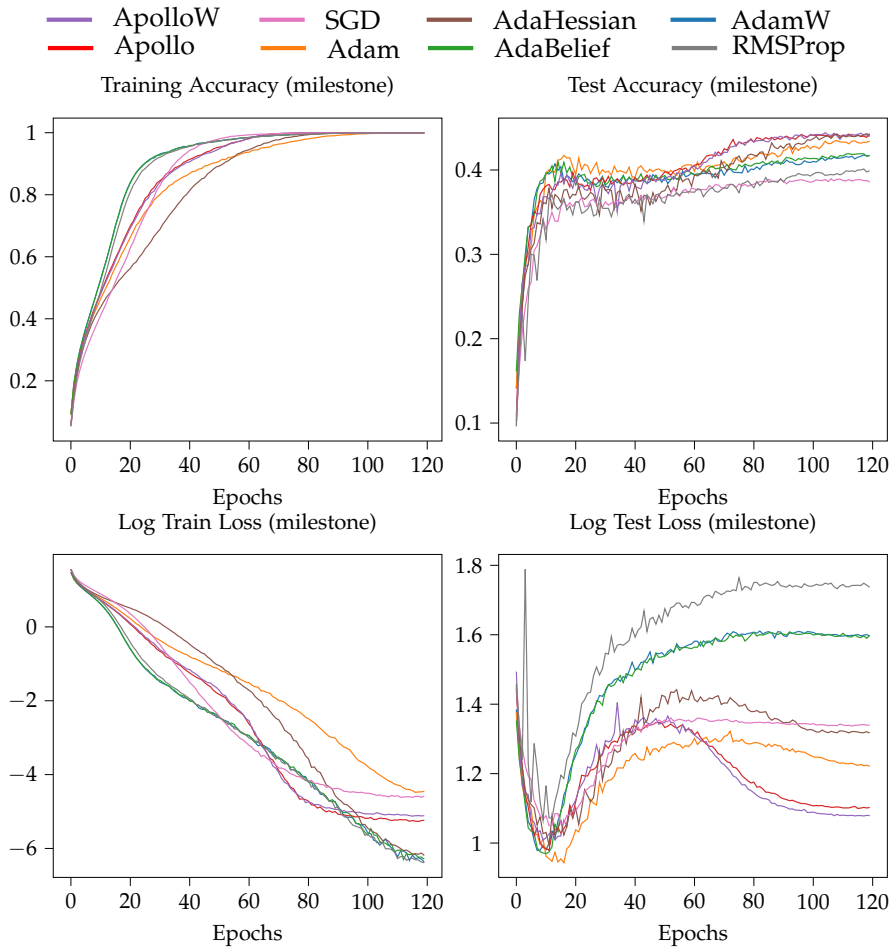

Figure 2.6: Evaluation of optimizers on TinyImageNet using ResNet-18 with the *cosine* learning rate scheduler, where hyperparameters where hyperparameters are choosen optimally across all optimizers.

to converge and achieve decent results, although slower convergence speed is to be expected when training Transformer based models with *SGD* [11]. Regarding *RMSProp*, we were unfortunately unable to find a suitable set of hyperparameters, even after extensive tuning.

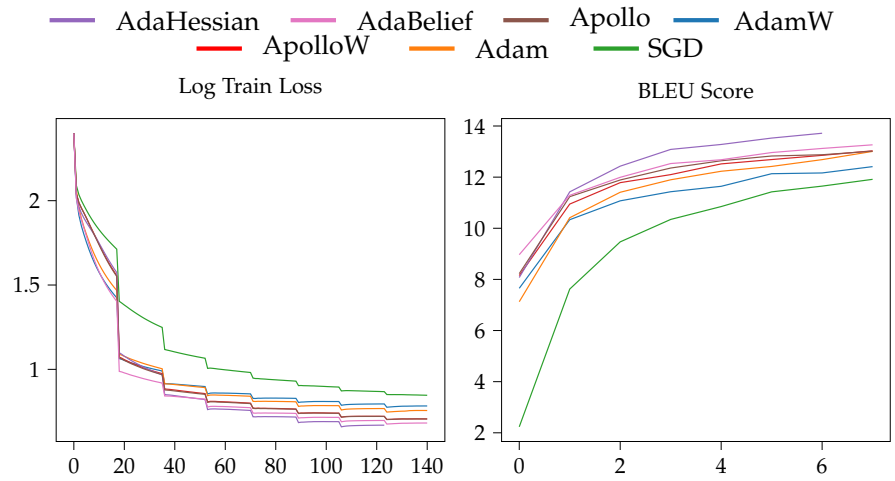

Figure 2.7: Evaluation of optimizers on WMT-14 using the Transformer architecture with the `InverseSquareRootLR` learning rate scheduler. Hyperparameters are individually tuned for optimal performance.

Table 2.3: Cost, Speed, and Memory Usage of Different Optimizers Across Various Datasets

	CIFAR-10		Tiny ImageNet		WMT-14	
Cost (x SGD)	Speed	Memory	Speed	Memory	Speed	Memory
SGD	1.0	1.0	1.00	1.00	1.00	1.00
Adam	1.0292	1.0112	1.0502	1.2192	1.0217	1.02174
AdamW	1.0317	1.0112	1.0458	1.2151	1.0299	1.02993
AdaBelief	1.0402	1.0112	1.0581	1.2273	1.0389	1.03895
RMSProp	1.0213	1.0	1.0274	1.0079	-	-
Apollo	1.1337	1.0223	1.1616	1.4313	1.1167	1.11669
ApolloW	1.1359	1.0223	1.1556	1.4336	1.1183	1.11828
AdaHessian	**2.6654**	**3.6443**	**1.9098**	**2.7969**	**2.6121**	**1.716**

3

HESSIAN APPROXIMIZATION QUALITY AND SAPOLLO

In this chapter, we will closely examine the behavior of *Apollo* and *AdaHessian* to understand how they differ from established first-order methods and whether their resource overhead is justified. As both *Apollo* and *AdaHessian* claim to provide more accurate second-order estimates than their predecessors, we therefore compare the calculated batch Hessian diagonal with the approximations generated by *Apollo*, *AdaHessian*, *Adam*, and *AdaBelief* across different batch sizes.

3.1 HESSIAN APPROXIMIZATION QUALITY

Before comparing the quality of the Hessian approximations provided by the previously mentioned optimizers, we need to address some additional details. Recalling their definitions, *Adam*, *AdaBelief*, and *AdaHessian* all use an estimate of the absolute curvature in their second-moment computations. The diagonal elements of the Hessian matrix of the loss function consist of the second-order partial derivatives with respect to each parameter $H_{ii} = \frac{\partial^2 L}{\partial \theta_i^2}$, where L is the loss function and θ_i is the i-th parameter. The entries H_{ii} are negative when the loss function is locally concave with respect to θ_i and positive when it is locally convex. Consequently, we will compare the *absolute* values of the Hessian diagonal elements with the optimizers' approximations, since they only consider the *absolute* curvature—that is, they ignore the sign of the Hessian diagonal entries, as they all use the squared approaximations in their second moment. To do so, we first calculate the Hessian diagonal for the current batch using `torch.autograd`. Since PyTorch does not natively support the calculation of the Hessian diagonal in a fully vectorized form, we implemented this by iterating over each gradient element, performing a second backward pass, and extracting the corresponding second-order derivative. Given that this process is inherently slow, we limited our investigation to a relatively small CNN model with 13.5K parameters, consisting of two convolutional layers and two fully connected layers. For training, we used the MNIST dataset, which contains 60,000 images of handwritten digits, each of size 28×28. To measure the similarity between the approximated Hessian diagonal and the actual batch Hessian diagonal, we needed a metric that is independent of the magnitude of the vectors. This is crucial because the scale of the second moments can be adjusted through the learning rate. Therefore, we utilized the cosine similarity measure, which is defined as follows:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}, \quad \mathbf{A}, \mathbf{B} \in \mathbb{R}^n.$$

This provides a measure of the angle between the two vectors, where $\cos(\theta) = 1$ indicates perfect directional alignment, and $\cos(\theta) = 0$ indicates orthogonality. We record the measured cosine similarity and plot its development throughout the training process. This is done for both batch sizes of 124 and 1024 to observe

whether the increased stochastic variance in the gradient significantly impacts the quality of the approximation results. In Figures 3.2 and 3.1, we observe the angle between the approximations and the *absolute* batch Hessian diagonal for both small and large batch sizes. For improved visualization, a moving average with a window size of 10 is applied to the small batch plots. The results indicate that *Apollo*'s approximations are much worse than those of the other optimizers across most layers and for both batch sizes.

Although the approximation quality of *Apollo* improves significantly with larger batch sizes, even approaching that of *Adam* and *AdaBelief*, it still fails to match their performance. *AdaHessian* is able to produces more accurate approximations than *Adam* and *AdaBelief* both in the small batch setting, as well as in the large batch setting. Finally, we see that *AdaBelief* performs similarly to *Adam*, but is able to provide a noticeably better approximation of the batch Hessian when less noisy gradient estimates are available. This brings us to the conclusion that, although we could only evaluate *Apollo* on a small model, it does not seem to live up to its claim of providing a better curvature approximation than traditional first-order methods. *AdaHessian*, on the other hand, is able to significantly outperform both traditional first-order methods, as well as *Apollo*, in terms of curvature approximation. However, due to the nature of *AdaHessian*'s stochastic approximation, it will always require a warm-up period for its approximation to become accurate, as $\text{diag}(H) = \mathbb{E}[z \odot (Hz)]$ (see 1.4.3) needs several evaluations of $z \odot (Hz)$. We can see this in Figure 3.2, were it starts of with a much higher degree until arriving at better approximates in later steps.

The good approximation performance of *AdaBelief*,in the *big* batch setting, may be explained by regarding the *EMA* of the belief term, $(g_t - m_t)^2$ (see 1.3.7), as a approximation for the diagonal entries of the gradient variance, as $Var(g)_{ii} = E[(g_i - E[g_i])^2] \quad g \in \mathbb{R}^n, i \leq n$. It can be shown that if we model our loss function as the negative log-likelihood, $\mathcal{L}(y, x, \theta) = -\log p(y|x, \theta)$, with $y \in \mathbb{R}^m$ beeing the lables of the input $x \in \mathbb{R}^n$, then

$$\text{Var}(\nabla \log p(y|x,\theta)) = \mathbb{E}[\nabla^2 \log p(y|x,\theta)] = -\mathbb{E}[H_{\log p(y|x,\theta)}] \quad [36]. \tag{3.1}$$

In this way, the belief term of *AdaBelief* directly approximates the diagonal entries of the expected negative Hessian. This might provide a new perspective on the superior approximation abilities of *AdaBelief*, as we have not seen this connection in the literature yet, to the best of our knowledge. Finally, the most interesting observation arises when we examine Figure 3.3, where the evolution of the loss is depicted for both batch sizes. We can see that, despite tuning each optimizer for optimal performance, with Hyperparameter in **??**, *AdaHessian* is unable to achieve the same loss as *Apollo*, even though it is capable of much more accurate curvature approximations. *Apollo* on the other hand is able to achive basicly the same convergence behavior as first-oder methods, although providing much less accurate curvature approaximations. This raises the question of whether very accurate curvature estimates might actually hinder the model's performance. It is conceivable that such precision could lead to an earlier discovery of local minima, which are suboptimal, whereas optimizers like *Apollo* may explore more of the loss surface while still accounting for curvature. However, as we observed in

the benchmarks on *CIFAR-10* and *WMT-14*, both *AdaHessian* and *Apollo* are able to discover minima that generalize well. This leads to the hypothesis that the better generalization performance of the tested second-order methods may not be solely based on superior Hessian approximation capabilities, but rather on other mechanisms that have yet to be uncovered. We will undermine this oberservation by uncovering several flaws in *Apollo*, both theoretical and practical, including issues related to its implementation in the next section. In conclusion, our analysis shows that *Apollo* is not competitive with standard first-order methods in terms of curvature approximation ability, at least within the context of our small testing network. Furthermore, since *AdaHessian* requires 2 to 3 times the memory of *SGD* (see Table 2.3) and due to PyTorch's current inability to work with gradient graphs on *DDP*, training large, distributed models becomes infeasible. Therefore, one might conclude that the approximation of *AdaHessian* is unlikely to find widespread application over existing first-order methods.

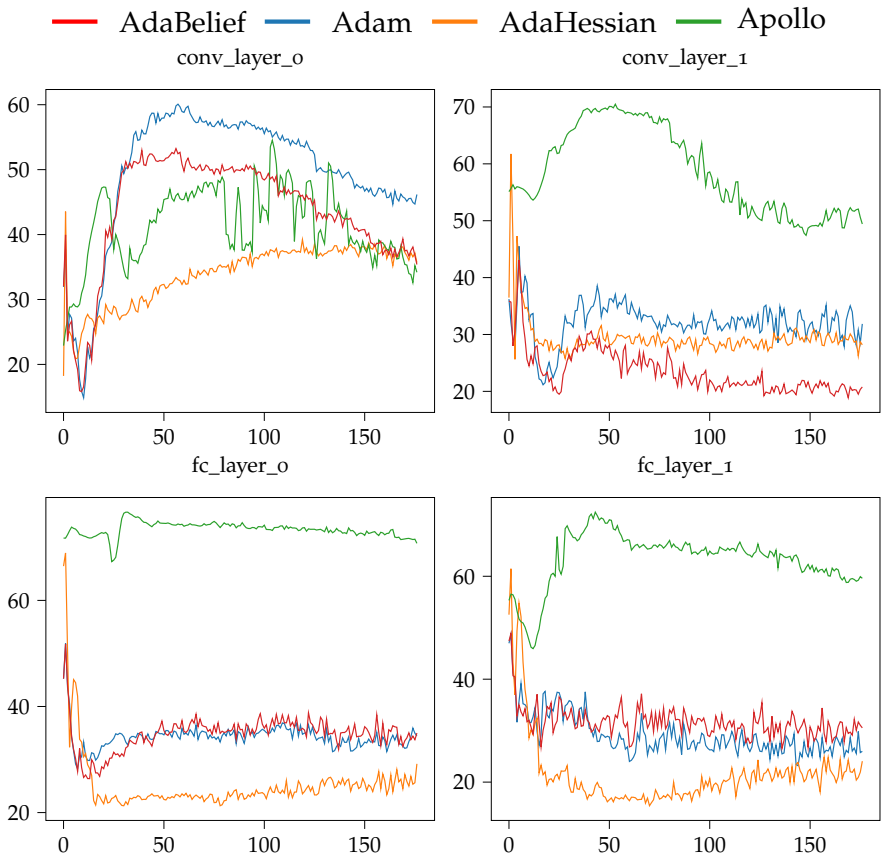

Figure 3.1: The cosine similarity (in degrees), y-axis, between the calculated batch Hessian diagonal and the corresponding optimizer approximations on a *big* batch (1028 samples). Optimizer updates are denoted on the x-axis. Note that these results represent only the Hessian diagonals for the network's *weights*. For the corresponding analysis on biases, please refer to Figure **??**.

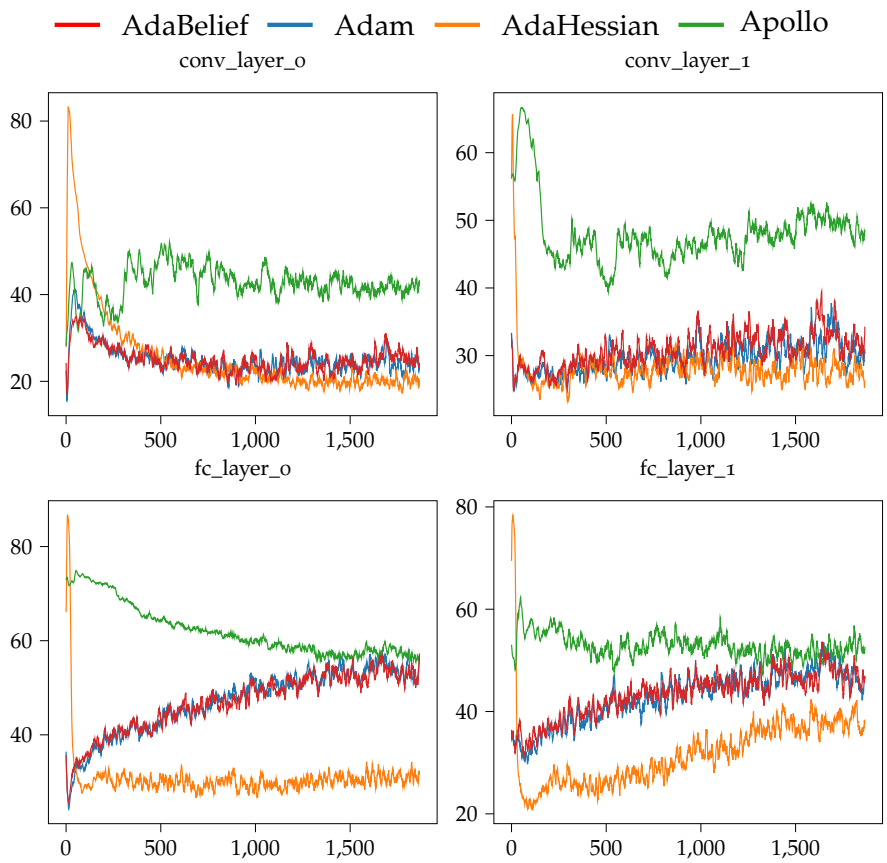

Figure 3.2: The cosine similarity (in degrees), y-axis, between the calculated batch Hessian diagonal and the corresponding optimizer approximations on a *small* batch (124 samples). Optimizer updates are denoted on the x-axis. Note that these results represent only the Hessian diagonals for the network's *weights*. For the corresponding analysis on biases, please refer to Figure **??**.

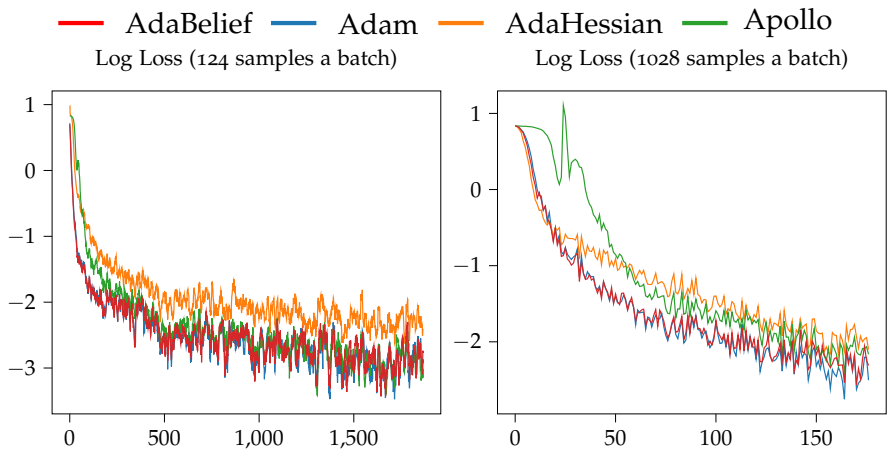

Figure 3.3: The log loss of the model during training, y-axis, after each update step, x-axis, while training with *small*- (left) and *big* batches (right) of training data.

3.2 THE SAPOLLO OPTIMIZER

As we noted in the previous section, the approximation quality of the batch Hessian in *Apollo* is not competitive with existing first-order methods. In this section, we will examine several flaws arising from the formulation and implementation of the *Apollo* algorithm (see Algorithm 1.4.4). We will then address both theoretical and practical issues, aiming to resolve these errors in the algorithm's design and implementation and comparing the results. We begin by demonstrating that the clamping operation in *Apollo* can lead to curvature information beeing lost, which causes the algorithm to behave similarly to *SGD* with momentum, thereby explaining its poor curvature approximation abilities.
For this we look at line **9** in 1.4.4. Here we can see that all values of B_t with $|B_t| \leq \sigma$ are clamped, losing their curvature information.

This implies that the clamping effect, due to the choice of σ in *Apollo*, cannot be rescaled by adjusting the learning rate. This is expected, as the clamping operation is inherently non-linear. In practice, this results in small values being capped by σ, which effectively leads to a rescaling of the learning rate by a factor of $\frac{1}{\sigma}$, disregarding any curvature information at that point. Thus, the higher the choice of σ, the more similar *Apollo*'s update step becomes to plain *SGD* with momentum. This phenomenon occurred in Figure 3.1, where most values of B were much smaller than the chosen σ, leading to a loss of curvature information. Note that in the standard implementation of *Apollo*, σ cannot be choosen as a hyperparamter, as it is hardcoded in the optimizer. To gain intuition for the small values of B from a theoretical perspective, we will again formulate the update rule of B, with

$$B_{t+1} = B_t - \frac{d_t^T(\Delta m_{t+1}) - (d_t^2)^T B_t}{\left(\|d_t\|_4 + \epsilon\right)^4} \cdot d_t^2 \tag{3.2}$$

$$= B_t - \frac{\sum_j^n d_{t,j}\Delta m_{t+1,j} - \sum_j^n d_{t,j}^2 B_{t,j}}{\left(\|d_t\|_4 + \epsilon\right)^4} \cdot d_t^2. \tag{3.3}$$

We now substitute $d_t = \frac{m_t}{\gamma}, \quad \gamma = \max(|B_t|, \sigma)$, and obtain the following for the k-th entry of B_t

$$\begin{aligned}
\Delta B_{t+1,k} &= \left(-\sum_j^n \frac{m_{t,j}}{\gamma_j}\Delta m_{t+1,j} - \sum_j^n \frac{m_{t,j}^2}{\gamma_j^2} B_{t,j} \right) \cdot \frac{m_{t,k}^2}{\gamma_k^2} \left(\|\frac{m_t}{\gamma}\|_4 + \epsilon \right)^{-4} \\
&= \left(-\sum_j^n \frac{m_{t,j}\Delta m_{t+1,j} m_{t,k}^2}{\gamma_j \gamma_k^2} - \frac{m_{t,j}^2 m_{t,k}^2}{\gamma_j^2 \gamma_k^2} B_{t,j} \right) \left(\|\frac{m_t}{\gamma}\|_4 + \epsilon \right)^{-4} \\
&\approx \left(-\sum_j^n \frac{m_{t,j}\Delta m_{t+1,j} m_{t,k}^2}{\gamma_j \gamma_k^2} - \frac{m_{t,j}^2 m_{t,k}^2}{\gamma_j^2 \gamma_k^2} B_{t,j} \right) \left(\|\frac{m_t}{\gamma}\|_4 \right)^{-4} \\
&= \Delta \hat{B}_{t+1,k}.
\end{aligned}$$

We will now examine the state of the algorithm in a situation where B_t is small, to observe how the updates behave in such a case. Specifically, we consider $B_{t,k} \leq \sigma$, and thus set $\gamma = \sigma$, which gives us the following expression

$$\begin{aligned}\Delta\hat{B}_{t+1,k} &\approx \left(-\sum_{j=1}^{n}\frac{m_{t,j}\Delta m_{t+1,j}m_{t,k}^2}{\sigma^3}-\frac{m_{t,j}^2m_{t,k}^2}{\sigma^4}B_{t,j}\right)\sigma^{-4}\left(\sum_{j=1}^{n}m_{t,j}^4\right)^{-1}\\ &= \left(-\sum_{j=1}^{n}\sigma m_{t,j}\Delta m_{t+1,j}m_{t,k}^2-m_{t,j}^2m_{t,k}^2B_{t,j}\right)\left(\sum_{j=1}^{n}m_{t,j}^4\right)^{-1}.\end{aligned}$$

We observe that, the updates to B can become very small when the absolute value of the gradient is less than 1. In neural network training, especially in deeper architectures, the gradient is often significantly smaller than 1. This phenomenon is related to the *vanishing gradient* problem, where gradients diminish due to repeated multiplications during backpropagation.

Although we divide by $\left(\sum_{j=1}^{n}m_{t,j}^4\right)$, which is also a small value in such cases, the application of ϵ typically occurs outside the parentheses for numerical stability reasons. This leads to very small updates to B. Due to the stochastic nature of the gradient, where the sign of the updates can flip from batch to batch, B is unable to accumulate large values. Therefore, the values of B remain small and are subsequently clamped to σ before gradient preconditioning, causing the loss of all curvature information in the process.

As mentioned earlier, there are also multiple issues in the implementation of *Apollo* (see 3.4), which deviate from its theoretical formulation. In line **5**, we see the implementation of Δm_{t+1}. However, `delta_grad` is not the true difference between the current and previous gradient, but rather the difference between the current gradient and `exp_avg_grad`, where `exp_avg_grad` is not the EMA of the gradient, but a recursive form of $\texttt{exp\_avg\_grad}_{t+1} = (1-\beta)(\texttt{grad}-\texttt{exp\_avg\_grad}_t)$ (see line **9**). Additionally, this value is used as a replacement for the EMA of the gradient in line **19**, where it is preconditioned using the Hessian diagonal approximation. Furthermore, the bias correction in line **9** does not follow the standard approach for updating an EMA (see Section 1.3.6). We suspect that the authors may have incorrectly updated `exp_avg_grad` with `delta_grad` instead of `grad`, as doing so would align with the theoretical formulation, aside from the incorrect bias correction.

```
(...)
bias_correction = 1 - beta ** state['step']
alpha = (1 - beta) / bias_correction
# calc the diff grad
delta_grad = grad - exp_avg_grad
rebound = 0.01
eps = eps / rebound
# Update the running average grad
exp_avg_grad.add_(delta_grad, alpha=alpha)
denom = d_p.norm(p=4).add(eps)
d_p.div_(denom)
v_sq = d_p.mul(d_p)
delta = delta_grad.div_(denom).mul_(d_p).sum().mul(-alpha)
- B.mul(v_sq).sum()
# Update B
B.addcmul_(v_sq, delta)
# calc direction of parameter updates
denom = B.abs().clamp_(min=rebound)
d_p.copy_(exp_avg_grad.div(denom))
# Perform step weight decay
(...)
p.add_(d_p, alpha=-curr_lr)
```

Figure 3.4: The implementation of *Apollo* in PyTorch

To address the identified issues, we propose a modified version of the Apollo optimizer, named *SApollo (Smoothed Apollo)*, which adheres to its theoretical formulation. Additionally, we incorporate an Exponential Moving Average (EMA) of the Hessian diagonal approximations, denoted as B, as suggested by the authors of *Apollo* as an interesting direction for future research [25]. Figure **??** illustrates our implementation. We calculate `exp_avg_grad` as a properly implemented EMA of the gradient and use it in the preconditioning step. Furthermore, we calculate `delta_grad`, by capturing the difference between the current gradient and it's EMA, similar to AdaBelief (see 1.3.7). The Hessian diagonal approximation B is implemented as a moving average of the updates derived from Eq. 1.49. Note that this does not require additional memory, as we simply replace the non-EMA version with the EMA version of B. As we identified the clamping operation to be problematic for small values, we instead add σ to the absolute values of B (line **21**) if they are smaller than σ. Although this can lead to a learning rate decrease of at most twice the amount of *Apollo*, it is able to capture important curvature information. To tune σ in practice, *SApollo* introduces σ as a hyperparameter, contrary to Apollo, where σ is hardcoded in the optimizer. As we observed that the updates of B can become very small, a high σ may therefore lead to poor Hessian approximations. In our performance evaluation, we use $\sigma = 0.0001$ (lr=0.001), while for the task of Hessian approximation, we use $\sigma = 0.01$ for optimal comparability with *Apollo*. It still needs to be determined how low the value of σ should optimally be in practice.

As this might vary across different model architectures, we believe introducing σ as a hyperparameter is a good choice. Figure **??** shows the results of the Hessian approximation. Note that these results only show the approximation quality regarding the *weights*, for the *biases* we refer to **??**. While *SApollo* is not yet competitive with first-order methods, it is on par or performs slightly better than *Apollo* especially in later layers. As said earlier, we hypothesize that decreasing the value of σ could further improve the quality of the Hessian approximation, as $\sigma = 0.01$ may be too large, making it an interesting direction for future research. In Figure **??**, we present the performance comparison between *SApollo* and *Apollo(W)*. The figure illustrates that *SApollo* decreases the loss much faster than both *ApolloW* and *Apollo*. Although *SApollo* performs better with $\sigma = 0.0001$ on this dataset, it remains to be determined whether this improvement can solely be attributed to the correction of the mentioned implementation mistakes, or if *Apollo(W)* would perform on par with or better than *SApollo* when its σ is set to similar values. Additionally, since `d_p` solely relies on values that are already stored, one could consider recomputing it at the start of each iteration using the not yet updated values of `exp_avg_grad` and B. This would reduce *(S)Apollo*'s memory requirements to those of popular first-order methods, such as *Adam*.

```
    (...)
    bias_correction1 = 1 - beta1 ** state['step']
    bias_correction2 = 1 - beta2 ** state['step']

    # calc the diff grad
    delta_grad = grad - (exp_avg_grad/bias_correction1)
    sigma = 0.01
    # Update the running average grad
    exp_avg_grad.mul_(beta1).add_(grad, alpha=1 - beta1)

    denom = d_p.norm(p=4).add(eps)
    d_p.div_(denom)
    v_sq = d_p.mul(d_p)
    B_hat = B / bias_correction2
    delta = delta_grad.div_(denom).mul_(d_p).sum().mul(-1)
    - B_hat.mul(v_sq).sum()
    # Update B
    B.mul_(beta2).addcmul_(v_sq, delta, value=1 - beta2)
    B_hat = B / bias_correction2
    # calc direction of parameter updates
    denom = B_hat.abs()
    denom = torch.where(denom,denom < sigma,denom.add_(sigma),denom)
    d_p.copy_((exp_avg_grad/bias_correction1).div(denom))
    # Perform step weight decay
    (...)
    p.add_(d_p, alpha=-curr_lr)
```

Figure 3.5: The implementation of *SApollo* in PyTorch

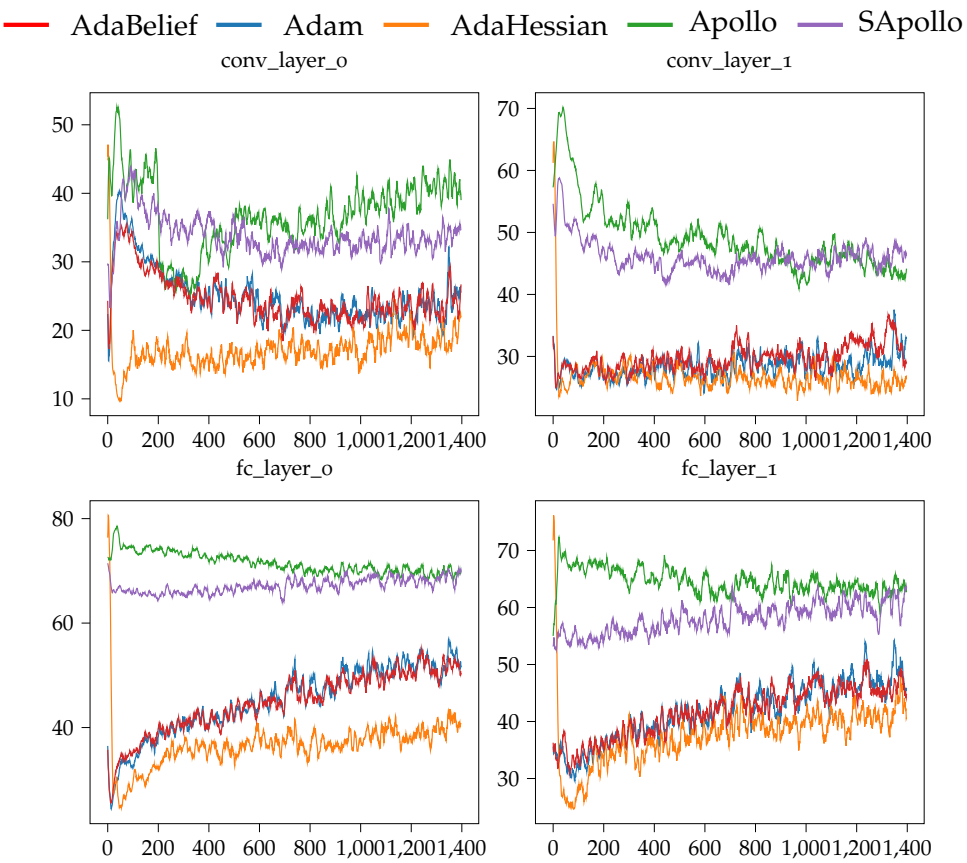

Figure 3.6: The cosine similarity (in degrees), y-axis, between the calculated batch Hessian diagonal and the corresponding optimizer approximations on a *small* batch (124 samples). Optimizer updates are denoted on the x-axis.

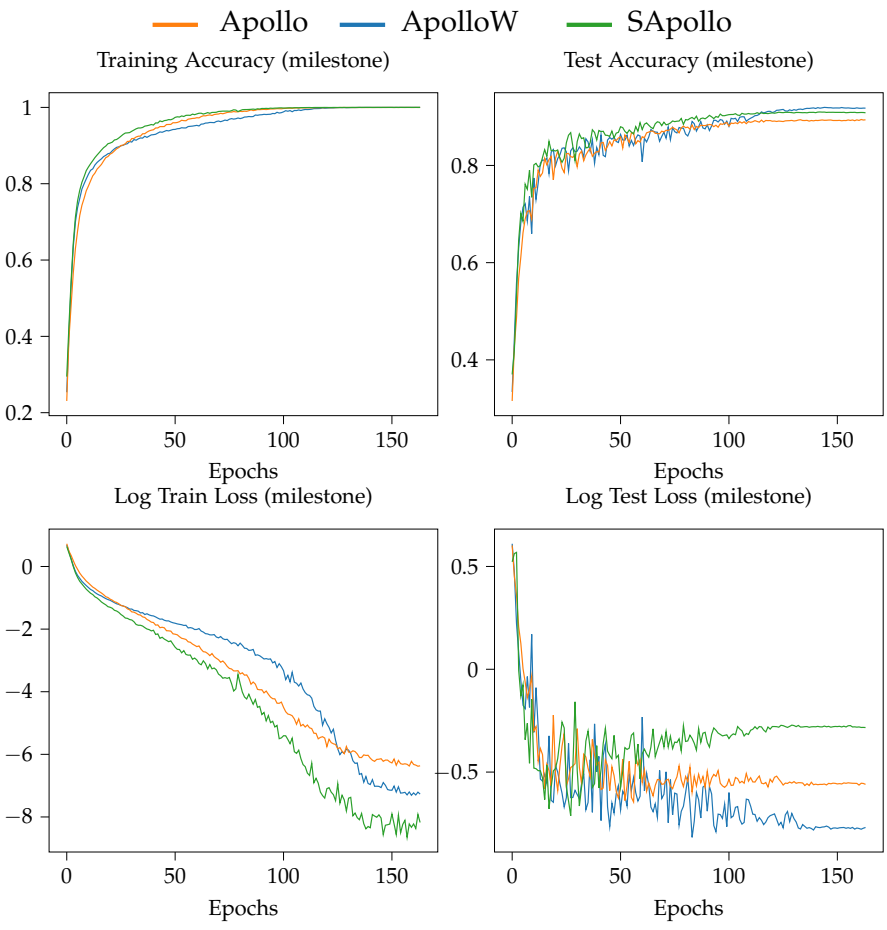

Figure 3.7: Evaluation of *SApollo* on CIFAR-10 using ResNet-110 with the *cosine annealing* learning rate scheduler.

4

CONCLUSION AND FUTURE DIRECTIONS

In this work, we examined *Apollo* and *AdaHessian*, two recent advancements in the field of second-order optimization for neural network training. We evaluated both optimizers in terms of performance on common datasets and resource consumption, compared to first-order methods. While *AdaHessian* demonstrated strong performance in approximating the batch Hessian diagonal, its significant resource requirements and the need for derivative graph creation in PyTorch make it impractical for non-academic use. On the other hand, *Apollo*, though able to perform similarly to *AdaHessian* in our experiments, exhibited a significant limitation in its Hessian approximation due to its clamping operation, which loses curvature information when values fall below σ. Furthermore, we identified several implementation flaws in *Apollo*, which we addressed in our modified version, *SApollo*. In addition to fixing these issues, *SApollo* introduces a smoothed version of the Hessian approximation, B, to reduce the influence of stochastic noise. Instead of clamping values below σ, *SApollo* adds σ to approximations that fall below this threshold, ensuring that small approximation information is retained. This approach led to improved Hessian approximation accuracy and better convergence performance on the *CIFAR-10* dataset. We hypothesize that using smaller values for σ, in the range of $[10^{-3}, 10^{-4}, 10^{-5}]$, could lead to significantly better performance. The value $\sigma = 0.01$, which Apollo uses, is relatively large when compared to the typical magnitude of gradients, which are often much smaller, especially when training deep networks like *ResNet*. By introducing σ as a tunable hyperparameter in *SApollo*, fine-tuning it could be an interesting direction for future research.

A

APPENDIX

A.1 APPENDIX

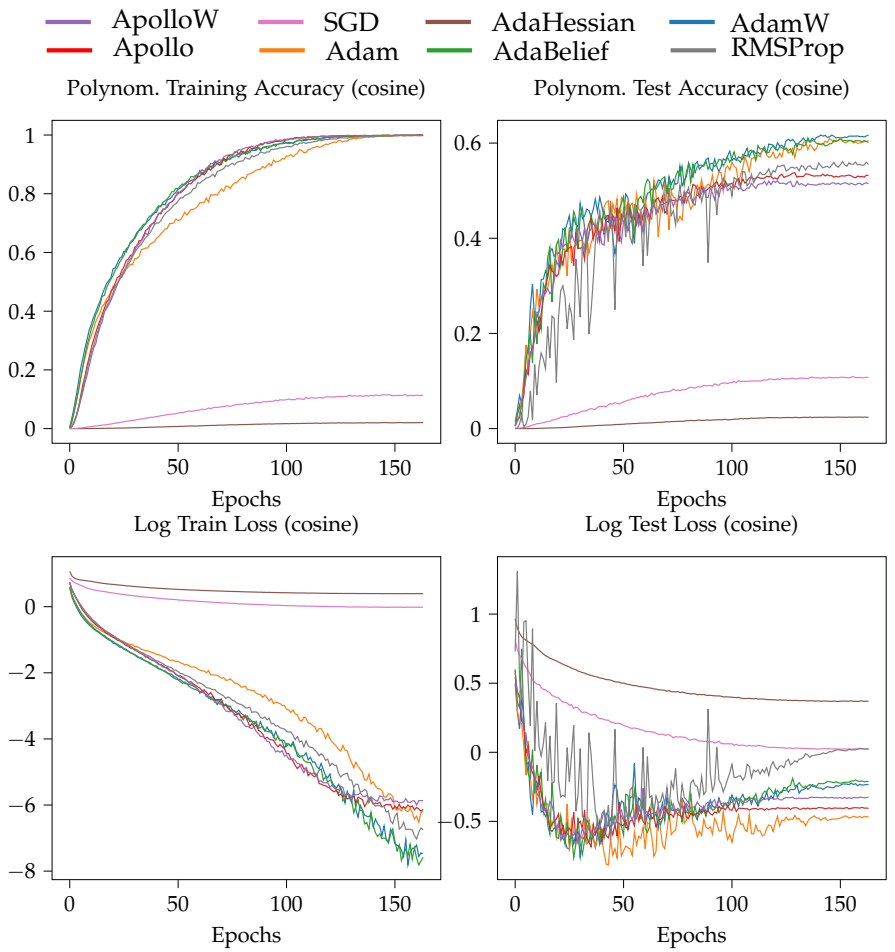

Figure A.1: Evaluation of optimizers on CIFAR-10 using ResNet-110 with the *cosine annealing* learning rate scheduler, where hyperparameters are held constant across all optimizers.For better visualization we applied a polynomial transformation, with $\hat{x} = x^{\alpha}$ and $\alpha = 5$, for every $x \in \mathcal{D}$ in the output data $\mathcal{D}$.

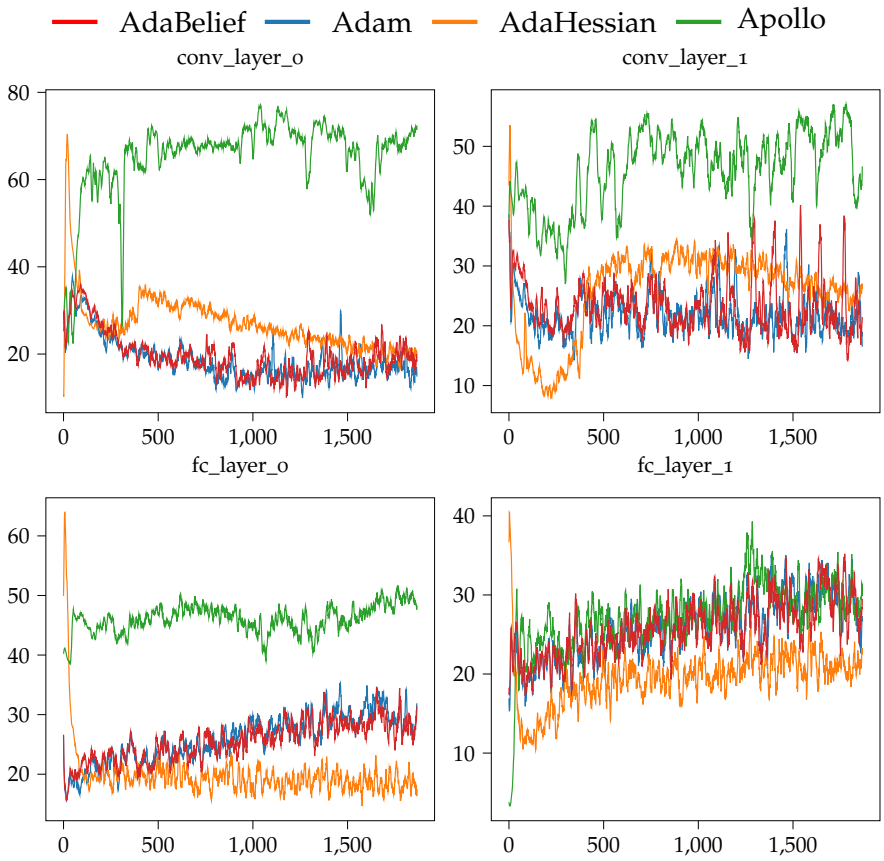

Figure A.2: The cosine similarity (in degrees), y-axis, between the calculated batch Hessian diagonal and the corresponding optimizer approximations on a *small* batch (124 samples). Optimizer updates are denoted on the x-axis. Note that these results represent only the Hessian diagonals for the network's *biases*.

Table A.1: Hyperparameter settings for CIFAR-10. Values in parentheses indicate configurations used for individual best-case evaluations.

	Learning rate	**Weight decay**	**Beta**	**Epsilon**	**Warmup**
SGD	1×10^{-3}	2.5×10^{-4}	(0.9)	-	0
Adam	1×10^{-3}	2.5×10^{-4}	(0.9,0.999)	1×10^{-08}	0
AdamW	1×10^{-3}	2.5×10^{-4} **(0.025)**	(0.9,0.999)	1×10^{-08}	0
AdaBelief	1×10^{-3}	2.5×10^{-4} **(0.025)**	(0.9,0.999)	1×10^{-08}	0
RMSProp	1×10^{-3}	2.5×10^{-4}	(0.99)	1×10^{-08}	0
Apollo	1×10^{-3} **(0.01)**	2.5×10^{-4}	(0.9)	1×10^{-4}	0 **(500)**
ApolloW	1×10^{-3} **(0.01)**	2.5×10^{-4} **(0.025)**	(0.9)	1×10^{-4}	0 **(500)**
AdaHessian	1×10^{-3} **(0.15)**	2.5×10^{-4} **(0.001)**	(0.9,0.999)	1×10^{-4}	0 **(500)**

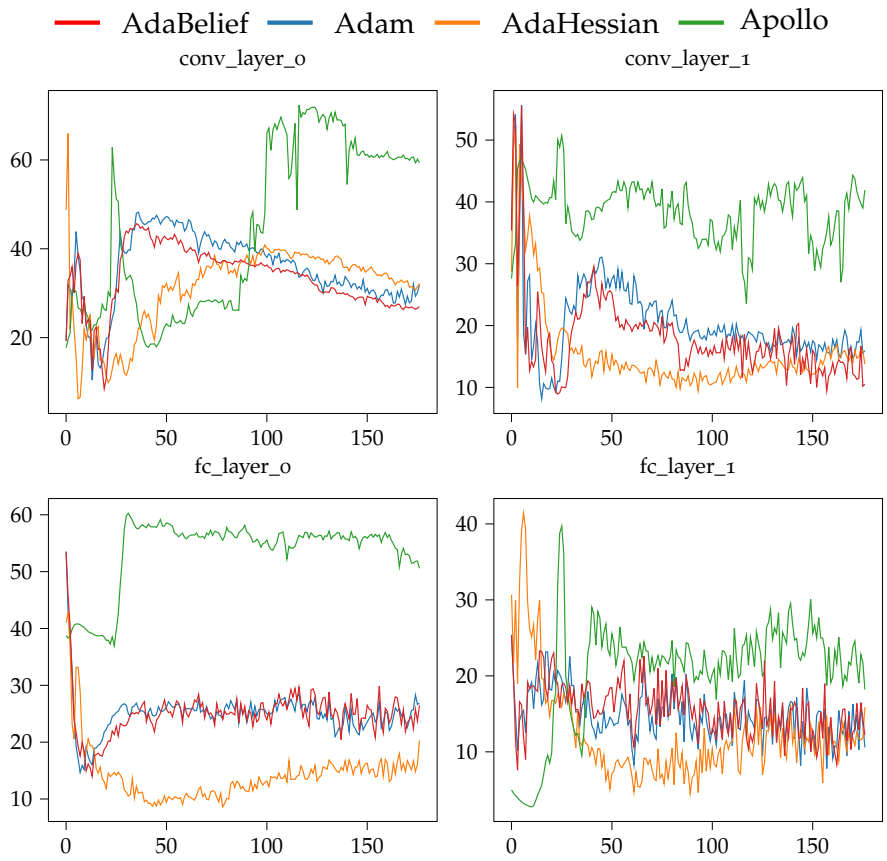

Figure A.3: The cosine similarity (in degrees), y-axis, between the calculated batch Hessian diagonal and the corresponding optimizer approximations on a *small* batch (124 samples). Optimizer updates are denoted on the x-axis. Note that these results represent only the Hessian diagonals for the network's *biases*.

Table A.2: Hyperparameter settings for TinyImageNet. Values in parentheses indicate configurations used for individual best-case evaluations.

	Learning rate	**Weight decay**	**Beta**	**Epsilon**	**Warmup**
SGD	1×10^{-3}	1×10^{-4}	(0.9)	-	0
Adam	1×10^{-3}	1×10^{-4}	(0.9,0.999)	1×10^{-08}	0
AdamW	1×10^{-3}	1×10^{-4} **(0.01)**	(0.9,0.999)	1×10^{-08}	0
AdaBelief	1×10^{-3}	1×10^{-4} **(0.01)**	(0.9,0.999)	1×10^{-08}	0
RMSProp	1×10^{-3}	1×10^{-4}	(0.99)	1×10^{-08}	0
Apollo	1×10^{-3} **(0.01)**	1×10^{-4}	(0.9)	1×10^{-4}	0 **(500)**
ApolloW	1×10^{-3} **(0.01)**	1×10^{-4} **(0.01)**	(0.9)	1×10^{-4}	0 **(500)**
AdaHessian	1×10^{-3} **(0.15)**	1×10^{-4} **(0.001)**	(0.9,0.999)	1×10^{-4}	0 **(500)**

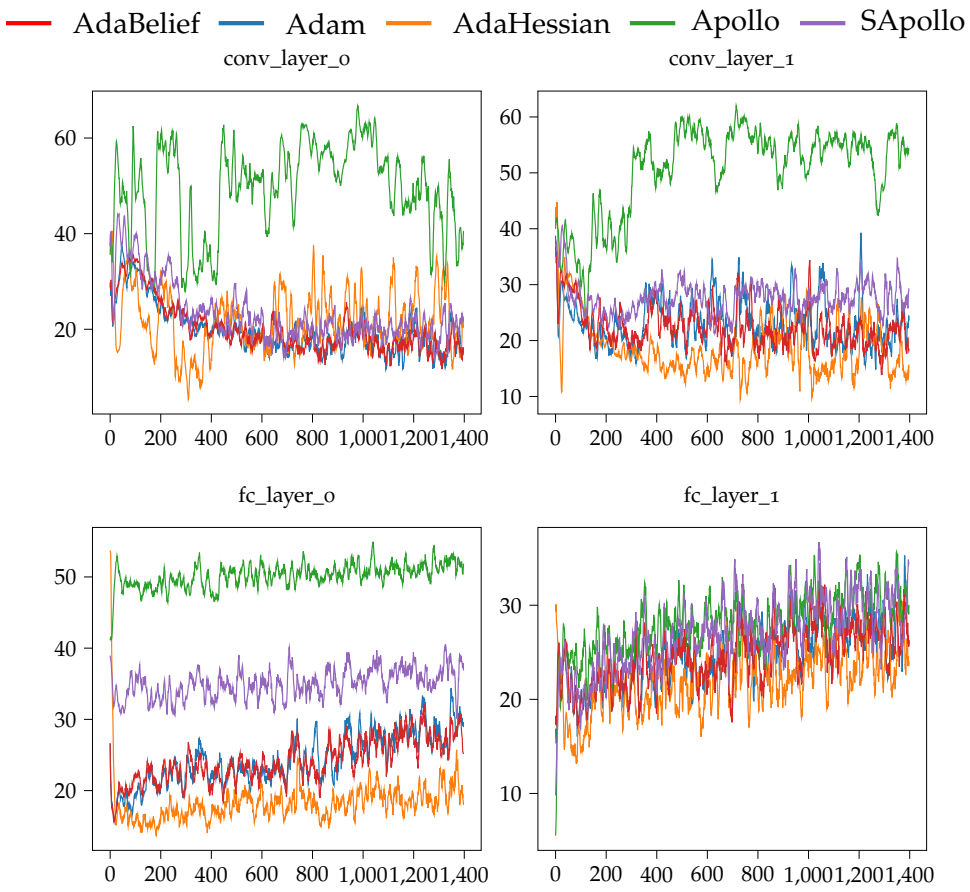

Figure A.4: The cosine similarity (in degrees), y-axis, between the calculated batch Hessian diagonal and the corresponding optimizer approximations on a *small* batch (124 samples). Optimizer updates are denoted on the x-axis. Note that these results represent only the Hessian diagonals for the network's *biases*.

Table A.3: Hyperparameter settings for WMT-14.

	Learning rate	**Weight decay**	**Beta**	**Epsilon**	**Warmup**
SGD	1×10^{-3}	1×10^{-4}	(0.9)	-	0
Adam	1×10^{-3}	0	(0.9,0.98)	1×10^{-09}	4000
AdamW	1×10^{-3}	1×10^{-4}	(0.9,0.98)	1×10^{-09}	4000
AdaBelief	1×10^{-3}	1×10^{-4}	(0.9,0.98)	1×10^{-09}	4000
RMSProp	-	-	-	-	-
Apollo	0.04	0	(0.9)	1×10^{-4}	4000
ApolloW	0.04	1×10^{-8}	(0.9)	1×10^{-4}	4000
AdaHessian	0.1	0	(0.9,0.98)	1×10^{-4}	4000

Table A.4: Hyperparameter settings for curvature approximation quality. Values in parentheses are used in the *SApollo* comparison

	Learning rate	**Weight decay**	**Beta**	**Epsilon**
Adam	1×10^{-2}	0	(0.9,0.999)	1×10^{-08}
AdaBelief	1×10^{-2}	0	(0.9,0.999)	1×10^{-08}
Apollo	3×10^{-3} (0.01)	0	(0.9)	1×10^{-4}
SApollo	0.01	0	(0.9)	1×10^{-4}
AdaHessian	2×10^{-1}	0	(0.9,0.999)	1×10^{-4}

BIBLIOGRAPHY

[1] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. *Scalable Second Order Optimization for Deep Learning*. 2021. arXiv: 2002.09018 [cs.LG].

[2] Tilo Arens, Frank Hettlich, Christian Karpfinger, Ulrich Kockelkorn, Klaus Lichtenegger, and Hellmuth Stachel. *Mathematik*. 2nd ed. Heidelberg: Spektrum Akademischer Verlag, 2011, p. 1506. ISBN: 978-3-8274-2347-4.

[3] David Bachman. *Advanced Calculus Demystified*. McGraw-Hill Education, 2007. ISBN: 978-0-07-147217-3.

[4] C. Bekas, E. Kokiopoulou, and Y. Saad. "An estimator for the diagonal of a matrix." In: *Applied Numerical Mathematics* 57.11 (2007). Numerical Algorithms, Parallelism and Applications (2), pp. 1214–1229. ISSN: 0168-9274. DOI: https://doi.org/10.1016/j.apnum.2007.01.003. URL: https://www.sciencedirect.com/science/article/pii/S0168927407000244.

[5] Christopher M. Bishop. *Exact Calculation of the Hessian Matrix for the Multilayer Perceptron*. Tech. rep. Accessed: 2024-05-27. Neural Computing Research Group, Aston University, 1992. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/bishop-hessian-nc-92.pdf.

[6] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[7] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. *Optimization Methods for Large-Scale Machine Learning*. 2018. arXiv: 1606.04838 [stat.ML].

[8] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004. ISBN: 9780521833783.

[9] Charles George Broyden. "Quasi-Newton methods and their application to function minimization." In: *Mathematics of Computation* 21.99 (1967), pp. 368–381.

[10] Charles George Broyden. "The convergence of a class of double-rank minimization algorithms 1. general considerations." In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90.

[11] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. *Understanding the Difficulty of Training Transformers*. 2020. arXiv: 2004.08249 [cs.LG]. URL: https://ar5iv.labs.arxiv.org/html/2004.08249.

[12] Dive into Deep Learning. *Adam*. Accessed: 2024-06-10. D2L.ai, 2023.

[13] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. German. 5th ed. Springer Vieweg, 2021. ISBN: 978-3-658-33723-5.

[14] Roger Fletcher and Michael JD Powell. "A rapidly convergent descent method for minimization." In: *The Computer Journal* 6.2 (1963), pp. 163–168.

[15] Guillaume Garrigos and Robert M. Gower. *Handbook of Convergence Theorems for (Stochastic) Gradient Methods*. 2024. arXiv: 2301.11235 [math.OC].

[16] GeoffreyHinton. *Neural Networks for Machine Learning*. Tech. rep. Accessed: 2024-06-04. Neural Computing Research Group, Aston University, 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

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

[18] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. *Deep Networks with Stochastic Depth*. https://www.researchgate.net/figure/Comparison-of-ResNets-with-110-and-1202-layers-When-trained-with-stochastic-depth-the_fig4_301879329. [Figure] Accessed 16 Jul 2024. 2016.

[19] Michael F Hutchinson. "A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines." In: *Communications in Statistics-Simulation and Computation* 19.2 (1990), pp. 433–450. DOI: 10.1080/03610919008812866.

[20] Ph.D. Jacob Murel and Eda Kavlakoglu. "What Is Regularization?" In: (2023). Accessed: 2024-05-27. URL: https://www.ibm.com/topics/regularization.

[21] Rohan Kashyap. *A survey of deep learning optimizers – first and second order methods*. 2023. arXiv: 2211.15596 [cs.LG].

[22] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[23] Ilya Loshchilov and Frank Hutter. "Fixing Weight Decay Regularization in Adam." In: (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101.

[24] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101.

[25] Xuezhe Ma. *Apollo: An Adaptive Parameter-wise Diagonal Quasi-Newton Method for Nonconvex Stochastic Optimization*. 2021. arXiv: 2009.13586 [cs.LG]. URL: https://arxiv.org/abs/2009.13586.

[26] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

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

[28] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. 2nd. Springer, 2006. ISBN: 978-0-387-30303-1.

[29] Barak A. Pearlmutter. "Fast exact multiplication by the Hessian." In: *Neural Computation* 6.1 (1994), pp. 147–160. DOI: 10.1162/neco.1994.6.1.147. URL: https://www.bcl.hamilton.ie/~barak/papers/nc-hessian.pdf.

[30] Boris T. Polyak. "Some methods of speeding up the convergence of iteration methods." In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[31] PyTorch. *torch.nn.DataParallel*. https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html. Accessed: 2024-07-01. 2023.

[32] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536.

[33] Gilbert Strang. *Introduction to linear algebra*. SIAM, 2022.

[34] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. *A Survey of Optimization Methods from a Machine Learning Perspective*. 2019. arXiv: 1906.06821 [cs.LG].

[35] Fred E. Szabo. "S." In: *The Linear Algebra Survival Guide*. Ed. by Fred E. Szabo. Boston: Academic Press, 2015, pp. 320–377. ISBN: 978-0-12-409520-5. DOI: https://doi.org/10.1016/B978-0-12-409520-5.50026-6. URL: https://www.sciencedirect.com/science/article/pii/B9780124095205500266.

[36] Jake Tae. *Fisher Information and Its Applications*. Accessed: 2024-09-13. 2021. URL: https://jaketae.github.io/study/fisher/.

[37] *Tiny ImageNet Dataset*. Accessed: 2024-07-06. 2023. URL: https://paperswithcode.com/dataset/tiny-imagenet.

[38] Marc Toussaint. *Gradient Descent - Algorithm, History, and Applications*. https://www.user.tu-berlin.de/mtoussai/notes/gradientDescent.pdf. Accessed: 2024-05-08. 2014.

[39] Stanford University. *Optimization: Stochastic Gradient Descent*. http://deeplearning.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/. Accessed: 2024-05-08.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[41] X. Wang. *Optimization Basics*. Lecture slides. Accessed: [Insert access date here]. 2021. URL: https://www.stat.purdue.edu/~wang4094/resources/slides/2021_spring_DL_meeting_01_opt_basics.pdf.

[42] Xiaoxia Wu, Rachel Ward, and Léon Bottou. *WNGrad: Learn the Learning Rate in Gradient Descent*. 2020. arXiv: 1803.02865 [stat.ML].

[43] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W. Mahoney. *ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning*. 2021. arXiv: 2006.00719 [cs.LG].

[44] M. Zhu, John Lawrence Nazareth, and Henry Wolkowicz. "The Quasi-Cauchy Relation and Diagonal Updating." In: *SIAM J. Optim.* 9 (1999), pp. 1192–1204. URL: https://api.semanticscholar.org/CorpusID:949459.

[45] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S. Duncan. *AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients*. 2020. arXiv: 2010.07468 [cs.LG]. URL: https://arxiv.org/abs/2010.07468.

[46] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S. Duncan. "AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020. URL: https://juntang-zhuang.github.io/adabelief/.

DECLARATION

I hereby certify that I, have independently authored the above-mentioned work and have not used any sources or aids other than those indicated, and have properly marked all quotations. I have used a large language model (LLM) solely for purposes of formulation and grammar

Bonn, October 2024

Jan Niclas Hardtke