Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät
B.Sc. Kognitionswissenschaft

Curvature-Based Step Sizes for Training Deep Neural Networks

Bachelorarbeit

von

David Suckrow

geb. am 06. 06. 1999 in Stuttgart

Matrikel-Nr. 5764570

Gutachter: Prof. Dr. Philipp Hennig

Bearbeitungszeitraum: 21.11.2024 – 25.03.2025

Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe. Eine (Selbständigkeits-)Erklärung zum Einsatz von (generativer) KI befindet sich in Appendix A.

D. Suckrow

. . . .

David Suckrow
Tübingen, den 24. 03. 2025

Danksagung

Mein besonderer Dank gilt meinem Betreuer Lukas Tatzel, der mich während dieser Arbeit durchgehend sehr eng begleitet hat. Er stand mir jederzeit für Fragen zur Verfügung, gab konstruktives Feedback und sprach mir immer wieder Mut zu. Ich danke auch Prof. Dr. Philipp Hennig für die allgemeine fachliche Orientierung und hilfreichen Anregungen, die zum gelingen dieser Arbeit beigetragen haben. Runa Eschenhagen und Dr. Frank Schneider danke ich für ihre stets schnelle und hilfreiche Unterstützung bei fachlichen Fragen – das war für mich eine große Hilfe. Franziska Weiler unterstützte mich häufig bei administrativen Aufgaben und sorgte damit dafür, dass ich mich auf das Wesentliche konzentrieren konnte.

Abstract

Deep neural networks have proven highly effective across a wide range of tasks and domains. However, training them remains a sensitive and complex process, with performance depending heavily on various optimization algorithm design choices, including the step size. As a result, finding a well-performing configuration can be computationally expensive. While first-order optimization algorithms—based on gradient information—remain the standard, the use of second-order curvature information has recently become more practical and accessible.

In this thesis, we explore a curvature-based step size strategy designed to complement existing first-order optimizers. We evaluate (1) the applicability of this step size method, including a debiasing strategy, by comparing it to a conventional learning rate schedule, and (2) its performance across different variations of the method. These evaluations are conducted using a state-of-the-art deep neural network, dataset, and benchmarked against leading optimization algorithms. Our results show that (1) the debiased step size yields smaller and therefore more applicable step sizes compared to the naïve variant, but exhibits high variability. To address this, we propose a modified strategy that stabilizes the step size. Furthermore, (2) we find that our method performs more competitively when expensive curvature information is computed less frequently, although it still falls short of our strong comparison baselines.

Contents

1 Introduction **8**
  1.1 Motivation and Context 8
  1.2 Problem Statement 8
  1.3 Research Questions 9

2 Background **10**
  2.1 Fundamentals of Neural Networks 10
  2.2 Deep Neural Networks 12
    2.2.1 Deep Neural Network Optimization 12
    2.2.2 First Order Optimization Algorithms 12
  2.3 Second Order Optimization Algorithms 14
    2.3.1 Newton’s Method 14
    2.3.2 Quasi-Newton Methods 15
    2.3.3 Approximate Curvature Matrices 15
    2.3.4 Hessian-Free Optimization 18
    2.3.5 Full Second Order Methods in Training Deep Neural Networks . 19
  2.4 Step Size Selection 19
  2.5 Bridging the Gap: Efficient Step Size Estimation 21
  2.6 Related Work 22

3 Materials and Methods **23**
  3.1 Overview 23
  3.2 Step Size Derivation 23
  3.3 Evaluation Framework, Dataset, Model and Optimizer 25
    3.3.1 Evaluation Framework 25
    3.3.2 Criteo 1TB 26
    3.3.3 DLRMsmall Model 27
    3.3.4 Optimizer 27
  3.4 Algorithm and Implementation 28
  3.5 Comparison Baselines 29

4 Experiments, Results and Discussion **30**
  4.1 Pre-Test and Sanity Check 30
    4.1.1 Sanity Check 30
    4.1.2 Pre-Test 31
  4.2 Experiment 1: Observing Alpha 31
  4.3 Results: Experiment 1 34
    4.3.1 Alpha Values 34
    4.3.2 Numerator and Denominator 36
    4.3.3 Performance Metrics 37
  4.4 Discussion: Experiment 1 38
    4.4.1 Extreme Outlier in the First Step 38
    4.4.2 Relative Learning Rate Magnitudes 38

4.4.3 Absolute Magnitudes and Dispersion 38
4.4.4 Trend in Absolute Biased Alpha 39
4.4.5 Numerator and Denominator Contribution 40
4.4.6 Performance 41
4.5 Experiment 2: Grid Search 42
4.6 Results: Experiment 2 43
4.6.1 Grid Search: Step-Based Comparison 43
4.6.2 Grid Search: Time-Based Comparison 46
4.7 Discussion: Experiment 2 50
4.7.1 Step-Based Comparison: Baselines vs. Grid Search 50
4.7.2 Step-Based Comparison: Grid Search 50
4.7.3 Time-Based Comparison: Baselines vs. Grid Search 52
4.7.4 Time-Based Comparison: Grid Search 52
4.7.5 Limitations 53

5 Conclusion 55
5.1 Overall Conclusion 55
5.2 Research Questions 55
5.3 Future Work 56

Appendix 63

List of Figures and Tables

List of Figures

Figure 2.1: Basic Structure of a Feedforward Neural Network. 10
Figure 2.2: Computations of a Single Neuron in a Feedforward Neural Network. 11
Figure 2.3: Optimization Algorithm using Adaptive Moment Estimation (Adam). 13

Figure 3.1: Algorithm for Observing or Applying α^*. 28

Figure 4.1: Original and Adapted Learning Rate Schedule 32
Figure 4.2: Algorithmic Structure of Experiment 1. 33
Figure 4.3: Averages of Relative Alpha Biased and Unbiased 34
Figure 4.4: Absolute Alpha Unbiased: Mean, Median and Standard deviation . 35
Figure 4.5: Absolute Alpha Unbiased: Mean, Median and Standard deviation . 36
Figure 4.6: Comparison of Absolute Alpha (Biased vs. Unbiased) 37
Figure 4.7: Losses on Training, Test and Validation Split 37
Figure 4.8: General Algorithm of Experiment 2. 43
Figure 4.9: Best Validation Split Loss: Full Schedule 44
Figure 4.10: Applied Learning Rates and LR Schedule 45
Figure 4.11: Best Validation Split Loss: Time of Full Schedule 46
Figure 4.12: Best Validation Split Loss within 61,624 seconds 47
Figure 4.13: Learning Rates of Best and Worst on 61,624 Seconds Budget. . . . 48
Figure 4.14: Learning Rates of Best and Worst on 61,624 Seconds Budget. . . . 49

List of Tables

Table 4.1: Average Computation Times per Step. 31
Table 4.2: Grid-Search Parameters. 42
Table 4.3: Ranges of Applied Learning Rates 46

1 Introduction

1.1 Motivation and Context

Neural networks have become fundamental tools in modern machine learning due to their ability to learn complex patterns from large datasets. Specifically, deep neural networks (DNNs), which are large networks often consisting of millions to billions of parameters, have significantly advanced the state of the art in domains such as computer vision (Deng et al., 2009), natural language processing, and many more (LeCun et al., 2015). The success of DNNs is built on many foundations, with some of the most important being well-chosen network architectures, high-quality and sufficient data, adequate computational power, and efficient training algorithms.

Training algorithms iteratively adapt the parameters of a network to given training data to improve its performance. This corresponds to navigating a complex and high-dimensional loss landscape to find sufficient parameters aiming to reduce the loss or error. A central aspect of this optimization process is the learning rate, which determines the step size of a parameter update during training. Due to its significant impact on the training process, the learning rate is central in neural network optimization. Too small learning rates can lead to slow convergence or suboptimal solutions, while too large learning rates can cause instability and divergence (LeCun et al., 2015). Conversely, well-chosen learning rates can save time and resources and achieve better results.

1.2 Problem Statement

The majority of current optimization methods used in deep learning primarily rely on first-order gradient information. Adaptive training methods such as Adam (Kingma & Ba, 2015) adjust individual parameter step sizes based on first-order gradient magnitudes but still typically require careful tuning of the learning rate. This incorporates the slope of the loss function but neglects the curvature. Therefore, finding efficient learning rates remains a challenge in many cases. Second-order optimization methods, which consider curvature information in the training algorithm, often offer alternatives theoretically superior to first-order methods (Agarwal et al., 2017). These methods are known for their potential to accelerate convergence compared to first-order methods. However, high computational and memory costs of second-order methods have limited their efficiency. Several second-order optimization methods that aim to mitigate this issue have been proposed, but widespread adoption has been limited. This research addresses the gap between first-order adaptive methods and computationally expensive second-order approaches by investigating a curvature-based step size adjustment strategy that aims to balance efficiency and performance.

1.3 Research Questions

To achieve these objectives, this thesis seeks to answer the following questions:

- Does the proposed step size offer magnitudes that can be used for training deep neural networks, and, if so, how do stochasticity and inherent biases need to be considered when leveraging this curvature-based step size?
- Can we implement this method in a way that balances computational cost to achieve competitive performance against state-of-the-art optimizers and learning rate schedules?

2 Background

2.1 Fundamentals of Neural Networks

Neural networks are computational models originally inspired by the structure and function of biological neurons. Feedforward neural networks (FNNs) were among the first artificial neural networks developed and are the most basic type. They take inputs, transform them in a linear or non-linear fashion, and produce outputs. They consist of layers of artificial neurons which are based on the 'perceptron' originally proposed by Rosenblatt (1958). The layers are organized, where the first layer ('input layer') takes the input of the model and the last layer ('output layer') produces the output of the model. Generally, each layer besides the last layer is connected to the following layer by passing information on to it.

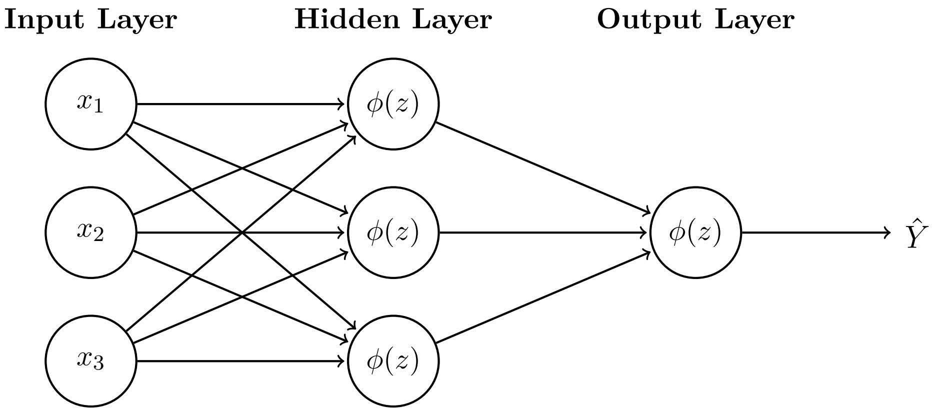


Figure 2.1: Basic Structure of a Feedforward Neural Network. A simple feedforward neural network with three input neurons, three hidden neurons, and one output neuron. Each layer is fully connected to the next. $\phi(z)$ represents the activation function applied to the weighted sum of inputs, $\hat{Y}$ denotes the output of the network.

Each neuron in a FNN operates by taking inputs, processing them, and producing an output. It receives input values $x_1, \ldots, x_n$, each associated with a weight $w_1, \ldots, w_n$. The neuron calculates the weighted sum of these inputs, adding a bias term b to the result (see figure 2.2). This sum is then passed into an activation function (e.g., ReLU, sigmoid, tanh), which often introduces a non-linearity to the neuron's behavior. The result of this operation becomes the neuron's output, which can either serve as the input to the next layer of neurons or as the final output of the model.
FNNs are trained through a process that enables them to adapt to data by adjusting their parameters (weights and biases) to minimize a predefined error or loss function. A single data point typically consists of an input vector X, which contains the features

fed into the neural network, and a target Y, which represents the true outcome or label. For example, the input vector could contain different features like temperature, humidity, and wind speed, while the target could represent the weather condition, such as “rainy,” “sunny,” or “cloudy.” The training process of a neural network involves initializing the model's parameters randomly and then passing input vectors through the model (forward propagation), calculating the network's predictions $\hat{Y}$.

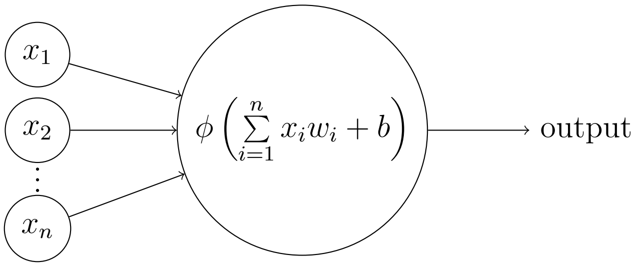


Figure 2.2: Computations of a Single Neuron in a Feedforward Neural Network.
Each neuron in a feedforward neural network computes a weighted sum of its inputs $x_1, x_2, \ldots, x_n$, applies a bias b, and then passes the result through an activation function $\phi(z)$. The output of the neuron is then passed to the next layer or used as the final output of the model. This figure illustrates a single neuron receiving multiple inputs and computing its activation.

These predictions are then compared to the target output using a loss function. A loss function is a mathematical function that measures the difference or distance between a predicted output of a neural network and the actual target or true value. Once the loss is computed, the network leverages backpropagation, centrally introduced to neural network training by Rumelhart et al. (1986), to compute the gradient of the loss function with respect to each parameter. This gradient indicates how much the loss would change if a particular weight or bias were adjusted.

After backpropagation, the weights and biases are updated using an optimization algorithm like gradient descent. With gradient descent, the network adjusts its parameters in the opposite direction of the gradients to minimize the loss. The step size (how much each weight is adjusted) is determined by a factor called the *learning rate*. Due to large datasets, the training process is often performed in *mini-batches*, where the model is updated based on a random subset of the data. A mini-batch contains more than one but fewer than all the data points in the dataset. This introduces stochasticity into the training process, which can help the model generalize better to unseen data but also has further implications regarding optimization. There are many different optimization algorithms designed for this process, such as Stochastic Gradient Descent (SGD) which performs gradient descent on mini-batches of data, and many variations of it. This process of forward propagation, loss calculation, backpropagation, and weight update is repeated across multiple iterations in neural network training.

In practice, the dataset is split into three sets before training. To monitor the training progress, the model's performance can be repeatedly evaluated on the *training split*, which consists of the training data, and on the *validation split*, which consists of data the model has not seen. Finally, after training, the *test split* (another set of data that the model has not seen) can be used to predict the model's performance on unseen data. Early neural networks, such as Rosenblatt's perceptron (1958), were initially used for classification tasks, while later models expanded their application to regression and function approximation.

2.2 Deep Neural Networks

A deep neural network (DNN) is a network that consists of an input layer, an output layer, and at least one layer in between called a *hidden layer*. These networks often include specialized layers, such as convolutional, recurrent, or attention layers, which extend beyond the basic fully connected layers of sequential models. While these layers enable the network to process data more effectively for specific tasks, the fundamental principles of training—such as optimizing a loss function using gradient-based methods—remain unchanged.

2.2.1 Deep Neural Network Optimization

Increased size and complexity of DNNs comes with optimization challenges. When training large DNNs, a process also referred to as *deep learning*, vanishing and exploding gradients are common challenges that arise during backpropagation. In deep networks, gradients can either become exceedingly small (vanishing gradients) or excessively large (exploding gradients), leading to slow learning or unstable updates, respectively (Hochreiter, 1998, Pascanu et al., 2013). Another issue is overfitting, where a model with many parameters may fit the training data too closely, failing to generalize well to new, unseen data. This is especially problematic when training datasets are small (Ying, 2019). Long training times also pose a significant challenge, as the complexity and size of models increases the demand in computational resources for network training. This is particularly true when large datasets are involved, and optimizing such networks requires careful management of computational power (Dean et al., 2012).

2.2.2 First Order Optimization Algorithms

To cope with the inherent difficulties in training DNNs, optimization algorithms have evolved over time, balancing computational efficiency and effectiveness. SGD provided the foundation for training neural networks (Bottou, 1991, LeCun et al., 1998). However, because SGD suffers from several problems including slow convergence (Goodfellow et al., 2016), multiple variants have been proposed over time.

The method of momentum (Polyak, 1964) accumulates past gradients to smooth updates, in order to accelerate learning when gradients are small or noisy (Goodfellow et al., 2016). Sutskever et al. (2013) introduced Nesterov momentum, inspired by Nesterov (1983), which improves convergence by computing gradients at a "lookahead" position. Further optimization algorithms like Adagrad (Duchi et al., 2011) and its modifications, e.g., Adam (Kingma & Ba, 2015), RMSProp (Hinton et al., 2012) Adelta (Zeiler, 2012), or Nadam (Dozat, 2016), incorporate mechanisms like adaptive per-parameter learning rates or momentum into their update rules to overcome problems like poor conditioning (when gradients behave inconsistently due to an uneven loss landscape), vanishing gradients (when gradients shrink during backpropragation), and exploding gradients (when gradients become very large). Note that while these methods adaptively change an individual per-parameter learning rate, this mechanism is separate from the overall step size which is still required in these methods.

```
Require: Learning rate $\alpha$, exponential decay rates $\beta_1, \beta_2$, small constant $\epsilon$
  Initialize $m_0 = 0$, $v_0 = 0$, $t = 0$
  while not converged do
      $t \leftarrow t + 1$
      Compute gradient $g_t = \nabla f(\theta_{t-1})$
      $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$                ▷ First moment estimate
      $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$              ▷ Second moment estimate
      $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$                        ▷ Bias correction
      $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$                        ▷ Bias correction
      $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
  end while
```

Figure 2.3: Optimization Algorithm using Adaptive Moment Estimation (Adam).

Despite the success of these first-order optimization methods, they have inherent limitations. First-order methods base their update strategy on gradient information, which can necessitate exhaustive hyperparameter searches. Hyperparameters are parameters which are set before the training process begins and are not learned from the data, such as the learning rate. Second-order optimization methods, which incorporate information about the curvature of the loss function, offer a theoretically strong alternative. By leveraging second-order derivatives, these methods can accelerate convergence and often come with convergence guarantees (Nocedal & Wright, 1999). The following section explores second-order optimization techniques, beginning with Newton's method and extending to modern approaches that aim to mitigate its computational cost.

2.3 Second Order Optimization Algorithms

In this section, we will lay out some of the foundations and advancements to second order optimization algorithms. We will regularly visit the applicability and limitations of these methods in the context of deep learning optimization.

2.3.1 Newton's Method

Second order methods incorporate the second order derivative of the loss functions landscape along with first order derivatives (gradients) to guide the search for minima or maxima. Central to this is Newton's method, pioneered by Isaac Newton and refined by Joseph Raphson, as it offers a mathematically elegant solution to accelerate the search for optimal parameters. It involves repeating the *Newton step* iteratively until a sufficient solution is found. To derive the Newton step based on Bosch et al. (2022), we consider the quadratic Taylor approximation of the expected risk[1] $\mathcal{L}_{\mathcal{P}_{\text{data}}}(\theta)$ around a point $\theta^{(k)}$ in the parameter space:

$$\mathcal{L}_{\mathcal{P}_{\text{data}}}(\theta) \approx q(\theta) := \mathcal{L}_{\mathcal{P}_{\text{data}}}(\theta^{(k)}) + (\theta - \theta^{(k)})^\top g + \frac{1}{2}(\theta - \theta^{(k)})^\top H(\theta - \theta^{(k)}),$$

where

$$g = \nabla \mathcal{L}_{\mathcal{P}_{\text{data}}}(\theta^{(k)}), \quad H = \nabla^2 \mathcal{L}_{\mathcal{P}_{\text{data}}}(\theta^{(k)}) \succ 0.$$

The next iterate is obtained by minimizing $q(\theta)$, which requires setting its gradient to zero:

$$\nabla q = g + H(\theta - \theta^{(k)}) \overset{!}{=} 0.$$

Since H is invertible, we solve for θ:

$$H(\theta - \theta^{(k)}) = -g.$$

Rearranging, we obtain the Newton step:

$$\theta^{(k+1)} = \theta^{(k)} - H^{-1} g. \tag{2.1}$$

Notably, Newton's method achieves local quadratic convergence when certain conditions are met (Nocedal & Wright, 1999). However, Newton's method is not widely used in deep learning due to several fundamental problems that make it impractical for large-scale neural network training: Firstly, it requires computing and storing the Hessian of the loss function with respect to n network parameters, which has a size of

---

[1] The expected risk is defined as: $\mathbb{E}_{(x,y)\sim\mathcal{P}_{\text{data}}}\left[\ell\left(f(\theta, \mathbf{x}), \mathbf{y}\right)\right]$, where ℓ is the loss function, f is the model, θ are the model parameters, $\mathbf{x}$ is the input, and $\mathbf{y}$ is the target label.

$n \times n$, resulting in a high computational and storage complexity of $O(n^2)$. Secondly, the Hessian needs to be inverted, which is an operation with a complexity $> O(n^2)$, depending on the inversion algorithm. Thirdly, due to the large size of datasets, training relies on mini-batch sampling, and full-batch optimization is in most cases computationally prohibitive. This introduces stochasticity into the training process, such that taking the Hessian with respect to a mini-batch introduces noise. Newton's method in its standard form assumes full-batch optimization. Lastly, loss functions are often highly non-convex, containing many local minima and saddle points. Newton's method is well-suited for convex functions, but in non-convex cases, the Hessian can have negative eigenvalues, leading to a direction of ascent instead of descent.

2.3.2 Quasi-Newton Methods

In search for alternatives we briefly turn to *Quasi-Newton Methods*. Quasi-Newton Methods are a class of optimization methods that build on Newton's method and can be used when the Hessian matrix is either unavailable or too expensive to compute and invert. Instead of computing the inverse Hessian matrix directly, these methods approximate it using past gradient information, potentially saving computational and memory cost. The first quasi-Newton method was proposed in the 1950s and later published by Davidon (1991) and refined by Fletcher and Powell (1963). This method, now referred to as the "Davidon-Fletcher-Powell formula," iteratively approximates the inverse Hessian by leveraging gradient and variable changes. Building on this approach, separate similar proposals in 1970 lead to the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm (Broyden, 1970, Fletcher, 1970, Goldfarb, 1970, Shanno, 1970), which improved numerical stability and convergence behavior. However, these methods are still computationally and memory-intensive, making them unsuitable for very large-scale optimization problems common in deep learning. To address this, Limited-memory BFGS (L-BFGS) was proposed by D. C. Liu and Nocedal (1989), significantly reducing the memory burden by storing only a limited history of updates. Despite this improvement, L-BFGS is not widely used in deep learning as it is primarily designed for full-batch optimization. While its memory footprint is smaller than that of BFGS, the memory and computational costs still scale with network size and often become too large to compete with more efficient methods.

2.3.3 Approximate Curvature Matrices

Next, we turn to a class of optimization methods that aim to approximate the curvature of the loss function in a computationally efficient way. While quasi-Newton methods approximate the inverse Hessian using past gradients, second-order approximation methods often directly leverage the structure of the optimization problem to estimate curvature more efficiently.

Gauss-Newton Method

The Gauss-Newton (GN) method, named after Carl Friedrich Gauß and Isaac Newton, provides a structured approximation to Newton's method that is designed for least-squared problems (Nocedal & Wright, 1999). It replaces the full Hessian matrix with an approximation based on the Jacobian of the model outputs, making it more practical while still capturing useful curvature information. In the following two derivations we consider a data point $(input, target) = (x, y)$, a model function $f(x, \theta)$ with the output z and m output dimensions, v_i as the i-th entry in a vector, and model parameters θ. For simplicity, the derivations are presented for one sample. However, they generalize directly to the empirical risk (i.e., training on a dataset or mini-batch) by averaging the individual contributions of each sample—i.e., by defining the objective function, the Hessian, and the Gauss-Newton matrix as the mean over all data points. To derive the Gauss-Newton, based on Martens (2020), we consider an loss function of the form:

$$\mathcal{L}(y, z) = \frac{1}{2} \left\| y - z \right\|^2 ,$$

The Hessian of $\mathcal{L}(y, z)$ with respect to (w.r.t.) θ is given by:

$$H = \nabla^2 \mathcal{L}(y, z) = J_f^\top J_f - \sum_{i=1}^{m} \Big[y - f(x, \theta) \Big]_i H_{[f]_i},$$

Where J_f is the Jacobian of f w.r.t. θ and $H_{[f]_i}$ the Hessian of the i-th component of f w.r.t. θ. The Gauss-Newton method approximates the Hessian by neglecting the second-order term:

$$H \approx J_f^\top J_f = H_{\text{GN}}. \tag{2.2}$$

Substituting this into the Newton step (2.1), we obtain the Gauss-Newton step:

$$\theta^{(k+1)} = \theta^{(k)} - (J_f^\top J_f)^{-1} g.$$

To improve its stability in the presence of ill-conditioned problems, Levenberg (1944) introduced a damping factor, which was later refined by Marquardt (1963) into the well-known Levenberg-Marquardt algorithm. Despite their success in classical optimization, these methods are rarely used in deep learning due to their memory overhead and reliance on full-batch optimization.

Generalized Gauss-Newton Method

While the classical Gauss-Newton (GN) method provides an efficient approximation to Newton's method for least-squares problems, it does not generalize well to arbitrary loss functions. To address this, the Generalized Gauss-Newton (GGN) method extends the GN approximation to a broader class of functions, particularly those used in deep learning (Schraudolph, 2002). We derive the GGN based on Martens and Sutskever (2012). Consider a differentiable loss function of the form $\mathcal{L}(f(x,\theta),y)$. Unlike the least-squares case, the loss function here is more general. The Hessian of $\mathcal{L}$ is given by:

$$H = J_f^\top \, H_{\mathcal{L}} \, J_f \; + \; \sum_{i=1}^{m} \Big[\nabla_z \mathcal{L}(y,z) \Big]_i \, H_{[f]_i}.$$

where J_f denotes the Jacobian of f w.r.t. θ, $H_{\mathcal{L}}$ and $\nabla_z \mathcal{L}(y,z)$ respectively are the Hessian and the gradient of $\mathcal{L}$ w.r.t. z evaluated at $z = f(x,\theta)$. $H_{[f]_i}$ is the Hessian (w.r.t. θ) of the i-th output of f. In analogy to the classical Gauss-Newton method (2.2), the Generalized Gauss-Newton (GGN) approximation simplifies the Hessian by neglecting the second term $\sum_{i=1}^{m} \Big[\nabla_z \mathcal{L}(y,z) \Big]_i \, H_{[f]_i}$, leading to the approximation

$$H_{\text{GGN}} = J_f^\top H_{\mathcal{L}} J_f, \tag{2.3}$$

Substituting this into the Newton step (2.1), we obtain the update step

$$\theta^{(k+1)} = \theta^{(k)} - (J_f^\top H_{\mathcal{L}} J_f)^{-1} g.$$

This formulation extends Gauss-Newton to general loss functions while maintaining its structured approximation. Note that if the loss function is least squares the GGN matrix reduces to the classical Gauss-Newton matrix. To provide further context of the GGN method and curvature approximations, we will now discuss the Fisher Information Matrix (FIM) and its role in optimization.

Fisher Information Matrix and Natural Gradient Descent

In the context of neural networks, the Fisher Information Matrix (FIM), based on Fisher Information (Fisher, 1922), is defined in terms of the loss function $L(\theta)$. It measures how much information the observed data provides about the model parameters θ and is given by:

$$F(\theta) = \mathbb{E}_{p(y|x;\theta)} \left[\nabla_\theta L(y, f(x;\theta)) \cdot \nabla_\theta L(y, f(x;\theta))^T \right]$$

where $f(x;\theta)$ represents the neural network output, and $L(y, f(x;\theta))$ is the loss function (Martens & Grosse, 2015). This expectation is taken over the data distribution, and the FIM serves as an approximate second-order curvature matrix. Note that the GGN and the FIM are closely related. When the loss is the negative log-likelihood from a well-specified model, the GGN matrix exactly matches the sensitivity measured by the Fisher information matrix (Martens, 2020).

To account for the intrinsic geometry of the parameter space, Amari (1998) proposed Natural Gradient Descent which modifies the Gradient Descent update rule by incorporating the inverse Fisher information matrix:

$$\theta_{t+1} = \theta_t - \eta F(\theta)^{-1} \nabla_\theta L(\theta)$$

Although the FIM and Natural Gradient Descent are powerful tools, they are still computationally expensive due to the inversion of $F(\theta)$ and are not widely used in deep learning.

Kronecker-Factored Approximate Curvature

Another curvature approximation that Martens and Grosse (2015) proposed is the Kronecker-Factored Approximate Curvature (K-FAC) method, which approximates the FIM by making assumptions about the particular structure of the network and the way it is parameterized. K-FAC is designed to scale to large neural networks by approximating the FIM with low-rank factors, reducing the computational cost of inverting the curvature matrix. This framework, which was originally only applicable for fully connected networks, has since been extended to convolutional neural networks (Grosse & Martens, 2016), recurrent neural networks (Martens et al., 2018) and weight-sharing architectures (Eschenhagen et al., 2023).

Further Approximations

Diagonal, block-diagonal (like K-FAC), low-rank, and other approximations are used to reduce the computational complexity of curvature matrices. Diagonal approximations retain only the per-parameter curvature, ignoring interactions between parameters, while low-rank approximations approximate a curvature matrix with a matrix of lower rank, reducing computational cost and memory requirements. For example Ada-Hessian (Yao et al., 2021) focuses on the diagonal Hessian and has seen success in training large models. Dangel et al. (2022) or Yang et al. (2022) leverage the low rank structure of the GGN and the FIM to access curvature.

2.3.4 Hessian-Free Optimization

Hessian-free optimization is a class of optimization methods that avoid forming curvature matrices explicitly. Martens (2010) introduced Hessian-free optimization in the context of training deep neural networks, showing that it can outperform first-order methods in certain scenarios. One key idea is to approximate the Hessian-vector product using the Pearlmutter method (Pearlmutter, 1994), which avoids direct Hessian computation. This allows for efficient second-order updates without requiring explicit

Hessian or approximate Hessian storage, making it more scalable for deep learning applications. Hessian-free methods have been applied in training e.g. recurrent neural networks, where gradient-based methods struggle due to vanishing and exploding gradients (Martens & Sutskever, 2011).

2.3.5 Full Second Order Methods in Training Deep Neural Networks

Despite the progress made in finding second-order approximations that are more affordable, most approaches to training deep neural networks do not incorporate these methods. Some of the reasons for this, as previously mentioned, include computational cost, memory requirements and struggles with the stochasticity of mini-batch training process. Overall, first order methods often offer good performance without expensive curvature information, making them more practical for training deep neural networks. However, first order methods also have their limitations, such as the need for careful hyperparameter tuning and step size selection. In the next section, we will discuss step size selection and hyperparameter optimization, which are crucial for the performance of first order optimization methods.

2.4 Step Size Selection

Line-search methods are optimization techniques used to determine a suitable step size along a given direction in iterative optimization algorithms.
Their aim is to find the optimal step size α^* that minimizes the objective function $f(x)$ along the direction d_k, given a current iterate x_k and a descent direction d_k:

$$\alpha^* = \arg\min_{\alpha>0} f(x_k + \alpha d_k) \tag{2.4}$$

Exact line search methods such as ‘Golden Section Search’ (Kiefer, 1953) or ‘Newton’s Line Search’ (Nocedal & Wright, 1999) find the optimal step size in each iteration.
Inexact line search methods, on the other hand, rely on conditions that ensure a sufficient decrease in the objective function per step and sometimes also a gradient change. Examples of these used in traditional optimization are the ‘Goldstein conditions’ (Goldstein, 1965), or the ‘Wolfe conditions’ (Wolfe, 1969). However, both exact and inexact methods typically require multiple function evaluations, making them costly in deep networks.

Step Size Selection in Training Deep Neural Networks

As previously discussed, significant progress has been made in first order methods leveraging adaptive learning rates based on gradient information on a single parameter basis, but these methods still require a hyperparameter learning rate which controls the overall magnitude of parameter updates. Setting this learning rate too high can cause the algorithm to diverge while setting it too low can lead to slow convergence (Goodfellow et al., 2016). Considering this, we find that constant learning rates can lead to suboptimal performance and the ideal step size changes over time. Therefore the learning rate is often adapted over time, which we will discuss in the following.

Learning Rate Schedules

Learning rate schedules have proven to be an effective method for training deep neural networks. Often, the learning rate decays over time, such that it is high enough in early stages of training and smaller near convergence. Also, increasing the learning rate linearly initially as a so-called “warm-up” has shown to be beneficial as it forces the network to more well-conditioned areas of the loss landscape (Kalra & Barkeshli, 2025, L. Liu et al., 2019). Many different variants of learning rate schedules have been proposed. Step decay schedules reduce the learning rate at pre-determined stages by a fixed factor and exponential decay schedules iteratively reduce following an exponential function. Cyclical learning rate methods oscillate the learning rate between predefined minimum and maximum values over training epochs (Smith, 2017). Cosine annealing schedules adapt the learning rate to follow a cosine curve, sometimes resetting at specified intervals to encourage exploration (Loshchilov & Hutter, 2017b). While these schedules lead to significant performance advances, they typically still require a search for efficient hyperparameters, such as the peak of the learning rate or decay factors. The next section will outline approaches to hyperparameter optimization, to find sufficient hyperparameters such as the learning rate.

Hyperparameter Optimization

This section will focus on hyperparameter optimization techniques, which seek to choose a set of optimal hyperparameters for a training algorithm including the step size or learning rate. Specifically, we will discuss each technique and its implications for employing it in training deep neural networks. Note that, in the context of DNNs, finding optimal hyperparameters is generally not possible, as it would involve evaluating all possible hyperparameter combinations. Thus, in practice, the search-spaces for hyperparameter searches are finite and aim to find a sufficiently good solution.

Manual search is the most basic hyperparameter optimization method: The user takes an initial guess that is based on intuition and experience and then attempts to find a sufficient solution through trial and error. While this approach has often been deployed, it can be time-and resource-consuming as it is prone to the biases of the user.

A more systematic hyperparameter optimization method is grid search: The user specifies a finite set of values and evaluates the performance of all combinations of them. In training DNNs, this typically means performing a full training run on each configuration and then evaluating the results through, e.g., the loss on the validation set. A problem with this approach is that the number of combinations and hence evaluations grow exponentially with the dimensionality of the configuation space (Feurer & Hutter, 2019).

An alternative to grid search is random search, where random configurations are tested until a budget for the search is depleted. Random search has been shown to be more efficient than grid search in many cases (Bergstra & Bengio, 2012).

Bayesian Optimization is an iterative process that builds a probabilistic model, often called 'surrogate model' to estimate the performance of different hyperparameter values (Feurer & Hutter, 2019). The acquisition function determines which hyperparameter values to test next, aiming to balance exploration and exploitation (Feurer & Hutter, 2019). Bayesian optimization has been succesfully deployed in training deep neural networks (Dahl et al., 2013, Snoek et al., 2012, 2015), but is not widely used, as it can be computationally expensive and often struggles with high-dimensional hyperparameter spaces. While there are several other hyperparameter optimization techniques not discussed here, identifying effective configurations remains an open research challenge and is often associated with high computational cost. In the following section we connect our findings, which lead to our research proposal.

2.5 Bridging the Gap: Efficient Step Size Estimation

In section 2.1, we introduced neural networks as capable function approximators and in section 2.2, we discussed the vast abilities of DNNs, their optimization challenges and the sucess of first-order optimization algorithms. We also focussed on the drawbacks of first order optimization algorithms, as they do not consider curvature when training DNNs. Section 2.3 introduced second order optimization methods as a theoretically strong alternative to first order methods and explained their limitations in deploying them for DNNs, as well as progress reducing the cost in different curvature matrix approximations. Section 2.4 introduced line search methods in general and also current step size selection methods for training DNNs, which are often costly as they involve multiple function evaluations. The availability of more cost-effective curvature approximations, such as the GNN, and the high cost and complexity of hyperparameter searches and therefore also step size selection motivates our curvature-based step size approach.

2.6 Related Work

There is a large body of work exploring step size selection or, more specifically, curvature-based step size selection in training DNNs. This section will briefly introduce some related work that incorporates curvature information in the step size selection process, while relying on first order methods to find a descent direction in training DNNs.

We find the most striking similarities to our work in Balboni and Bacciu (2023). They compute step sizes based on the quadratic approximation of the loss leveraging an exponential averaging scheme, a positive semi-definite approximation of the Hessian (which they derive) and SGD. They report outperforming both classical SGD with grid searches and the incorporation of the Gauss-Newton method into their step size selection scheme in their experiments on different network architectures. The main difference compared to our method is that we will not employ SGD but a more complex optimization algorithm in our work, which means that we will select our direction of descent differently. We also do not make use of exponential averaging schemes, use a significantly larger dataset and model and will choose time-to-result as our core performance indicator (in contrast to epochs).

Mutschler and Zell (2020) define a parabola based on two loss evaluations along a search direction which can, in theory, be deterimined by any optimizer. This parabola can be viewed as an approximation of the curvature of the loss. They then propose to choose the step size based on the step size which minimizes this parabola, and report promising results in their experiments. This is tied to the core idea of our proposed step size selection method, such that we will also rely on a quadratic approximation of the loss landscape to select a step size which minimizes this parabola. Another example are Robles-Kelly and Nazari (2019) incorporating the “Barzilai-Borwein step size” (Barzilai & Borwein, 1988), also involving curvature approximation and multiple loss evaluations, into neural network training and known DNN optimization methods. We acknowledge that there is more work on this subject with similarities to our approach, which we present in the following section.

3 Materials and Methods

3.1 Overview

In this section, we introduce our curvature-based step size selection method, which aims to provide an efficient alternative to conventional step size selection techniques. We first describe the theoretical foundation behind our approach, followed by implementation details and the experimental setup used for evaluation. The derivation of the step size is not unique to this work, it is based on common optimization principles. However how exactly we deploy it in the context of training DNNs is, to our knowledge, novel.

3.2 Step Size Derivation

The proposed step size can be derived as follows:

Given a descent direction d define a quadratic approximation q of the loss $L(\theta)$ around the point θ_0:

$$L(\theta) \approx q(\theta) := \frac{1}{2}(\theta - \theta_0)^\top H(\theta - \theta_0) + (\theta - \theta_0)^\top g + L(\theta_0) \tag{3.1}$$

Consider a cut h through the quadratic q along d:

$$h(\alpha) = q(\theta_0 + \alpha \cdot d) = \frac{1}{2}\alpha^2 \cdot d^\top H \cdot d + \alpha \cdot d^\top g + L(\theta_0)$$

Minimize h:

$$\begin{aligned} \frac{\partial h}{\partial \alpha} &= \alpha \cdot d^\top H \cdot d + d^\top g \overset{!}{=} 0 \\ \Leftrightarrow \quad \alpha \cdot d^\top H d &= -d^\top g \\ \Leftrightarrow \quad \alpha^* &= \frac{-d^\top g}{d^\top H d} \end{aligned} \tag{3.2}$$

$$\frac{\partial^2 h}{\partial \alpha^2} = d^\top H d > 0, \text{ if } H \text{ is positive definite}$$

Thus, α^* represents a step size that minimizes the quadratic q when stepped in the direction d and if H is positive definite. As discussed in section 2.3.4, 'Hessian-Free' optimization can leverage that a multiplication of the form $\underset{N\times N}{M} \cdot \underset{N\times 1}{v}$ can be achieved at much more feasible cost in terms of memory and computation when the matrix M is not explicitly stored, making it more cost-effective when deployed in DNNs (Martens, 2010). This can be leveraged to compute α^* at lower cost. Furthermore, one problem remains: The Hessian can not be assumed to be positive definite in non-convex loss landscapes. In training DNNs loss landscapes are usually non-convex, which leads us to an alternative curvature measure: The GGN matrix, derived in section 2.3.3, provides an approximation of the Hessian. The GGN matrix can be guaranteed to be positive semi-definite (Martens & Sutskever, 2012, Tatzel et al., 2024). Therefore, if $H_{GGN} = 0$ the term α^* becomes ill-defined. This can theoretically lead to problems when computing α^* using the GGN matrix and we will see in the experimental section if it does so. In all other cases $d^\top H_{GGN} d > 0$, therefore the step $\alpha^* \cdot d$ is in a direction of descent and the step size α^* is well-defined. The GGN has also been shown to outperform the Hessian in DNN training in some cases (Martens, 2010), making it a suitable candidate to replace the Hessian in the computation of α^*. Thus, we propose to compute α^* as:

$$\alpha^* = \frac{-d^\top g}{d^\top H_{\mathrm{GGN}} d} \tag{3.3}$$

The step size α^* can be computed for any direction d. In practice this means that any algorithm which selects a descent direction d can be used. Therefore one advantage of this curvature-based step size selection in training DNNs is that it can be employed with validated and efficient optimization algorithms such as SGD, Adam (Kingma & Ba, 2015), or their variants.

Debiasing Alpha

We will now motivate a different approach to computing α^* to cope with possible biases in mini-batch training. Tatzel et al. (2024) investigate mini-batch quadratic approximations that involve computing directions and curvature estimates, such as we do in our proposal. Through comparison of full-batch and mini-batch quadratics, they identify bias in curvature (approximations), most pronounced in late stages of training, and in directional slopes, most pronounced in early and late stages of training. They suggest that the bias in the first direction of the Conjugate Gradient (CG) method (Hestenes, Stiefel, et al., 1952) is due to the gradient in each mini-batch maximizing steepness on that minibatch, leading to an overestimation compared to other mini-batches. Tatzel et al. (2024) identify curvature bias in mini-batch quadratic approximations as primarily resulting from misaligned eigenspaces between mini-batch Hessians. They argue, that the highest and lowest curvature in one mini-batch are, respectively, overestimated and underestimated compared to other mini-batches, as they have different directions of extreme curvature. Tatzel et al. (2024) expect this bias to extend also to different datasets, models and curvature proxies such as ours, because they seem to be inherent to the stochastic mini-batch process. They also suggest a strategy to mitigate this issue: They propose a two-batch strategy, where the

directions are determined using one mini-batch, but directional derivatives (curvature and slope) are estimated using a different mini-batch. They find that this approach leads to improved stability and performance when deployed on CG method or in the Laplace approximation (MacKay, 1992).
To adress and investigate this issue, we will examine both the biased α^* as derived in equation (3.2) and "unbiased" α^*, which we denote as $\alpha^*_{unbiased}$, derived through the two-batch strategy. The unbiased version will be computed as:

$$\alpha^*_{unbiased} = \frac{-d_2^\top g_1}{d_2^\top H^1_{\mathrm{GGN}} d_2} \tag{3.4}$$

where d_2 refers to a derivation from batch 2 and g_1, H^1_{GGN} from batch 1.

Additionally, to quantify the impact of the numerator and denomitator on the difference between α^* and $\alpha^*_{\text{unbiased}}$ in later analysis, we also define the hybrid metric as

$$\alpha_{\text{hybrid}} = \frac{\text{numerator}_{\text{unbiased}}}{\text{denominator}_{\text{biased}}} = \frac{d_2^\top g_1}{d_2 \top H^2_{\mathrm{GGN}} d_2}. \tag{3.5}$$

We then decompose the overall difference:

$$\alpha^*_{\text{unbiased}} - \alpha^*_{biased} = (\alpha_{\text{hybrid}} - \alpha^*_{biased}) + (\alpha^*_{\text{unbiased}} - \alpha_{\text{hybrid}}).$$

Since

$$\alpha_{\text{hybrid}} - \alpha^*_{biased} = \frac{\text{numerator}_{\text{unbiased}} - \text{numerator}_{\text{biased}}}{\text{denominator}_{\text{biased}}},$$

this term isolates the numerator correction, while $\alpha^*_{\text{unbiased}} - \alpha_{\text{hybrid}}$ captures any remaining effects due to the denominators.

3.3 Evaluation Framework, Dataset, Model and Optimizer

3.3.1 Evaluation Framework

This section will focus on the specific framework, dataset, model and optimizer that were used to train and evaluate the applicability of the α^* step size. The aim of this thesis is to deploy and evaluate the proposed step size approach on a state-of-the-art DNN and optimizer to ensure that it is tested in a realistic setting.
Evaluating and comparing algorithms in training DNNs is a non-trivial task. There are many different training algorithms and evaluation procedures that sometimes lead to different results, e.g. (Choi et al., 2019) show that hyperparameter tuning protocols have strong influence on performance. Therefore, to test our method, a sufficient

benchmark procedure is needed. Building on advances in comparison benchmarks (see e.g., Moreau et al., 2022, Schmidt et al., 2021, Schneider et al., 2019), Dahl et al. (2023) proposed the "AlgoPerf: Training Algorithms" benchmark, which provides large, diverse workloads, strong baselines, clear tuning procedures and scores training runs. Therefore, we selected it as the basis for our DNN trainings and comparisons. The original benchmark was introduced in a competition and is split into the 'external tuning', which allows submissions to define hyperparameter search spaces and the 'self tuning' ruleset, in which submissions are "not allowed to have user-defined hyperparameters and therefore receive no extra tuning"(Dahl et al., 2023, Sec. 4.4.2). To eliminate biases and pre-tuning, competitors had to submit their algorithms, which were then tested on known fixed workloads and held-out randomized workloads, disallowing workload-specific hyperparameter choices in the self-tuning ruleset. As our method proposes curvature-based step sizes which aims to eliminate excessive tuning cost, our trainings and comparisons are based on the 'self tuning' branch of the benchmark.
The benchmark provides several workloads and modified architecturural variations within each workload. We selected the non-modified base workload 'Criteo1TB' for the following reasons: The original AlgoPerf benchmark competition was designed to train on 8× NVIDIA V100 GPUs with 16GB VRAM per GPU, we used one NVIDIA A100 with 40GB VRAM. We access this hardware in a compute cluster. To make multiple training runs more feasible, we sought a workload expected to require relatively little training time. This is the case in the Criteo1TB workload, which has the shortest maximum submission time, indicating that its targets were reached within the shortest amount of time compared to other workloads. The submission time to a result, which excludes regular logging and non-timed evaluations, is the central performance measure in the AlgoPerf competition (Dahl et al., 2023).

3.3.2 Criteo 1TB

The Criteo1TB workload includes the Criteo 1TB Logs dataset (Criteo A. I. Lab, 2014) and the Deep Learning Recommendation Model (DLRM) model to predict the click-through rate (Naumov et al., 2019). The dataset consists of 4,195,197,692 training examples, each with 13 numerical and 26 categorical features. Numerical features are log-transformed and categorical features are hashed into a single embedding table. The training data is shuffled in 5-million-line chunks. Dahl et al. (2023) employ sigmoid binary cross-entropy loss to enable per-example decomposition of model performance. The dataset is structured into 23 days for training and the 24th day split equally into validation and test sets (~89 million examples each).

3.3.3 DLRMsmall Model

The DLRMsmall model was originally proposed by Naumov et al. (2019) and is implemented by Dahl et al. (2023) for the AlgoPerf framework. DLRM is designed for large-scale recommendation systems, integrating both categorical and dense numerical features within a neural network architecture. The model consists of three main components: embedding layers, fully connected Multi-Layer Perceptron (MLP) layers, and interaction layers. The categorical features are mapped into dense representations through a single embedding table containing 4 million entries with an embedding dimension of 128. Meanwhile, numerical (dense) features are processed through a three-layer fully connected neural network with layer sizes of 512, 256, and 128 units. The outputs of these transformations are then concatenated.

A cross-interaction layer captures feature interactions by computing dot products between the transformed dense features and the embedded categorical features. This feature representation is subsequently passed through a deeper MLP network with five layers consisting of 1024, 1024, 512, 256, and a final output neuron. A dropout layer follows the 512-dimensional hidden layer in the second MLP network. All layers use ReLU[1] activations, except for the final output layer, which produces the model's prediction.

3.3.4 Optimizer

As the optimizer for our experiments, we used NAdamW algorithm, which is the baseline algorithm in the AlgoPerf competition. This optimizer combines a Nesterov momentum-accelerated adaptation of Adam ("NAdam") as in Choi et al. (2019) with Adam with decoupled weight decay regularization ("AdamW") as per Loshchilov and Hutter (2017a). This algorithm was selected as a baseline by Dahl et al. (2023) through testing several promising algorithms: AdamW, NadamW, Nesterov, Heavy Ball, LAMB, Adafactor, SAM (with Adam), and Distributed Shampoo were compared in terms of their overall performance across all workloads and following a specific tuning protocol. As per the self-tuning rules, the baseline algorithm can only have one hyperparameter configuration. The baseline NAdamW configuration for the self-tuning ruleset is the hyperparameter configuration with the best overall performance across all workloads (Dahl et al., 2023). That does not necessarily imply that this NAdamW configuration performed best on every single workload, or specifically on the Criteo1TB workload.

---

[1] The Rectified Linear Unit (ReLU) activation function is defined as

$$f(x) = \max(0, x)$$

3.4 Algorithm and Implementation

Next, we will focus on the algorithms and specific implementations that were employed to compute α^* during neural network training, enabling the computation and application of α^*. All experiments and implementations were conducted using the PyTorch deep learning framework (Paszke et al., 2019). For a given neural network and optimizer, the values that need to be obtained to calculate α^* during training are the gradient of the loss w.r.t. parameters g, a descent direction d suggested by the optimizer and the GGN matrix vector product $H_{GGN} \cdot d$ of the GGN matrix of the loss w.r.t. parameters.

```
Require: $\mathcal{D}$ (Dataset), $f(\cdot)$ (Forward Pass), $\mathcal{L}$ (Loss), $\theta$ (Parameters), $\mathcal{O}$ (Optimizer),
    learning rate $\eta$
while not converged do
    Sample new batch $\mathcal{B}$ from $\mathcal{D}$
    $\theta_0 \leftarrow$ parameters_to_vector($\theta$)
    $H_{\text{GGN}} \leftarrow$ GGNLinearOperator($f, \mathcal{L}, \mathcal{B}, \theta$)
    Forward pass $f(\mathcal{B})$
    Compute $\mathcal{L}$ and backpropagate gradients
    $g \leftarrow$ gradients
    Perform optimizer step using $\mathcal{O}$
    $\theta_1 \leftarrow$ parameters_to_vector($\theta_{\text{new}}$)
    $d \leftarrow \theta_1 - \theta_0$
    $Hd \leftarrow H_{\text{GGN}}(d)$
    $\alpha^*_{rel} \leftarrow \frac{-d^\top g}{d^\top (Hd)}$
    $\alpha^*_{abs} \leftarrow \alpha^*_{rel} \cdot \eta$
    if apply $\alpha^*$ then
        $\theta \leftarrow \theta_0 + \alpha^*_{rel} \cdot d$
    end if
end while
```

Figure 3.1: Algorithm for Observing or Applying α^*. An algorithm to determine α^* for any optimizer. Note: Through this derivation of the step direction d, it is scaled by the learning rate η that was applied during the step. Therefore, α^*_{rel} represents the factor by which our method proposes to scale the learning rate η with. α^*_{abs} represents the absolute learning rate that our method proposes.

The parameter gradients are computed during backpropagation and can be accessed directly. To obtain the step direction d from any optimizer, the parameters of the network need to be converted to a vector via `torch.nn.utils.parameters_to_vector` before (θ_t) and after an optimizer step (θ_{t+1}), with d computed as the difference (see figure 3.1). The step $\theta_t + \alpha^* \cdot d$ can then be calculated, converting the obtained vector to model parameters with `torch.nn.utils.vector_to_parameters`. In order to obtain the matrix vector product $H_{GGN}d$, we used the curvlinops interface developed by

Dangel et al. (2025), specifically `curvlinops.GGNLinearOperator`.

Whenever necessary in the following sections, we will refer to α^*_{rel} as the relative step size and α^*_{abs} as the absolute step size. α^*_{rel} indicates the factor by which our method proposes to scale the current learning rate η with, and α^*_{abs} indicates the absolute step size that our method proposes.

Figure 3.1 provides the algorithmic logic that we deployed to obtain α^* and that can be used to apply α^* during training. Chapter 4 will lay out how exactly we used this logic in the specific experiments.

3.5 Comparison Baselines

We selected two algorithms as a direct comparison to our proposed method. First, the NAdamW algorithm, which is the baseline algorithm in the AlgoPerf 'self tuning' framework with a learning rate schedule. This algorithm was introduced in section 3.3.4, as we also selected it as our core optimization algorithm which selects the direction d in each iteration.

As our second comparison baseline algorithm, we selected the "Schedule-Free AdamW" algorithm (Defazio et al., 2024) provided by Meta Platforms, Inc., which is licensed under the Apache License 2.0[2] . This is the winning algorithm of the AlgoPerf 'self tuning' competition, as it performed best in their aggregated score across all workloads. The Schedule-Free AdamW algorithm is designed to eliminate the need for manually specifying a predetermined number of steps T for the learning rate schedule (Defazio et al., 2024). It incorporates momentum via interpolation, a linear warm-up period of the learning rate, and Adam's bias-correction term (Kingma & Ba, 2015). Additionally, it deploys a weighted parameter averaging scheme which reduces the step size contributions for later iterates, similar to a decaying learning rate schedule. This algorithm represents a good comparison, as it aims to achieve a similar goal as our method, namely to eliminate the need for manual tuning of the learning rate schedule. However, it does so without directly accessing curvature.

In the original competition, both algorithms completed the workload Criteo1TB consuming 25% of its maximum time budget.

---

[2] We used the implementation provided by Meta Platforms, Inc. through their AlgoPerf competition submission. [https://github.com/mlcommons/algorithms_results_v0.5/tree/main/AlgoPerf_Team_10/self_tuning/schedule_free_adamw]. We did not modify the implementation but used it with a different batch size.
The algorithm is licensed under https://github.com/facebookresearch/schedule_free/blob/main/LICENSE

4 Experiments, Results and Discussion

This section will lay out the exact experimental designs we chose to observe the magnitude of our proposed step size in Experiment 1 and to deploy our method in Experiment 2. First, we will lay out some of our pre-experiments, then we will present both experiments, their results, and discuss them.

4.1 Pre-Test and Sanity Check

4.1.1 Sanity Check

To check if our method works as intended, we first deployed a sanity check. For this check we leveraged that the GGN matrix and the Hessian coincide under certain conditions:

Consider a linear model with one parameter given mean and squared error (MSE) loss

$$f(\theta) = \theta \cdot x, \qquad L(\theta) = \frac{1}{2}\left(f(\theta) - y\right)^2 = \frac{1}{2}(\theta x - y)^2$$

The Hessian is defined as:

$$H = \frac{d^2 L}{d\theta^2} = \frac{d}{d\theta}\left[\frac{dL}{d\theta}\right] = \frac{d}{d\theta}\left[(\theta x - y)x\right] = x^2.$$

The generalized Gauss-Newton matrix is defined as:

$$H_{GGN} = J^T J = x^2 \Leftrightarrow H_{GGN} = H$$

where J is the Jacobian of f with respect to θ.
Given this relationship and considering that the quadratic approximation q of the loss from equation 3.1 is exact because $L(\theta)$ is strictly quadratic over θ when L is the MSE loss, we can conclude the following: The quadratic optimal step size α^* is also the optimal step size in L. Thus, given a one-dimensional linear model with $L =$ MSE loss, one random datapoint $(x, y) = (input, target)$, and a random parameter θ, applying α^* as the step size for one step should (1) result in the optimal parameter, (2) adjust the model output $f(x)$ to be exactly the target and (3) result in a loss $L(f(x))$ of zero.

We used this relationship to test[1] if our method works as intended: We constructed the model, loss and datapoint as described above and then computed and applied the quadratic optimal step $\alpha^* \cdot d$. We used SGD as the optimizer to derive d. Accepting an error $\leq 1 \cdot 10^{-6}$ to account for possible accumulated floating point precision errors, the test passed and achieved (1),(2) and (3).

4.1.2 Pre-Test

As an additional test, we implemented the full computation of α^* during training on our desired dataset and model to estimate the cost of computing α^*. To adjust for the reduction in number of GPUs from eight in the AlgoPerf competition to one, we reduced the original batchsize of 262,144 by a factor of eight to 32,768. We found that the maximum allocation of GPU memory in this setting was >85% of its total memory. We adjusted the available baseline submission that implements the NAdamW optimizer and adapted it to also compute the GGN matrix H_{GGN}, direction d, gradient vector and α^*. The training time was heavily dominated by the computation of the GGN matrix and the other computations that we introduced to compute α^* (see table 4.1).

Computation category	Average time in seconds
Complete step	43.1561
GGN linear operator	36.9928
Other computations related to α^*	5.5510
Non alpha α^*-related	0.6124

Table 4.1: Average Computation Times per Step. Average per-step times needed to compute α^* over 5,000 steps of training. The computation of the `GGN.LinearOperator` dominates the time per step (>85% of time per step), while necessary calculations for α^* account for >98% of the time per step, indicating a more than 50× increase in time per step compared to when not computing α^*.

4.2 Experiment 1: Observing Alpha

In our first experiment[2], we aimed to analyze α^* and $\alpha^*_{\text{unbiased}}$ along with their key components, such as the numerator $(-d^T g)$ and the denominator $(d^T H_{\text{GGN}} d)$, throughout a realistic training run. For this, we selected the self-tuning ruleset configuration of the baseline algorithm NAdamW, which employs a learning rate schedule. This schedule

---

[1] The test can be found in the following notebook: lr_scheduler_test.ipynb https://github.com/Davidundso/taylor/blob/main/experiments/exp_06_lr_scheduler/lr_scheduler_test.ipynb

[2] The implementation of Experiment 1 can be found in the following file: exp01_nadamw_criteo.py https://github.com/Davidundso/taylor/blob/main/algorithmic_efficiency/my_submissions/sub4_unbiased/exp01_nadamw_criteo.py

initially applies a linear warm-up, gradually increasing the learning rate from zero to a predefined value, followed by a cosine decay. It dynamically adjusts to a predefined maximum number of steps, ensuring that the learning rate decreases to zero after this limit is reached.

As the maximum step number, we used the step hint provided by AlgoPerf, which represents the “max number of steps the baseline algorithms used for their learning rate schedules”(Schneider et al., 2025). To account for the eight times smaller batch size compared to the original baseline, we proportionally scaled the schedule length by a factor of eight. Consequently, by the time the maximum number of global steps is reached-i.e., the learning rate schedule is completed—the same amount of data has been processed as with the original batch size.

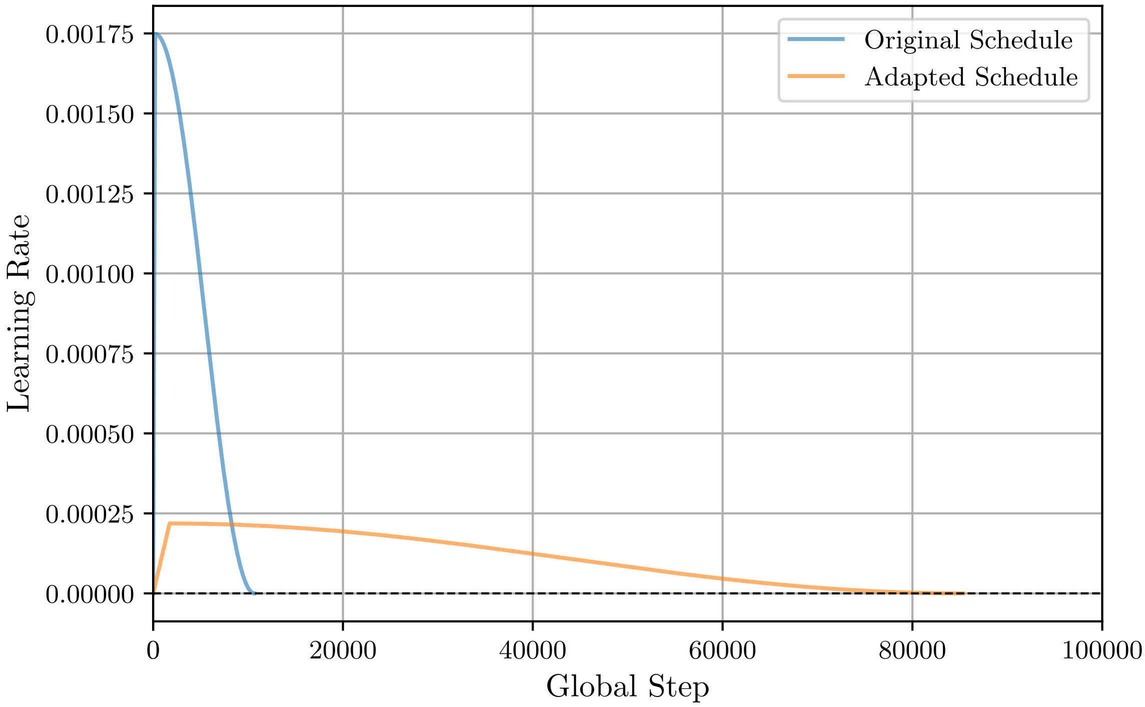


Figure 4.1: Original and Adapted Learning Rate Schedule

Additionally, we reduced the learning rate by a factor of eight to account for the eight times smaller batch size compared to the AlgoPerf baseline trainings (see figure 4.1). This follows an effective scaling rule for adjusting the learning rate to different batch sizes, as demonstrated by Goyal et al. (2017).

Due to the high computational cost of evaluating our target values (see table 4.1), we chose to compute α^* and $\alpha^*_{\text{unbiased}}$ only at specific intervals rather than at every step. Specifically, for the first 50 steps of every 1000-step interval, we used two batches of data, $\mathcal{B}_1$ and $\mathcal{B}_2$: we computed α^* and $\alpha^*_{\text{unbiased}}$ using both batches and performed an optimizer step on $\mathcal{B}_2$. In all other steps, we applied a standard optimizer step using $\mathcal{B}_1$. In every iteration where target values were computed, we also logged the following metrics: the numerators and denominators of α^* and $\alpha^*_{\text{unbiased}}$, the L_2 norms of the gradient vectors and the d_2 vector, as well as the loss on batch $\mathcal{B}_1$ and batch $\mathcal{B}_2$. The L_2 norms and losses were only logged as sanity checks, but not used for further analysis. Additionally, we used cosine similarity[3] to measure the alignment between the vectors g_1 and d_2, as well as g_2 and d_2, and logged the results. Figure 4.2 shows the algorithmic structure of the experiment.

```
Require: $\mathcal{D}$ (Dataset), $f(\cdot)$ (Forward Pass), $\mathcal{L}$ (Loss), $\theta$ (Parameters), $\mathcal{O}$ (Optimizer)
while max global steps not reached do
    Sample new batches $\mathcal{B}_1, \mathcal{B}_2$ from $\mathcal{D}$
    if (global step mod 1000) $> 50$ then
        Perform normal update step on $\mathcal{B}_1$
    else
        $H^1_{\mathrm{GGN}}, g_1 \leftarrow \mathrm{GGN}(f, \mathcal{L}, \mathcal{B}_1, \theta)$, gradients($\mathcal{B}_1$)
        $H^2_{\mathrm{GGN}}, g_2 \leftarrow \mathrm{GGN}(f, \mathcal{L}, \mathcal{B}_2, \theta)$, gradients($\mathcal{B}_2$)
        Perform optimizer step using $\mathcal{O}$ on $\mathcal{B}_2$
        $d_2 \leftarrow$ step direction $d$
        $\alpha_{unbiased} \leftarrow \frac{-d_2^T g_1}{d_2^T H^1_{\mathrm{GGN}} d_2}$
        $\alpha_{biased} \leftarrow \frac{-d_2^T g_2}{d_2^T H^2_{\mathrm{GGN}} d_2}$
    end if
end while
```

Figure 4.2: Algorithmic Structure of Experiment 1.

We also performed evaluation measurements provided by the AlgoPerf framework (Dahl et al., 2023) after every 2000 steps. These measurements include the accumulated submission time, the loss on the training split, which consists of 4,195,197,692 examples (95.92% of the total data), the loss on the test split, which consists of 95,000,000 examples (2.17%), and the loss on the validation split, which consists of 83,274,637 examples (1.90%). We report these precise numbers as they differ slightly from those in Dahl et al. (2023) and therefore also from the numbers in section 3.3.2 where we introduce the workload. These values reflect the exact quantities logged during training. The performance metrics were used both for monitoring the training progress and as ex-post performance indicators.

---

[3] Cosine similarity is defined as:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

where $\mathbf{A}$ and $\mathbf{B}$ are vectors, and $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ denote their Euclidean norms.

4.3 Results: Experiment 1

In this section, we present the results of the first experiment, focusing on the total number of steps required for the learning rate schedule to complete. The schedule concluded after 85,328 steps. Discussion will be provided directly after the results, as the trends observed in the data influenced the design of the second experiment.

4.3.1 Alpha Values

First, we present the main metrics of interest: the values of α^* and $\alpha^*_{\text{unbiased}}$ throughout the training run. Note that the first recorded values of α^* and $\alpha^*_{\text{unbiased}}$ were omitted from all statistics and visualizations, as they are extreme outliers: both relative α^*_{biased} and $\alpha^*_{\text{unbiased}}$ exceeded 10^{14}.
Figure 4.3 displays the averages of the relative α^* values, i.e., the factors by which the learning rate η would need to be scaled to match the step size suggested by α^*_{biased} and $\alpha^*_{\text{unbiased}}$, respectively. On average, biased α^*_{rel} suggests increasing the learning rate by a factor of 1881.79, with a median factor of 55.62. In contrast, unbiased α^*_{rel} suggests an average increase by a factor of 210.88, with a median factor of 1.64. The Pearson correlation coefficient between all values of biased and unbiased α^*_{rel} is 0.842.

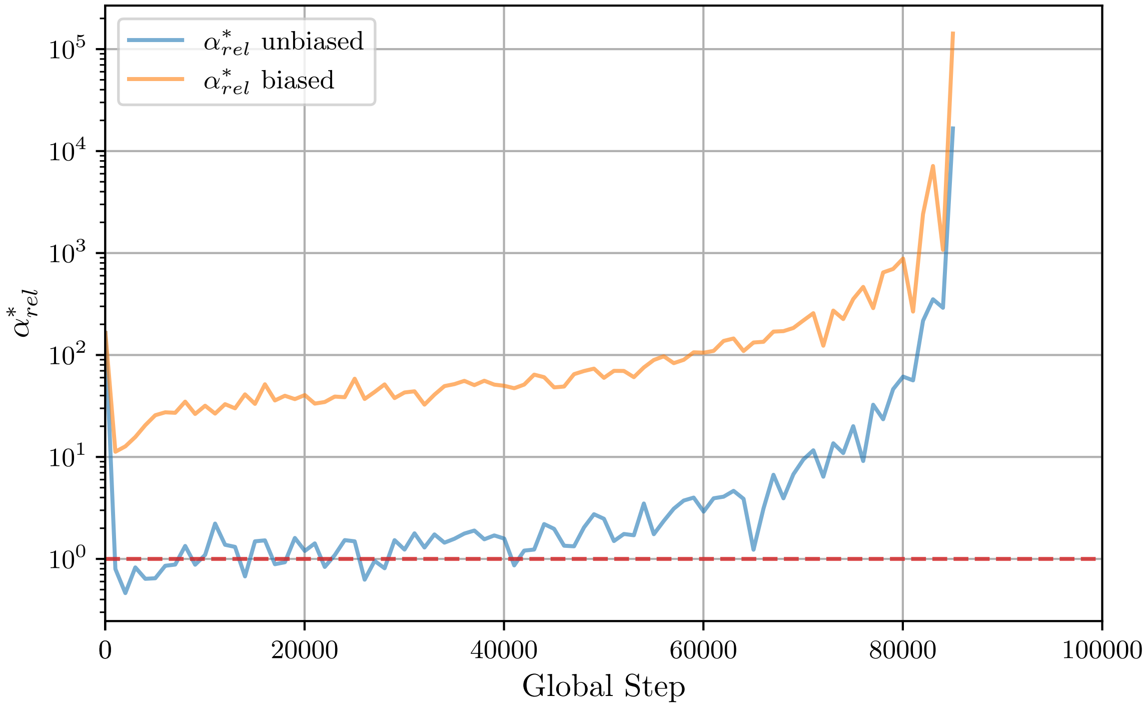


Figure 4.3: Averages of Relative Alpha Biased and Unbiased. Each data point represents the average α^* value of the 50 steps, as α^* was calculated at the start of a 1000-step interval. The red dotted line at y=1 indicates a learning rate equal to the actual current learning rate η from the schedule.

Next, we will examine the absolute values of biased and unbiased α^*, which represent the actual step sizes they suggest. These values are obtained by multiplying the learning rate η with the respective α^*_{rel} value at each step. Figure 4.4 presents the median, mean, and standard deviation of unbiased α^*_{abs}, computed for chunks of 50 consecutive values. Thus each datapoint represents the mean or median of the values computed in the first 50 steps of a 1000-step interval. The mean of unbiased α^*_{abs} is 1.80×10^{-4}, while the median is 1.31×10^{-4}. For reference, the highest learning rate in the schedule was $\eta = 2.19 \times 10^{-4}$, its mean and median is 1.09×10^{-4}. The standard deviation of all computed unbiased α^*_{abs} values is 3.32×10^{-4}. The values of unbiased α^*_{abs} spanned a range of 3.61×10^{-3}, while the means and medians varied within ranges of 4.34×10^{-4} and 3.38×10^{-4}, respectively. Additionally, 1,102 out of 4,300 values (25.63%) of unbiased α^*_{abs} were negative.

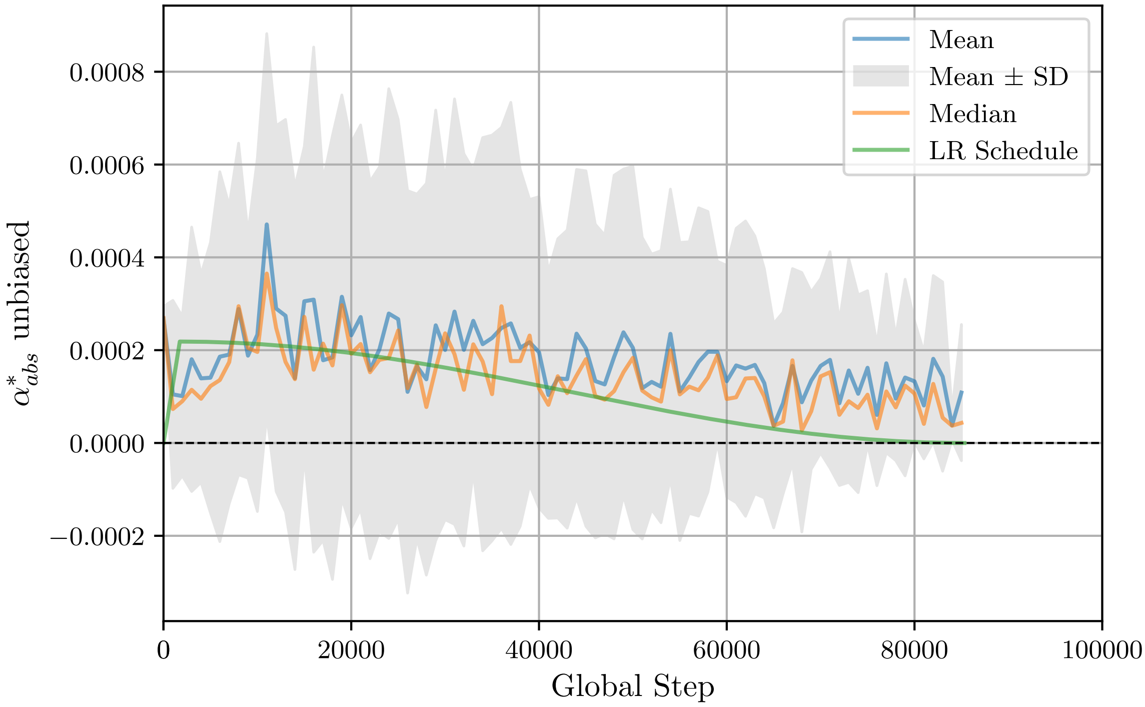


Figure 4.4: Absolute Alpha Unbiased: Mean, Median and Standard Deviation.

Figure 4.5 shows the median, mean, and standard deviation of biased α^*_{abs}. The mean of biased α^*_{abs} is 5.23×10^{-3}, which is 29.03× larger than the mean of unbiased α^*_{abs} and 23.91× larger than the highest learning rate in the schedule.

The median of biased α^*_{abs} is 3.68×10^{-3}, which is 28.17× larger than the median of unbiased α^*_{abs} and 16.86× larger than the highest learning rate in the schedule. The standard deviation of all computed biased α^*_{abs} values is 4.78×10^{-3}, which is 14.37× larger than for unbiased α^*_{abs}. The values of biased α^*_{abs} spanned a range of 2.98×10^{-2}, or a 8.24× larger range compared to unbiased α^*_{abs}. The means varied within a range of 1.04×10^{-2}, which is 23.88× larger than for unbiased α^*_{abs}, while the medians spanned a range of 9.54×10^{-3}, making it 28.24× larger in comparison. All biased α^*_{abs} values were positive.

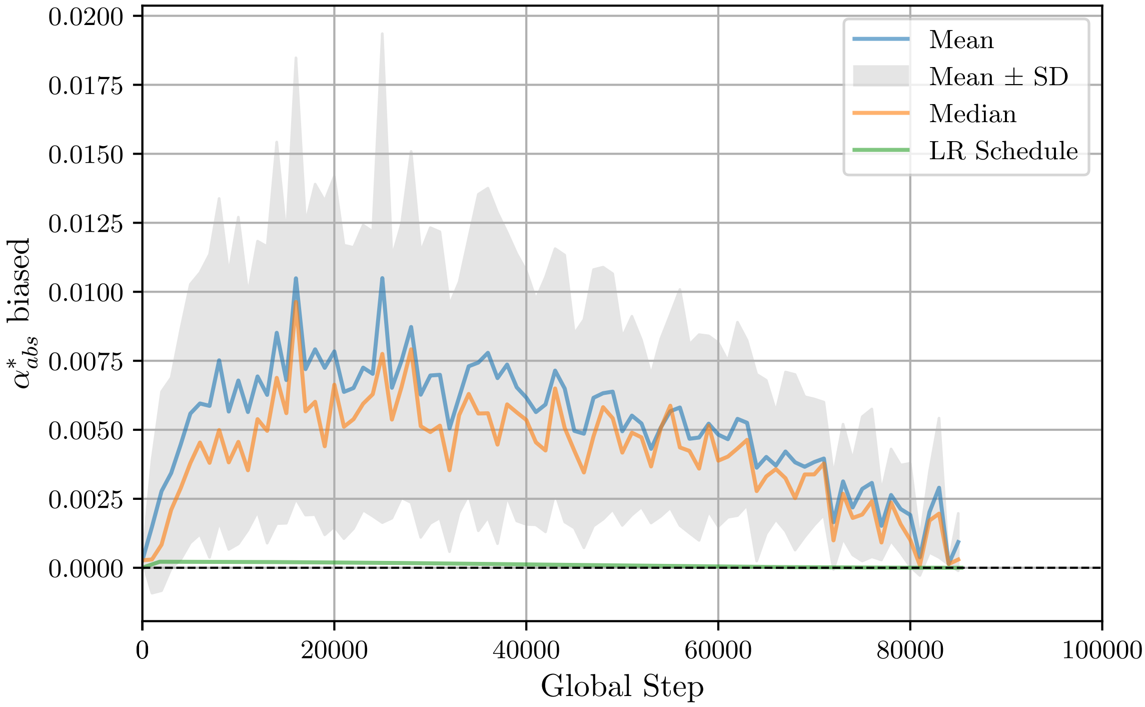


Figure 4.5: Absolute Alpha Biased: Mean, Median and Standard Deviation.

4.3.2 Numerator and Denominator

Next, we present the numerators and denominators of absolute α^*_{biased} and $\alpha^*_{\text{unbiased}}$. Figure 4.6a shows the numerators and denominators of biased α^*. The mean of the numerators is 3.65×10^{-4}, while the median is 3.17×10^{-4}. The denominator has a mean of 3.25×10^{-5} and a median of 5.68×10^{-6}. Figure 4.6b shows the numerators and denominators of unbiased α^*. The mean of the numerators is 5.75×10^{-5}, while the median is 5.23×10^{-6}. The mean and median of the denominator are 3.17×10^{-5} and 5.03×10^{-6}, respectively.

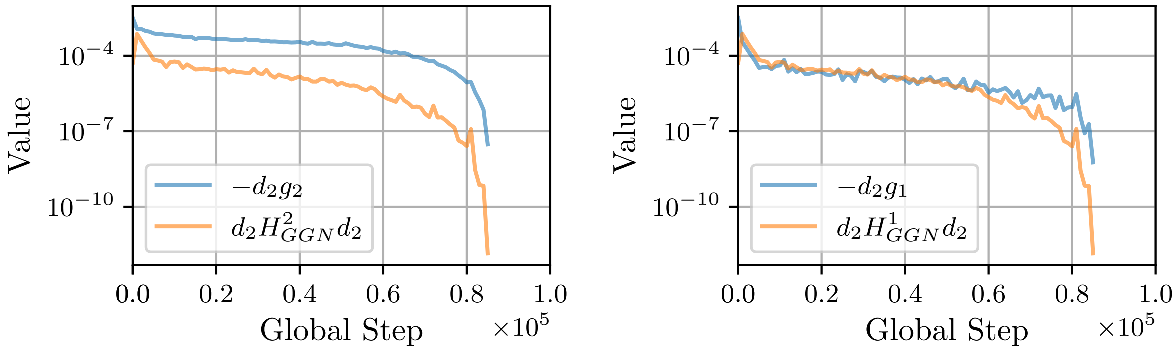
The denominators of α^*_{biased} and $\alpha^*_{\text{unbiased}}$ are highly correlated, with a Pearson correlation coefficient of 0.9998 when comparing individual denominator values. In contrast, the correlation between the numerators is significantly lower at 0.8407. The cosine similarity between d_2 and g_1 was -0.0063 on average, whereas the cosine similarity between d_2 and g_2 was more negative at -0.0751.

Our hybrid[4] analysis results in the following mean contributions of the denominator

---

[4]

$$\alpha_{\text{hybrid}} = \frac{\text{numerator}_{\text{unbiased}}}{\text{denominator}_{\text{biased}}} = \frac{d_2^T g_1}{d_2^T H_{\text{GGN}}^2 d_2}.$$



(a) Numerator and Denominator of Absolute Alpha Biased.

(b) Numerator and Denominator of Absolute Alpha Unbiased.

Figure 4.6: Comparison of Components of Absolute Alpha (Biased vs. Unbiased). Both plots show the the numerators and denominators of α^* and $\alpha^*_{unbiased}$. Each datapoint represents the average of the 50 values of the respective component that were computed at the start of a 1000-step interval.

and numerator to the difference between α^* and $\alpha^*_{\text{unbiased}}$:

$$\overline{\alpha_{\text{hybrid}} - \alpha^*_{\text{biased}}} = -1692.29 \quad \text{and} \quad \overline{\alpha^*_{\text{unbiased}} - \alpha_{\text{hybrid}}} = 21.77. \tag{4.1}$$

4.3.3 Performance Metrics

Next, we will present the performance metrics that were logged during the training run. Figure 4.7 shows the losses on the training, test, and validation splits. The target performances set by the AlgoPerf competition were not reached in our training run.

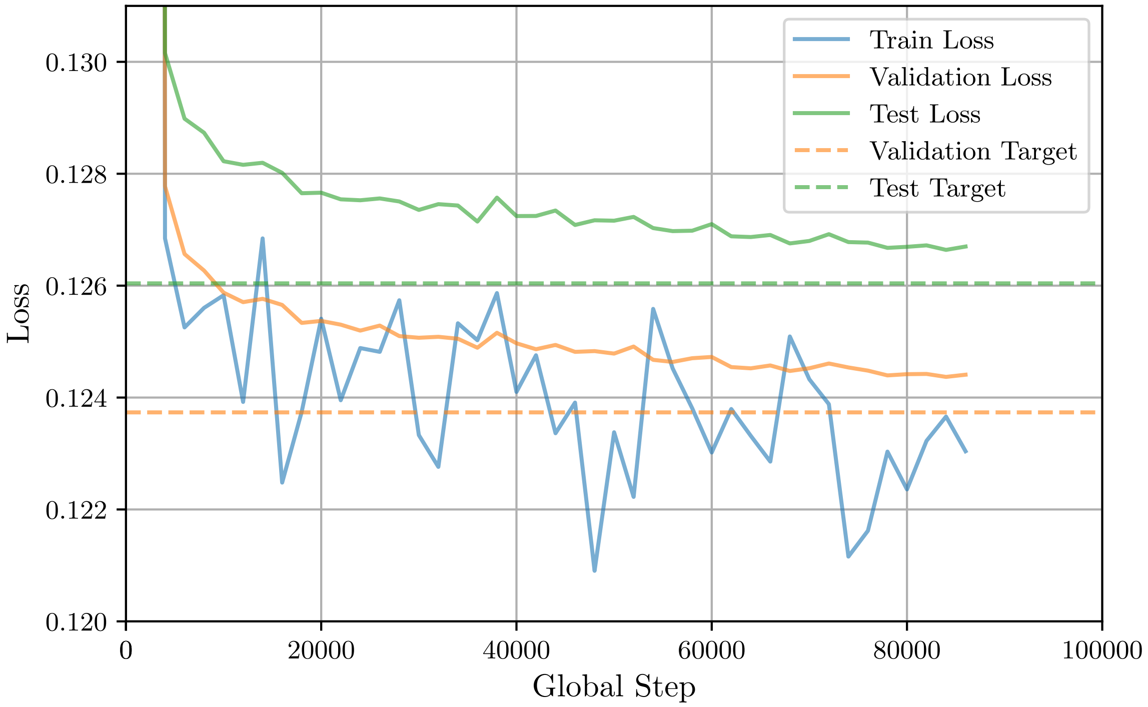


Figure 4.7: Losses on Training, Test, and Validation Split.

4.4 Discussion: Experiment 1

In this section, we discuss the results of the first experiment and their implications for the applicability of the α^*-step.

4.4.1 Extreme Outlier in the First Step

One unexpected observation is the extreme outlier in the first computed value of both relative α^*_{biased} and $\alpha^*_{unbiased}$, which exceeds 10^{14}. This is surprinsingly large. We can observe that due to the very small learning rate in the first step, the norm of the directional step was extremely small ($< 10^{-11}$), leading to denominators smaller than 10^{-29}. We do not observe any significant differences in the gradient norms or the loss values on the two batches between the first and the following steps. We do find that the cosine similarity between the gradient vectors and the directional step was very low at -0.00012 for both g_1 and g_2, compared to the average cosine similarity of -0.0063 and -0.0751 respectively, and that the norm of the directional step was significantly smaller than in the following steps at 3.76×10^{-12} compared to an average value of 0.1903. While we cannot pinpoint the exact cause of this phenomenon, we suspect that numerical instability may have contributed to this extreme outlier value, as per the extremely small magnitudes of learning rate and thus also the small norm of the directional step.

4.4.2 Relative Learning Rate Magnitudes

Our results indicate that both biased and unbiased α^*_{rel} suggest higher learning rates in the mean and median compared to the applied learning rate schedule. This implies that our curvature-based step size generally recommends increasing the learning rate most of the time.

The peaks observed at the beginning and end of the mean relative learning rates (see figure 4.3) indicate that neither biased nor unbiased α^* suggest a similar linear warm-up phase at the start or a decay towards zero at the end of the schedule. Additionally, we find a notable difference between α^*_{biased} and $\alpha^*_{unbiased}$: on average and in the median, α^*_{biased} suggests a much greater increase in the learning rate compared to $\alpha^*_{unbiased}$. However, the strong linear correlation (>0.8) between the two shows that despite differences in absolute values, they follow similar trends throughout training overall.

4.4.3 Absolute Magnitudes and Dispersion

The absolute values of α^*_{biased} and $\alpha^*_{unbiased}$ provide insight into the actual learning rates suggested by the α^*-step when derived from the mean or median. On average and in the median of all values, $\alpha^*_{unbiased}$ is within a factor of two of the mean and median learning rates of the applied schedule. Although this suggests a certain level of similarity in these statistical measures, this does not necessarily indicate a pattern in when to increase or decrease the learning rate.

A key observation is that $\alpha^*_{unbiased}$ displays large fluctuations between consecutive steps, as shown by the error bars in Figure 4.4. One contributing factor could be the parameter updates between steps, which slightly shift the location in parameter space where α^* is computed in the subsequent step. However, it is highly likely that the stochasticity of the mini-batch sampling process plays a significant role in these variations. Another important aspect when assessing the applicability of the $\alpha^*_{unbiased}$ step is that more than a quarter of the computed values were negative, indicating a direction of ascent rather than descent when applied. This could cause the optimizer to diverge from the minimum when applying the $\alpha^*_{unbiased}$ step.

The absolute values of α^*_{biased} differ significantly from this. Analyzing the mean and median, we find that α^*_{biased} exceeds both the learning rate schedule and $\alpha^*_{unbiased}$ by a large factor (over a 20-fold increase). This is also at display in Figure 4.5, where the learning rate schedule is constantly far below the average values of α^*_{biased} in terms of magnitude. When considering dispersion measures such as standard deviation and range across all values, means, and medians, α^*_{biased} shows a greater absolute dispersion compared to $\alpha_{unbiased}$. However, when adjusting for scale by computing the coefficient of variation (CV), we find that the relative dispersion in α^*_{biased} ($CV = 0.914$) is smaller compared to $\alpha^*_{unbiased}$ ($CV = 1.846$). This trend also holds true when comparing the relative standard deviation with respect to the range[5] (RSD), where $\alpha^*_{unbiased}$ scored significantly higher at $RSD_{unbiased} = 3.56$, compared to α^*_{biased} ($RSD_{biased} = 0.996$). Thus, we conclude that while the absolute dispersion of α^*_{biased} is significantly larger than that of $\alpha^*_{unbiased}$, this relationship reverses when considering relative dispersion due to the overall larger magnitudes of α^*_{biased}. Additionally, α^*_{biased} shows significantly larger magnitudes compared to the learning rate schedule or $\alpha^*_{unbiased}$, indicating that it could struggle in practice to find a suitable learning rate.

4.4.4 Trend in Absolute Biased Alpha

We also observe that, relative to their mean and median values, α^*_{biased} suggests lower learning rates at the beginning and towards the end of the learning rate schedule (see Figure 4.5). This trend is not as strong for unbiased α^*. A closer examination reveals that biased and unbiased α^* are more similar in the initial phase. Over the first 50 steps, the mean of absolute $\alpha^*_{unbiased}$, is 0.000268, compared to 0.000267 for α^*_{biased}.

---

[5]

$$\mathrm{RSD}_{\mathrm{range}} = \frac{\mathrm{SD}_{\alpha^*}}{\alpha^*_{\max} - \alpha^*_{\min}} \tag{4.2}$$

However, as training progresses, they begin to diverge. By steps 1000–1049, the mean values already become more distinct, with $\alpha^*_{unbiased}$ at 0.000104 and α^*_{biased} significantly higher at 0.001483.
This pattern of α^*_{biased} being closer to $\alpha^*_{unbiased}$ is also evident towards the end of the learning rate schedule, where α^*_{biased} reduces in magnitude and thereby approaches the relatively small values of $\alpha^*_{unbiased}$. Therefore, we find that at the beginning and end of the learning rate schedule, where very small learning rates are used, unbiased and biased α^* exhibit more similar magnitudes. However, throughout the rest of the training process, their values differ significantly.

4.4.5 Numerator and Denominator Contribution

Our analysis of the numerator and denominator (w.r.t. relative α^*) reveals that the difference between biased and unbiased α^* is primarily driven by the numerator. This is evident from the extremely high correlation between the denominators ($r > 0.999$), the comparatively lower correlation of the numerators ($r \approx 0.85$), and our hybrid analysis. Specifically, the hybrid analysis shows that the numerator accounts for more than 98% of the differences between α^*_{biased} and $\alpha^*_{unbiased}$ on average, while the denominator contributes less than 2% (see individual contributions in 4.1).
One possible explanation for the difference in the numerators could be the stronger opposite alignment of the gradient and the step direction of d_2 and g_2. We observe that the mean cosine similarity between d_2 and g_2 is more negative (-0.0751) compared to that between d_2 and g_1 (-0.0063). This is not surprising, as the step direction d_2 is partially based on the gradient g_2 in methods that incorporate the gradient in finding a descent direction such as NAdamW, and it could be the reason why the numerator ($-d^T g$) is significantly larger in α_{biased} compared to $\alpha_{unbiased}$.
In contrast, the difference in the denominator term, $d^\top H_{\text{GGN}} d$, is much more subtle. Although we cannot directly observe H_{GGN} due to our matrix-free multiplication approach, we can infer that H^1_{GGN} and H^2_{GGN} are probably highly similar within the same iteration, even though computed on different mini-batches. This is evident as the product $d_2^\top H^1_{\text{GGN}} d_2$ is very close to $d_2^\top H^2_{\text{GGN}} d_2$ in our data. Our optimization method NadamW does not directly access curvature when choosing a step direction, therefore the same reason as for the numerator should not systematically cause difference in the biased vs. unbiased denominator. Tatzel et al. (2024) identify that bias in curvature is small in the beginning and stronger in late training stages, i.e., after the accumulated data, which has been passed through the model in mini-batch training is a multiple of the size of the entire dataset. Due to the very large dataset ($> 4 \times 10^9$ training examples) we only passed ≈ 0.66 times the size of the training split through the model. This could be part of the reason why this bias had relatively little impact on our

fraction. While we are limited to analyzing differences in the denominators without access to eigenvectors and eigenvalues, we find that the debiasing approach impacts mainly the numerator of the fraction α^*.

4.4.6 Performance

Dahl et al. (2023) point out that testing a training algorithm, such as the baseline algorithm, can only define its performance for the specific instance of its hyperparameters and hardware. So, due to differences in hardware and hyperparameter settings (learning rate, schedule length, and batch size), a direct comparison to the AlgoPerf competition results is not possible. However, we did find that neither the test nor the validation loss targets set for the AlgoPerf competition were achieved in this training run. Several factors may have contributed to this outcome.

Firstly, we followed the learning rate schedule provided by the self-tuning AlgoPerf competition baseline (Dahl et al., 2023), adjusting for our smaller batch size. We found that after this adjustment, the targets were not reached within the scope of the total schedule. Therefore the adjustment in the learning rate schedule could have led to this result. The original baseline achieved the targets within 25% of its total time budget, indicating that there is a significantly worse performance in our adjusted implementation.

Secondly, we only used the provided learning rate schedule as the basis for our learning rate schedule and did not use the specific time limit from the AlgoPerf competition, due to the different hardware. In the self-tuning competition, the competitors were allowed to use a time budget three times larger compared to the external-tuning competition. Therefore, we cannot say exactly how the time limit would have transferred to our training run and if we could have reached the targets within the time limit.

Another explanation for this result could be that smaller batch sizes can influence training performance. Gradients that are computed on a mini-batch can be viewed as an estimation of the true gradient of the whole dataset. Therefore, the smaller the batch size, the noisier the gradient estimate (Goodfellow et al., 2016). While there are reports of faster convergence in terms of training time when using larger batch sizes and deploying GPU parallelization, see e.g. Goyal et al. (2017), there are also reports of better generalization when using smaller batch sizes, see e.g. Keskar et al. (2017). We adjusted the schedule length and learning rate according to effective scaling rules, but these changes may not have fully compensated for the difference in batch size. There are multiple other factors that could have contributed the performance of the training run, such as other hyperparameters that we did not change or interactions between them. For example, Smith (2018) suggests that an effective learning rate, which we adjusted, also has implications on the choice of the momentum parameter.

In conclusion, we find that the performance of the training run did not meet the targets set by the AlgoPerf competition. We see many possible reasons for this, such as the adjustment of the learning rate schedule, the smaller batch size or other hyperparameters that we did not change. Instead of a comparison to the AlgoPerf competition results, in the next experiment, we will interpret our findings by focusing on direct comparisons between trainings of our adjusted baseline algorithm, our approach to deploying our curvature-based step size α^*, and the winning algorithm of the AlgoPerf competition.

4.5 Experiment 2: Grid Search

In this experiment[6], we iteratively applied the median of absolute $\alpha^*_{unbiased}$ as the learning rate. We decided, that due to the relatively similar magnitude of $\alpha^*_{unbiased}$ compared to the learning rate of the learning rate schedule, it could possibly deliver applicable step sizes. Therefore, we performed a grid search varying the number of $\alpha^*_{unbiased}$ we computed in consecutive steps and the frequency thereof. To safeguard against outliers and for more stable learning rates, we applied the median of the computed $\alpha^*_{unbiased}$ values as the learning rate. For even numbers of values, we computed the median as the midpoint between the two median values.

	Number Computed		
Interval Size	10	25	50
1000	(1000, 10)	(1000, 25)	(1000, 50)
2000	(2000, 10)	(2000, 25)	(2000, 50)
5000	(5000, 10)	(5000, 25)	(5000, 50)

Table 4.2: Grid-Search Parameters. Testing various settings for the interval size at the beginning of which $\alpha^*_{\text{unbiased}}$ is initially computed and the number of consecutive $\alpha^*_{\text{unbiased}}$ computations in one interval.

We ran this grid search twice: once evaluating the performance every 1000 steps and once at a fixed time interval. For the time interval, we evaluated every 960 seconds or 8 times less often compared to the AlgoPerf competition to account for our different hardware. This enabled us to compare the performance in a given number of steps and the performance in a given time frame across all combinations. We also compared the performance of the grid search to the performance of the baseline algorithms. Table 4.2 shows the settings of the grid search. The dataset, model, and optimizer remained unchanged, only the interval size and the number of computed $\alpha^*_{unbiased}$ values were varied. This approach requires an initial learning rate: we used the learning rate

---

[6] The implementation of Experiment 2 can be found in the following folder: sub5_grid_search https://github.com/Davidundso/taylor/tree/main/algorithmic_efficiency/my_submissions/sub5_grid_search

schedule as in the first experiment for the first 2000 steps, then kept the learning rate constant during the computations of N consecutive $\alpha^*_{unbiased}$, and then used the median of the computed absolute α^* unbiased values as the learning rate until there is a new learning rate defined in the next interval. This utilizes the linear warm-up that the learning rate schedule provides approximately until its highest point. To ensure stable training, we only applied the median as the learning rate when it was positive. Otherwise we did not change the learning rate in that iteration (see Figure 4.8 for the algorithm). During training, we logged the same metrics as in Experiment 1. In the following, we will refer to the interval sizes as I and to the number of consecutively computed α^* values as N, whenever it aids readability.

```
Require: $I$ (Interval Size), $N$ (Number of Consecutively Computed $\alpha^*_{unbiased}$ Values)
while global step $< 2000$ do
    Perform normal update step with LR schedule
end while
while Max time or steps not reached do
    if (beginning of Interval $I$) then
        for $N$ Steps do
            Compute absolute $\alpha^*_{unbiased}$
            Perform normal update step
        end for
        Set learning rate to median of computed $\alpha^*_{unbiased}$ values (if positive)
    else
        Perform normal update step
    end if
end while
```

Figure 4.8: General Algorithm of Experiment 2. This is the general algorithmic structure of Experiment 2. The specific steps are not listed here, it is only a general outline.

4.6 Results: Experiment 2

In this section, we will present the results of the second experiment. First, we will present the results of the grid search which evaluated the performance every 1000 steps starting at step one. Then, we will present the results of the grid search which evaluated the performance at a set time interval.

4.6.1 Grid Search: Step-Based Comparison

None of the training runs of our grid search or our comparison baselines reached the target performance set by the AlgoPerf competition. We evaluated the performance of the grid search and baseline runs by comparing the best validation split loss within a given budget of steps. Figure 4.9 shows the best validation split loss for each combination of I and N for the budget of 85,328 steps, which is the length of the entire learning rate schedule. The best validation split loss was achieved with an interval size of 5000 steps and 10 consecutively computed values at 0.124372. This is 0.000086 worse than the AlgoPerf baseline, which achieved 0.124286 with a learning rate schedule. However, it outperforms Schedule-Free AdamW, which had a best validation split loss of 0.124606 (a 0.000234 increase compared to our approach).

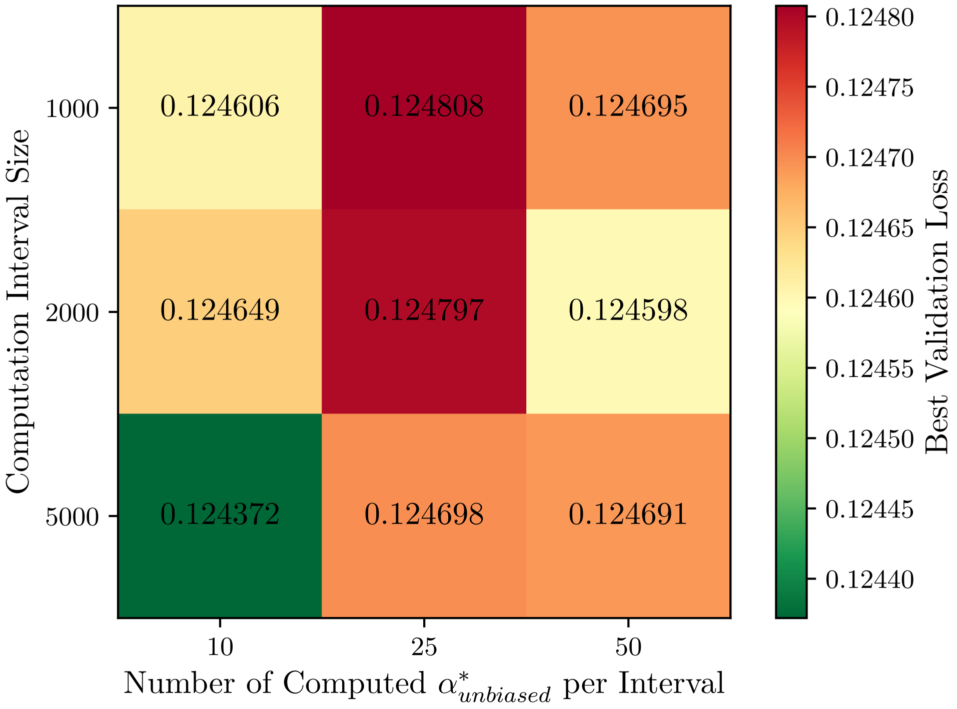


Figure 4.9: Best Validation Split Loss: Full Schedule. Shows the best validation split loss for each combination of interval size and number computed within the budget of the full schedule of 85,328 steps.

A Pearson correlation analysis was performed to investigate the linear relationship between both the interval size and the number of consecutively computed values and the best validation loss. The analysis resulted in a non-significant Pearson correlation of $r_{I,L} = -0.412$ (p = 0.270) for the relationship between interval size and best loss, and a non-significant Pearson correlation of $r_{N,L} = 0.303$ (p = 0.428) for the relationship between the number of computed values and best validation loss.

When considering half of the steps of the complete learning rate schedule, the lowest validation split loss across all combinations came from the combination of $I = 2000$ and $N = 50$. It is 0.124690, which is 0.000173 smaller compared to 0.124863 by the AlgoPerf baseline and 0.124951 by Schedule-Free AdamW with the same budget of steps. Again, there are no significant correlations between the interval size and the number of computed values and the best validation loss. See Figure B.1 in Appendix B

for the full heatmap of these results.
Figure 4.10 shows the best and worst performing configurations of the grid search. The best performing configuration (blue) had an interval size of 5000 and a number of consecutively computed values of 10. The worst (yellow) had an interval size of 1000 and a number of consecutively computed values of 25. The learning rate scheduled run (green) performed better than both other configurations. The mean learning rate applied in the best performing configuration of the grid search was 1.37×10^{-4}, that of the worst performing configuration was the smallest at 8.97×10^{-5}, whereas the better learning rate schedule had a mean learning rate of 1.09×10^{-4}. All mean and median applied learning rates are in Appendix B.

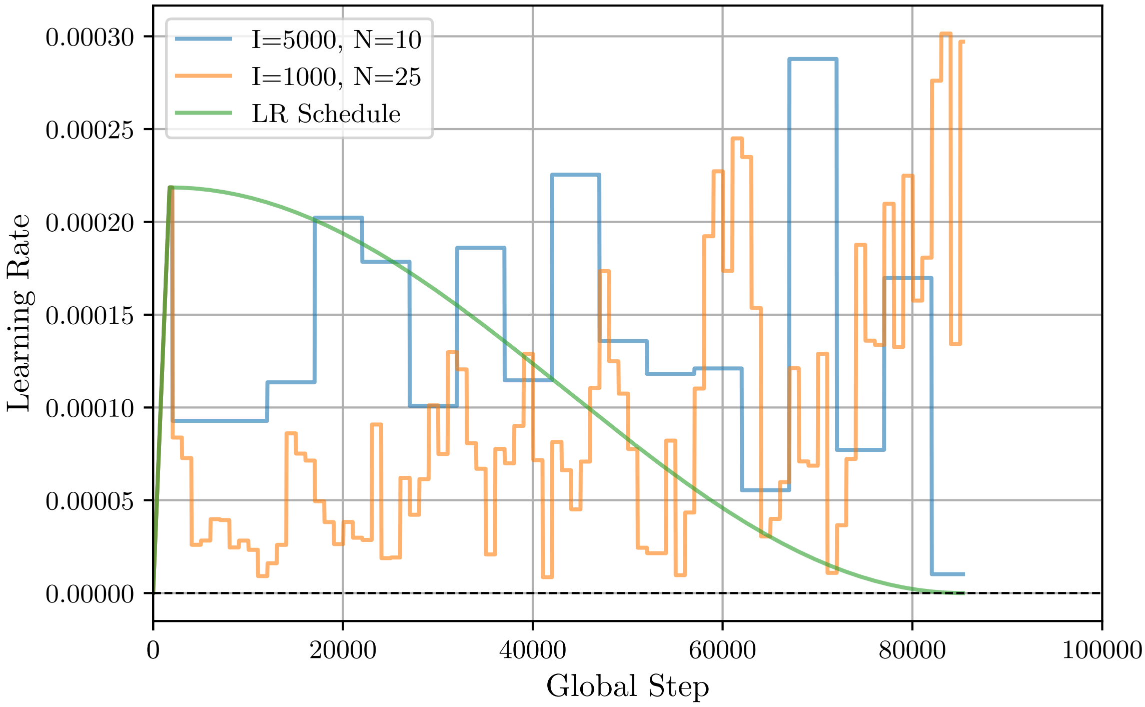


Figure 4.10: Applied Learning Rates and LR Schedule The learning rates applied in the best and worst performing configurations of the grid search and the learning rate schedule. I=Interval size, N= Number computed.

Next, we will briefly look at dispersion measures: Table 4.3 shows the ranges of applied learning rates in each run, only considering those learning rates which were derived from the median of the computed $\alpha^*_{unbiased}$ values. We find the largest dispersion in the configuration of $I = 1000$ and $N = 10$: The range of learning rates was 5.66×10^{-4}, which is significantly larger compared to the range of the learning rates in the learning rate schedule (range $= 2.19 \times 10^{-4}$) or the second largest range in the grid search (range $= 3.49 \times 10^{-4}$). The lowest dispersion can be found in the combination of 5000 consecutively computed values and an interval size of 50, with a range of 1.34×10^{-4}. These two candidates also score lowest and highest respectively for the standard deviations of their applied learning rates, which are listed in table B.3 in Appendix B.

Interval Size	Number Computed 10	25	50
1000	0.0005658	0.0002929	0.0002206
2000	0.0003491	0.0002875	0.0002782
5000	0.0002776	0.0001837	0.0001336
LR Schedule		0.0002186	

Table 4.3: Ranges of Applied Learning Rates. The ranges (maximum - minimum) of each run when only considering the learning rates applied by our method (excluding the linear warm-up phase)

4.6.2 Grid Search: Time-Based Comparison

Now, we will move on to the grid search, which focused on evaluating the performance at set time intervals, allowing us to compare the performance across different training runs within a given time budget. The baseline algorithm took 13,699.45 seconds to finish the learning rate schedule in this experiment. We used that time reference as our time budget and compared the best performances within the bugdet. Figure 4.11 shows the best validation split loss for each combination of the grid search within the time budget. The best validation split loss was achieved on the combination of $I = 5000$ and $N = 10$ with a best validation loss of 0.125151. That is 0.000813 higher compared to AlgoPerf baseline, which achieved a best validation loss of 0.124338 within the

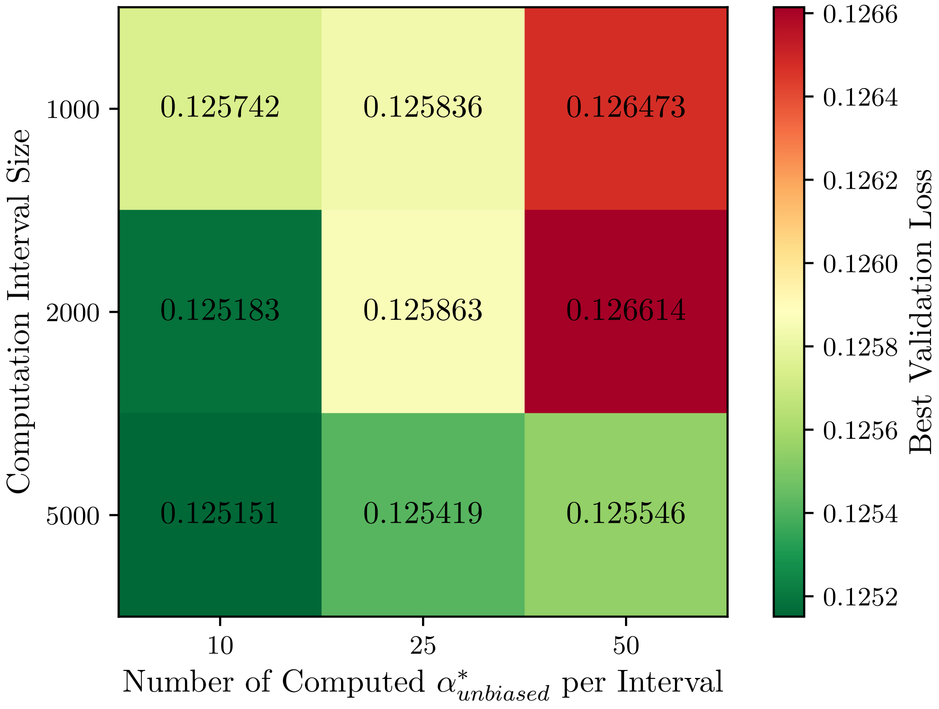


Figure 4.11: Best Validation Split Loss: Time of Full Schedule. Shows the best validation split loss for each combination of I and N within 13,699.45 seconds, or the time that elapsed until the baseline finished the schedule.

same time. The best validation loss of Schedule-Free AdamW was 0.124791 or 0.00036 lower compared to our best run. The Pearson correlation analysis resulted in a non-significant negative correlation between interval size and best validation loss of $r_{I,L} = -0.573$ (p = 0.107) and a significant correlation between number of computed values and best validation loss of $r_{N,L} = 0.721 (p = 0.028)$.
The Pearson correlation between the number of steps each run completed within the time budget and the validation loss was also significant at $r = -0.830$ (p=0.006). The highest number of steps within this time budget were 41,885 steps, achieved by the combination of an interval size of $I = 5000$ and $N = 10$. The lowest number of steps were 4,975, achieved by the combination of $I = 2000$ and $N = 50$. See table C.1 in Appendix B for the numbers of steps reached in each run.

We also compared the best and worst performing configurations of the grid search with a larger time budget. Note that this mainly allows us to compare the performance within the grid search and to Schedule-Free AdamW, as the baseline algorithm finishes its schedule early. The self-tuning competition had a time budget of $7,703 \cdot 3$ seconds and the external tuning had a budget of $7,703$ seconds, but we cannot directly scale this to our different hardware. We decided to additionally consider the time budget of $7,703 \cdot 8 = 61,624$ seconds. Figure 4.12 shows the best validation split loss for each combination of the grid search within that time budget. The best performance was

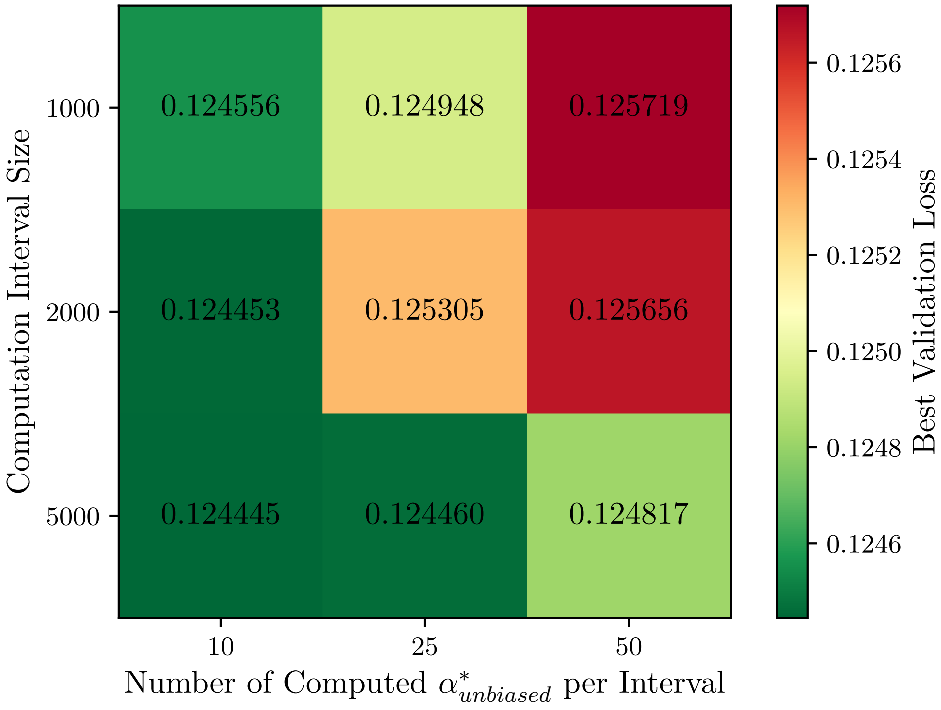


Figure 4.12: Best Validation Split Loss within 61,624 Seconds
Shows the best validation split loss for each combination of I and N within the 61,624 seconds budget.

again achieved by the combination of $I = 5000$ and $N = 10$ with a best validation loss of 0.124445, which is 0.000346 lower compared to 0.124791 by Schedule-Free AdamW within the same time budget. Schedule-Free AdamW did not improve compared to our lower time budget.

Again, there was a non-significant Pearson correlation between the interval size and the best validation loss ($r_{I,L} = -0.488, p = 0.183$), a significant correlation between the number of consecutively computed values per interval and the best validation loss ($r_{N,L} = 0.763, p = 0.017$) and a significant correlation between the number of steps completed within the time budget and the best validation loss ($r = -0.798, p = 0.010$). The highest number of steps was 207,009, achieved by the combination of an interval size of 5000 and 10 computed values, the lowest number of steps was 16,025, achieved by the combination of an interval size of 1000 and 50 computed values. See table C.2 in Appendix B for the number of steps reached in each run.

Figure 4.13 shows the learning rates applied in the best and worst performing configurations of the larger time budget and highlights the global step where the smaller and larger time budgets were reached. The best performing configuration had a mean learning rate of 2.64×10^{-4} (median $= 1.97 \times 10^{-4}$), which is higher than that of

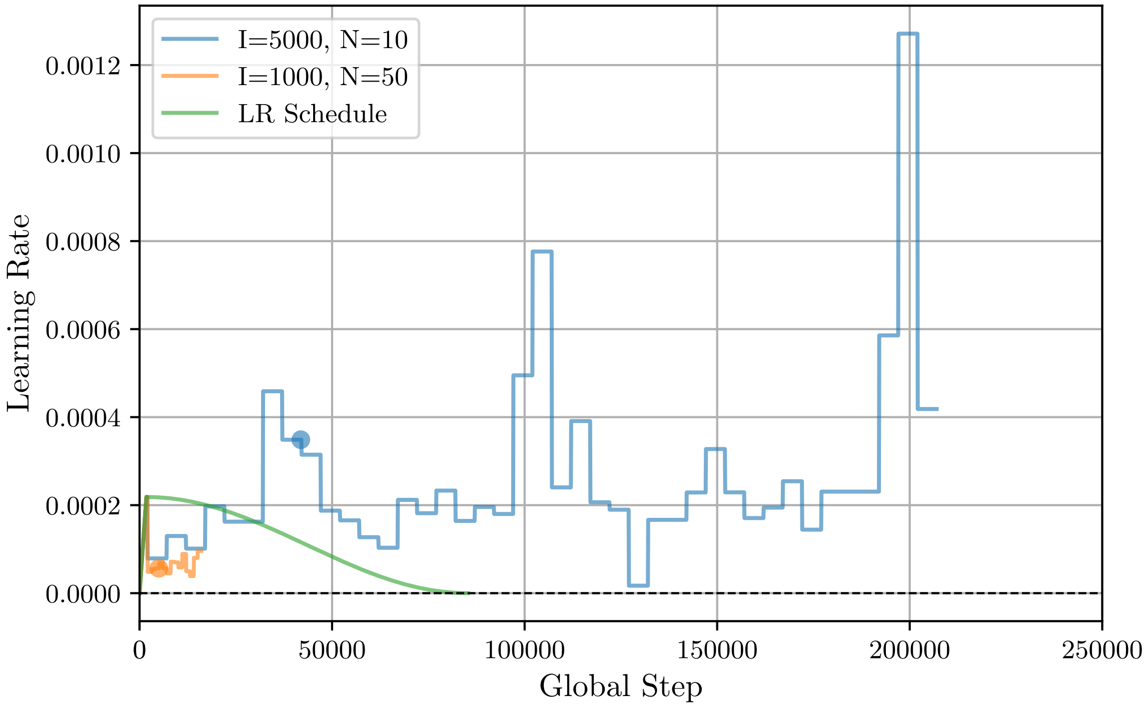


Figure 4.13: Learning Rates of Best and Worst Performing Configuration in 61,624 Seconds Budget.

Shows the learning rates applied in the best and worst performing configurations of the larger time budget grid search and the learning rate schedule. I=Interval size, N= Number computed. For the grid search configurations the dot of each line marks the global step where the smaller time budget of 13,699.45 seconds elapsed. Each line ends when the larger time budget of 61,624 seconds elapsed.

the worst performing configuration with a learning rate of 7.13×10^{-5} (median = 5.85×10^{-5}), and also higher than the learning rate schedule which had a mean learning rate of 1.09×10^{-4} (median = 1.09×10^{-4}).

Figure 4.14 shows the learning rates applied in all configurations with an interval size of 2000. The number of steps reached by each configuration varied strongly: for the larger time budget of 61,624 seconds, the configuration which computes 10 values within each interval reached 104,005, the one with 25 reached 26,006 steps and the one with 50 only 14,031 steps. As shown in Figure 4.12, ordering these trainings by the number of steps reached within the time budget results in the same order as ordering them by performance. All remaining learning rates applied in all runs of the larger and smaller time budget can be found in Appendix C.

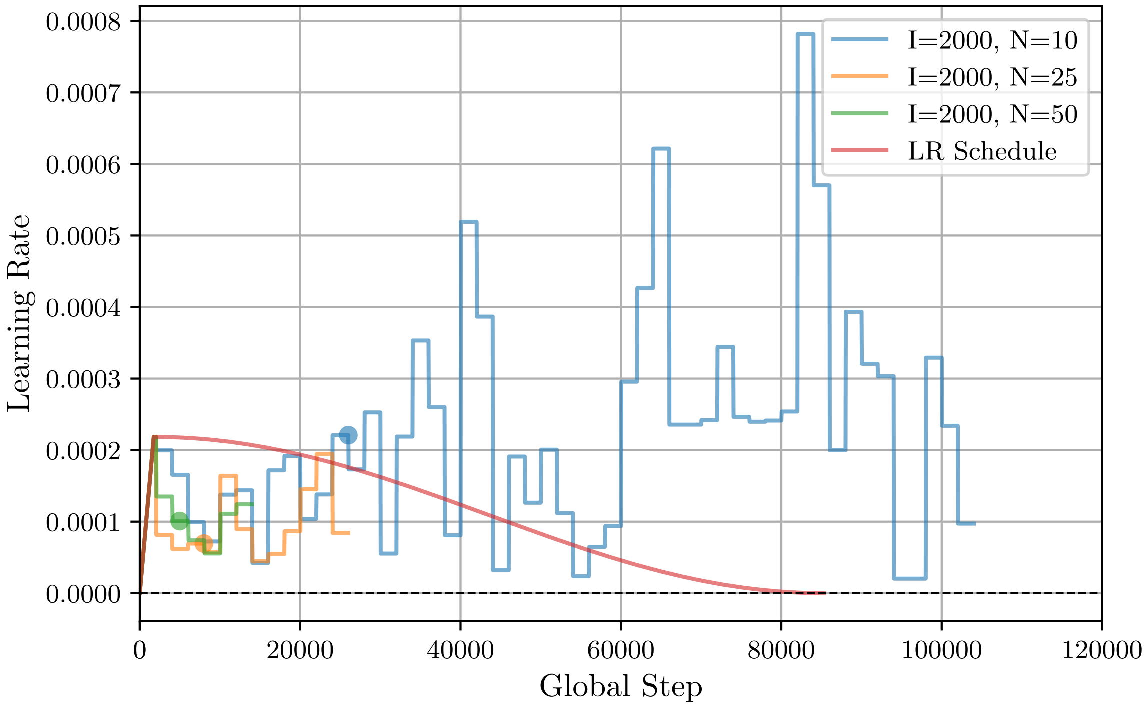


Figure 4.14: Learning Rates on 61,624 Seconds Budget.
Shows the learning rates applied in all configurations with an interval size of 2000. I=Interval size, N= Number computed. The dot of each line marks the global step where the smaller time budget of 13,699.45 seconds was reached. Each line ends when the larger time budget of 61,624 seconds was reached.

We already presented dispersion measures in the step-based comparison. We found no indication that dispersion in learning rates had a consistent effect on performance. Therefore, we will omit this analysis here. However, the ranges and standard deviations of the applied learning rates within the small time budget can be found in table C.3 and table C.4 and those for the large time budget in table C.5 and table C.6 in Appendix C.

4.7 Discussion: Experiment 2

In this section, we discuss the results of Experiment 2, focusing mainly on performance metrics. The aim of this experiment was to evaluate (1) whether our step-size approach is applicable and (2) how it performs in different configurations.

4.7.1 Step-Based Comparison: Baselines vs. Grid Search

The step-based comparison could reveal whether any of our configurations result in better performance based on a budget of steps, ignoring the extreme time-per-step differences across configurations. As in Experiment 1, the AlgoPerf targets for test and validation loss were not achieved in any of our runs. We suspect the same factors contributed to this as discussed in Experiment 1, and instead compare the best validation loss within the given step budget as an alternative reliable performance metric.

Our first takeaway is that none of our configurations outperformed the baseline adapted to the smaller batch size. Since both the baseline and our algorithm use the same optimization method and differ only in learning rates, we conclude that the baseline's schedule likely provides a more effective learning rate than our method. We compare nine different configurations of our method against a single baseline configuration. Therefore, if our method produced learning rates leading to similar performance, we would expect to see this in at least one configuration.

In the AlgoPerf competition, Schedule-Free AdamW performed identically to the baseline on this workload, both using 25% of the time budget to complete the Criteo1TB workload (MLCommons, 2024). However, in our training, Schedule-Free AdamW underperformed compared to the adapted baseline, indicating difficulties in transferring its performance to a smaller batch size. Although one of our configurations outperformed Schedule-Free AdamW within the step budget and modified batch size, we cannot conclude that it would do so with larger batch sizes or if hyperparameters Schedule-Free AdamW were adjusted to this different batchsize.

4.7.2 Step-Based Comparison: Grid Search

Figure 4.9 shows that performance varies across different configurations: The difference between the best- and worst-performing configurations is 4.36×10^{-4}, which is much larger than the difference between the best-performing configuration and our baseline (8.6×10^{-5}). However, when analyzing the heatmap, no clear trends emerge across our varying hyperparameters (interval size and number of consecutively computed α^* values). The correlation analysis supports this: neither interval size nor the number of consecutively computed α^* values show a significant linear relationship with performance. This remains true when considering only half of the step budget, indicating that our hyperparameters did not have a clear positive or negative impact on performance within these budgets.

Our dispersion measurements, such as the range of applied learning rates and standard deviation, show expected trends. The highest dispersion occurs in the configuration that changes the learning rate most frequently (interval size = 1000) and then applies the median of the smallest number of values (number of values = 10) as the learning rate. Conversely, the configuration with the largest interval size and number of consecutively computed α^* values has the lowest dispersion. This suggests that both taking the median from more consecutively computed α^* values and changing the learning rate less frequently decreases overall dispersion in the median values. These results align with our expectations for a stochastic process, such as repeatedly computing α^* on different batches. However, we must note that consecutive α^* values do not only vary due to the stochasticity in the mini-batch sampling process, but also due to the parameter updates in between subsequent α^* computations. We cannot separate these effects.

There are different ways of how these differences in dispersion of learning rates could affect the training performance: large dispersion in applied learning rates could be beneficial, if the learning rates adapt more effectively to the local loss curvature (i.e. reduce the loss more effectively in each iteration), whereas it could also induce instability or slow convergence when the learning rate often becomes extremely large or small for the local loss landscape (see section 2.4 or Goodfellow et al., 2016). We cannot find in our performance data that a lower or higher dispersion consistently resulted in better or worse training performance.

In conclusion of our step-based comparison, none of our training configurations were able to perform better than or equal to the baseline that we compared against. While we found that our hyperparameters have consistent effects on dispersion measures, we do not conclude that either higher or lower dispersion had a significant effect on performance in our data. The central finding of this comparison is that computing more α^* values to take the median from or computing and applying our curvature-based step size estimation more frequently did not have a clear effect on our performance measure when considering a certain budget of steps. Our data is therefore inconsistent with the notion that computing a more robust estimation of our curvature-based step size (i.e., computing more values in consecutive steps to take the median from) or adjusting to local curvature more frequently (i.e., computing the step size more frequently) yields better performance.

4.7.3 Time-Based Comparison: Baselines vs. Grid Search

For our time-based comparison, we used the time elapsed until the baseline algorithm completed its learning rate schedule as the time budget. This was our primary point of comparison, as time-to-result is the most critical metric for a deep learning algorithm (Dahl et al., 2023), whereas step-based comparisons ignore total training time.

We find that when considering time instead of steps, our configurations performed worse. The performance gap between our method and the adapted AlgoPerf baseline is larger than in the step-based comparison. Additionally, the Schedule-Free AdamW baseline performed slightly better than our training runs. This is likely due to the time-consuming calculation of α^*, as we will discuss in the within-grid search comparison.

4.7.4 Time-Based Comparison: Grid Search

For the smaller time budget, the configuration with an interval size of 5000 and 10 consecutively computed α^* values per interval yielded the best performance. The moderate negative correlation ($r = -0.573, p = 0.107$) between the interval size and the best validation loss shows that in our experiment, larger intervals tended to result in better performance. We also found that the significant correlation between the number of computed α^* values ($r = 0.721, p = 0.028$) and the best validation loss indicates that increasing this hyperparameter yields worse performance.

It is likely that these variables mostly impact performance through increasing or decreasing the average time per step: the highly significant negative correlation between the number of steps taken within the time budget and the validation loss ($r = -0.83, p < 0.01$) indicates that the number of steps taken within the given budget (or the average time per step) is the main contributor to performance in our different configurations. The same trend holds true for the much larger time budget: the best performance occurs in the configuration that achieved the most steps within the time budget, while the worst performances are found within the configurations which completed very few steps. Figure 4.13 and Figure 4.14 illustrate the extreme impact of computational cost: none of our configurations completed as many steps as the learning rate-scheduled optimizer, and configurations with smaller intervals and more α^* computations per interval completed only a fraction of the steps compared to those with opposite characteristics.

In conclusion, we find that the configuration with the largest interval size and the smallest number of consecutively computed α^* values performed best in both the larger and smaller time budget. Our analysis suggests that this is likely due to the smaller computational cost of computing a smaller number of α^* less frequently, which allows this configuration to complete more steps within a given time budget. This finding is consistent with the findings in our step-based comparison: if increasing the frequency or robustness of our curvature-based step-size estimator did not yield better performance within a certain number of steps, then the configuration which can complete the most steps within a time budget should yield the best performance within that budget.

4.7.5 Limitations

Next, we will discuss some of the limitations of our experimental design. The first and most obvious one has already been touched on before. We cannot compare any of our results directly to the AlgoPerf competition. This is because we used different hardware and adjusted the batch size and learning rate schedule of the baseline. The drawbacks of this approach were on display as both the AlgoPerf competition winner and the adjusted baseline algorithm performed significantly worse in our training runs. The difference in our setup also implies that we cannot use the same time budgets in our method as in the AlgoPerf competition and cannot scale them in a way that enables a direct comparison. For these reasons, we are also not able to compare our performance results directly to any other research. However, we do think that determining the time budget as the time that it took the adjusted baseline to finish its learning rate schedule does give us a relatively fair comparison within our experiments and holds some indication of how applicable and competitive our step size approach could be.

Secondly, we did incorporate the initial linear warm-up phase of the learning rate schedule into our implementation. We did this to ensure stability in the initial update steps, but it does mean that our method was not completely free of user-defined learning rates.

Thirdly, Dahl et al. (2023) point out that Criteo1TB is a data-pipeline-bound workload. This means that the time spent in preparation of batches exceeds the time that is used for computing the gradients and updating the weights. This impacts performance in a way that is not directly tied to the optimization algorithm.

Another observation is that the hardware we used through a computation cluster for the experiments seemed to vary in time-based performance: We find that both training runs that employed an interval size of 2000 and computed 25 and 50 consecutive α^* values per interval completed less steps within the smaller time budget than their counterparts with an interval size of 1000 (see table C.1). This is also true for the larger time budget (see tables C.1 and C.2). As the only difference in these algorithms is how often they employ expensive α^* computations, those implementations with the larger interval sizes should have completed more steps under fair conditions. We also find a drastic deviation in the time that it took the baseline algorithm to complete the learning rate schedule when evaluating every 1000 steps compared to every 16 minutes: The implementation which evaluated every 1000 steps (and therefore more often) averaged

≈ 0.29 seconds per step, whereas its time-based evaluating counterpart averaged ≈ 0.13 seconds per step. As evaluation time is explicitly not counted in our experiments, this fluctuation seems very significant. We suspect that the varying computation cluster performance and the more frequent training disruptions for evaluation might have contributed to this outcome. Therefore, we used the smaller time budget that the time-based evaluating training run took to complete the learning rate schedule as our time budget.

Additionally, we must point out that our experiments were not optimised for performance. We consistently computed and logged metrics which would not be necessary for the pure deployment of our approach, but had an impact on the time-based experiments. We did this to ensure that we can find meaningful data to analyse in the evaluation of our approach. Therefore, we must note that a strictly performance-based version of our experiments could yield significantly different results compared to our experiments.

Lastly, we must add to these inherent limitations within our experiments, there are also strong limitations regarding the generalization of our results. We only evaluated our method on a single dataset, model and optimizer. This is a very small sample size in general and also in the domain of training DNNs, as performance of methods across different datasets and models often varies. We acknowledge that our results cannot be generalized to other datasets, optimizers and models or be seen as a general proof of concept for our method.

5 Conclusion

5.1 Overall Conclusion

In conclusion, in our introduction and background we motivated the need for efficient learning rates in deep learning optimization, as well as progress and limitations in second-order optimization techniques. We derived a curvature-based step size, the quadratic-optimal step size along a direction, and integrated a method that approximates the Hessian (Generalized-Gauss-Newton Matrix). We also integrated a strategy to mitigate bias in slope, curvature and directional information in stochastic deep learning from Tatzel et al. (2024). We monitored the magnitude of these proposed step sizes during training with a validated optimizer and learning rate schedule and identified that the unbiased stepsize offered more promising magnitudes compared to the biased stepsize. We integrated this method into an existing optimizer, accounting for both the computational cost and the variance in our derived stepsizes through computing it at set intervals and estimating it with the median. Then we evaluated the proposed approach to assess its applicabilty and performance, comparing different variations of our proposed method. Now, we want to revisit our research questions to summarize our findings, which are limited to the scope of our experiments.

5.2 Research Questions

> *Does the proposed step size offer magnitudes that can be used for training deep neural networks, and, if so, how do stochasticity and inherent biases need to be considered when leveraging this curvature-based step size?*

Our results indicate that this method offers usable step size magnitudes for training deep neural networks when accounting for stochasticity by estimating the step size (e.g. through median) and accounting for inherent biases in the slope, curvature and directional information. These biases can be mitigated utilizing a debiasing strategy proposed by Tatzel et al. (2024). Specifically, this strategy involves computing directions on one and slope and curvature (approximations) on another mini-batch.

> *Can we implement this method in a way that balances computational cost to achieve competitive performance against state-of-the-art optimizers and learning rate schedules?*

In short: no. Our results suggest that the computational cost of our implementation is too high to be competitive against state-of-the-art optimizers and learning rate schedules. However, we were able to identify schemes that can make our method more competitive: Reducing the number of costly computations of our step size both by employing the same learning rate longer and by reducing the number of step sizes we compute to estimate our step size made our method more competitive. We also see potential for efficiency improvement in future implementations.

5.3 Future Work

The limitations in our approach (section 4.7.5) combined with the fact that the method did deliver usable step sizes in our setup motivates our suggestion concerning future work on this subject. While our approach did not yield competitive performance compared to a learning rate schedule, we do see potential in the general idea or variations of it. We propose that our approach should be widened to more datasets and optimizers to assess if it can be generalized to more settings. Furthermore, we encourage optimizing the algorithm as we see decent room for efficiency improvements. We also suggest that different variations, such as the exponential averaging schemes deployed in the similar approach by Balboni and Bacciu (2023) on large networks, could be a subject of further research in this matter. With the final objective of this method being to compete against learning rate schedules which are determined through grid searches, we also suggest to make this comparison in the future, if more efficient versions of this method are identified.

Bibliography

Agarwal, N., Bullins, B., & Hazan, E. (2017). Second-order stochastic optimization for machine learning in linear time. *J. Mach. Learn. Res.*, *18*, 116:1–116:40. https://jmlr.org/papers/v18/16-491.html

Amari, S. (1998). Natural gradient works efficiently in learning. *Neural Comput.*, *10*(2), 251–276. https://doi.org/10.1162/089976698300017746

Balboni, D., & Bacciu, D. (2023). Adler–an efficient hessian-based strategy for adaptive learning rate. *arXiv preprint arXiv:2305.16396*. https://doi.org/10.48550/arXiv.2305.16396

Barzilai, J., & Borwein, J. M. (1988). Two-point step size gradient methods. *IMA journal of numerical analysis*, *8*(1), 141–148.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*(10), 281–305. http://jmlr.org/papers/v13/bergstra12a.html

Bosch, N., Grosse, J., Hennig, P., Kristiadi, A., Pförtner, M., Schmidt, J., Schneider, F., Tatzel, L., & Wenger, J. (2022). *Numerics of machine learning* (tech. rep. No. 12). Tübingen AI Center.

Bottou, L. (1991). Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, *91*(8), 12. http://leon.bottou.org/papers/bottou-91c

Broyden, C. G. (1970). The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, *6*(1), 76–90. https://doi.org/10.1093/imamat/6.1.76

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2019). On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446*. https://doi.org/10.48550/arXiv.1910.05446

Criteo A. I. Lab. (2014). Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

Dahl, G. E., Sainath, T. N., & Hinton, G. E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 8609–8613. https://doi.org/10.1109/ICASSP.2013.6639346

Dahl, G. E., Schneider, F., Nado, Z., Agarwal, N., Sastry, C. S., Hennig, P., Medapati, S., Eschenhagen, R., Kasimbeg, P., Suo, D., Bae, J., Gilmer, J., Peirson, A. L., Khan, B., Anil, R., Rabbat, M., Krishnan, S., Snider, D., Amid, E., . . . Mattson, P. (2023). Benchmarking Neural Network Training Algorithms.

Dangel, F., Eschenhagen, R., Ormaniec, W., Fernandez, A., Tatzel, L., & Kristiadi, A. (2025). Position: Curvature matrices should be democratized via linear operators. *arXiv preprint arXiv:2501.19183*. https://doi.org/10.48550/arXiv.2501.19183

Dangel, F., Tatzel, L., & Hennig, P. (2022). ViViT: Curvature access through the generalized gauss-newton's low-rank structure. *Transactions on Machine Learning Research (TMLR)*.

Davidon, W. C. (1991). Variable metric method for minimization. *SIAM Journal on Optimization*, *1*(1), 1–17. https://doi.org/10.1137/0801001

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q., & Ng, A. (2012). Large scale distributed deep networks. In F. Pereira, C. Burges, L. Bottou, & K. Weinberger (Eds.), *Advances in neural information processing systems* (Vol. 25). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

Defazio, A., Yang, X., Mehta, H., Mishchenko, K., Khaled, A., & Cutkosky, A. (2024). The road less scheduled. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, & C. Zhang (Eds.), *Advances in neural information processing systems* (pp. 9974–10007, Vol. 37). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2024/file/136b9a13861308c8948cd308ccd02658-Paper-Conference.pdf

Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

Dozat, T. (2016). Incorporating nesterov momentum into adam.

Duchi, J. C., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, *12*, 2121–2159. https://doi.org/10.5555/1953048.2021068

Eschenhagen, R., Immer, A., Turner, R., Schneider, F., & Hennig, P. (2023). Kronecker-factored approximate curvature for modern neural network architectures. *Advances in Neural Information Processing Systems*, *36*, 33624–33655. https://doi.org/10.48550/arXiv.2311.00636

Feurer, M., & Hutter, F. (2019). Hyperparameter optimization. In F. Hutter, L. Kotthoff, & J. Vanschoren (Eds.), *Automated machine learning: Methods, systems, challenges* (pp. 3–33). Springer International Publishing. https://doi.org/10.1007/978-3-030-05318-5_1

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Philosophical transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, *222*(594-604), 309–368.

Fletcher, R. (1970). A new approach to variable metric algorithms. *The computer journal*, *13*(3), 317–322.

Fletcher, R., & Powell, M. J. (1963). A rapidly convergent descent method for minimization. *The computer journal*, *6*(2), 163–168.

Goldfarb, D. (1970). A family of variable-metric methods derived by variational means. *Mathematics of computation*, *24*(109), 23–26.

Goldstein, A. A. (1965). On steepest descent. *Journal of the Society for Industrial and Applied Mathematics, Series A: Control*, *3*(1), 147–151.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). MIT press Cambridge.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., & He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*. https://doi.org/10.48550/arXiv.1706.02677

Grosse, R., & Martens, J. (2016). A kronecker-factored approximate fisher matrix for convolution layers. *International Conference on Machine Learning*, 573–582. https://doi.org/10.48550/arXiv.1602.01407

Hestenes, M. R., Stiefel, E., et al. (1952). Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, *49*(6), 409–436.

Hinton, G., Srivastava, N., & Swersky, K. (2012). Neural networks for machine learning. *Coursera, video lectures*, *264*(1), 2146–2153.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.*, *6*(2), 107–116. https://doi.org/10.1142/S0218488598000094

Kalra, D. S., & Barkeshli, M. (2025). Why warmup the learning rate? underlying mechanisms and improvements. *Advances in Neural Information Processing Systems*, *37*, 111760–111801.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=H1oyRlYgg

Kiefer, J. (1953). Sequential minimax search for a maximum. *Proceedings of the American mathematical society*, *4*(3), 502–506.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In Y. Bengio & Y. LeCun (Eds.), *3rd international conference on learning representations, ICLR 2015, san diego, ca, usa, may 7-9, 2015, conference track proceedings*. http://arxiv.org/abs/1412.6980

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, *2*(2), 164–168.

Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Math. Program.*, *45*(1-3), 503–528. https://doi.org/10.1007/BF01589116

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., & Han, J. (2019). On the variance of the adaptive learning rate and beyond. *CoRR*, *abs/1908.03265*. http://arxiv.org/abs/1908.03265

Loshchilov, I., & Hutter, F. (2017a). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*. https://doi.org/10.48550/arXiv.1711.05101

Loshchilov, I., & Hutter, F. (2017b). SGDR: stochastic gradient descent with warm restarts. *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=Skq89Scxx

MacKay, D. J. C. (1992). The evidence framework applied to classification networks. *Neural Computation*, *4*(5), 720–736. https://doi.org/10.1162/neco.1992.4.5.720

Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, *11*(2), 431–441. https://doi.org/10.1137/0111030

Martens, J. (2010). Deep learning via hessian-free optimization. *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 735–742.

Martens, J. (2020). New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, *21*(146), 1–76. http://jmlr.org/papers/v21/17-678.html

Martens, J., Ba, J., & Johnson, M. (2018). Kronecker-factored curvature approximations for recurrent neural networks. *International Conference on Learning Representations*.

Martens, J., & Grosse, R. (2015, July). Optimizing neural networks with kronecker-factored approximate curvature. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (pp. 2408–2417, Vol. 37). PMLR. https://proceedings.mlr.press/v37/martens15.html

Martens, J., & Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. *Proceedings of the 28th international conference on machine learning (ICML-11)*, 1033–1040.

Martens, J., & Sutskever, I. (2012). Training deep and recurrent networks with hessian-free optimization. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade: Second edition* (pp. 479–535). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_27

MLCommons. (2024). Algoperf: Training algorithms benchmark results [Accessed: 2025-03-08]. https://mlcommons.org/benchmarks/algorithms/

Moreau, T., Massias, M., Gramfort, A., Ablin, P., Bannier, P.-A., Charlier, B., Dagréou, M., Dupre la Tour, T., DURIF, G., Dantas, C. F., Klopfenstein, Q., Larsson, J., Lai, E., Lefort, T., Malézieux, B., MOUFAD, B., Nguyen, B. T., Rakotomamonjy, A., Ramzi, Z., … Vaiter, S. (2022). Benchopt: Reproducible, efficient and collaborative optimization benchmarks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in neural information processing systems* (pp. 25404–25421, Vol. 35). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2022/file/a30769d9b62c9b94b72e21e0ca73f338-Paper-Conference.pdf

Mutschler, M., & Zell, A. (2020). Parabolic approximation line search for dnns. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 5405–5416, Vol. 33). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2020/file/3a30be93eb45566a90f4e95ee72a089a-Paper.pdf

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. (2019). Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*. https://doi.org/10.48550/arXiv.1906.00091

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). *Dokl. Akad. Nauk. SSSR*, *269*(3), 543.

Nocedal, J., & Wright, S. J. (1999). *Numerical optimization*. Springer. https://doi.org/10.1007/978-0-387-40065-5

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, *28*, 1310–1318. http://proceedings.mlr.press/v28/pascanu13.html

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S.

(2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 32). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural computation*, *6*(1), 147–160.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, *4*(5), 1–17. https://doi.org/10.1016/0041-5553(64)90137-5

Robles-Kelly, A., & Nazari, A. (2019). Incorporating the barzilai-borwein adaptive step size into sugradient methods for deep network training. *2019 Digital Image Computing: Techniques and Applications (DICTA)*, 1–6. https://doi.org/10.1109/DICTA47822.2019.8945980

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386. https://psycnet.apa.org/doi/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Schmidt, R. M., Schneider, F., & Hennig, P. (2021, July). Descending through a crowded valley - benchmarking deep learning optimizers. In M. Meila & T. Zhang (Eds.), *Proceedings of the 38th international conference on machine learning* (pp. 9367–9376, Vol. 139). PMLR. https://proceedings.mlr.press/v139/schmidt21a.html

Schneider, F., et al. (2025). *Mlcommons™ algoperf: Technical documentation & faqs* [Retrieved March 17, 2025]. https://github.com/mlcommons/algorithmic-efficiency/blob/main/docs/DOCUMENTATION.md

Schneider, F., Balles, L., & Hennig, P. (2019). DeepOBS: A deep learning optimizer benchmark suite. *International Conference on Learning Representations*. https://openreview.net/forum?id=rJg6ssC5Y7

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, *14*(7), 1723–1738. https://doi.org/10.1162/08997660260028683

Shanno, D. F. (1970). Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, *24*(111), 647–656. https://doi.org/10.2307/2004840

Smith, L. N. (2017). Cyclical learning rates for training neural networks. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 464–472. https://doi.org/10.1109/WACV.2017.58

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*. https://doi.org/10.48550/arXiv.1803.09820

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. Burges, L. Bottou, & K. Weinberger (Eds.), *Advances in neural information processing systems* (Vol. 25). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., & Adams, R. (2015, July). Scalable bayesian optimization using deep neural networks. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (pp. 2171–2180, Vol. 37). PMLR. https://proceedings.mlr.press/v37/snoek15.html

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning*, 1139–1147. https://doi.org/10.1109/5.726791

Tatzel, L., Mucsányi, B., Hackel, O., & Hennig, P. (2024). Debiasing mini-batch quadratics for applications in deep learning. *arXiv preprint arXiv:2410.14325*. https://doi.org/10.48550/arXiv.2410.14325

Wolfe, P. (1969). Convergence conditions for ascent methods. *SIAM review*, *11*(2), 226–235. https://doi.org/10.1137/1011036

Yang, M., Xu, D., Wen, Z., Chen, M., & Xu, P. (2022). Sketch-based empirical natural gradient methods for deep learning. *Journal of Scientific Computing*, *92*(3), 94. https://doi.org/10.1007/s10915-022-01911-x

Yao, Z., Gholami, A., Shen, S., Mustafa, M., Keutzer, K., & Mahoney, M. (2021). Adahessian: An adaptive second order optimizer for machine learning. *proceedings of the AAAI conference on artificial intelligence*, *35*(12), 10665–10673. https://doi.org/10.1609/aaai.v35i12.17275

Ying, X. (2019). An overview of overfitting and its solutions. *Journal of physics: Conference series*, *1168*, 022022. https://doi.org/10.1088/1742-6596/1168/2/022022

Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Appendix

Appendix

A Erklärung zum Einsatz von KI **65**

B Additional Material Exp. 2a **66**

C Additional Material Exp. 2b **69**

D Environment and Versions **72**

A Erklärung zum Einsatz von KI

In folgendem Umfang wurde KI eingesetzt :

- Korrektur Rechtschreibung und Grammatik: Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

- Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

Diese Erklärung basiert auf der neuen Selbstständigkeitserklärung, die zum Zeitpunkt der Anmeldung dieser Arbeit noch nicht gültig war und deshalb freiwillig vorgenommen wird.

AI Tools
ChatGPT
Gemini
GitHub Copilot

Table A.1: Liste der verwendeten AI tools

B Additional Material Exp. 2a

Interval Size	**Number Computed** 10	 25	 50
1000	0.000225	0.0000897	0.0001303
2000	0.0001312	0.0001181	0.0001781
5000	0.0001365	0.0001476	0.0001146
LR Schedule		0.0001093	

Table B.1: Mean Applied Learning Rates. Showing the mean learning rates for different interval sizes and number of computations for the full schedule of 85,328 steps.

Interval Size	**Number Computed** 10	 25	 50
1000	0.0001999	0.0000723	0.0001243
2000	0.0000974	0.0001059	0.0001670
5000	0.0001181	0.0001522	0.0001212
LR Schedule		0.0001093	

Table B.2: Median Applied Learning Rates. Showing the median learning rates for different interval sizes and number of computations for the full schedule of 85,328 steps.

Interval Size	**Number Computed** 10	 25	 50
1000	0.0001234	0.0000682	0.0000479
2000	0.0000897	0.0000622	0.0000681
5000	0.0000638	0.0000516	0.0000341
LR Schedule		0.0000770	

Table B.3: Standard Deviations of Learning Rates. The standard deviations of learning rates for different interval sizes and number of computations, when only considering the learning rates applied by our method (excluding the linear warm-up phase) for the full schedule of 85,328 steps.

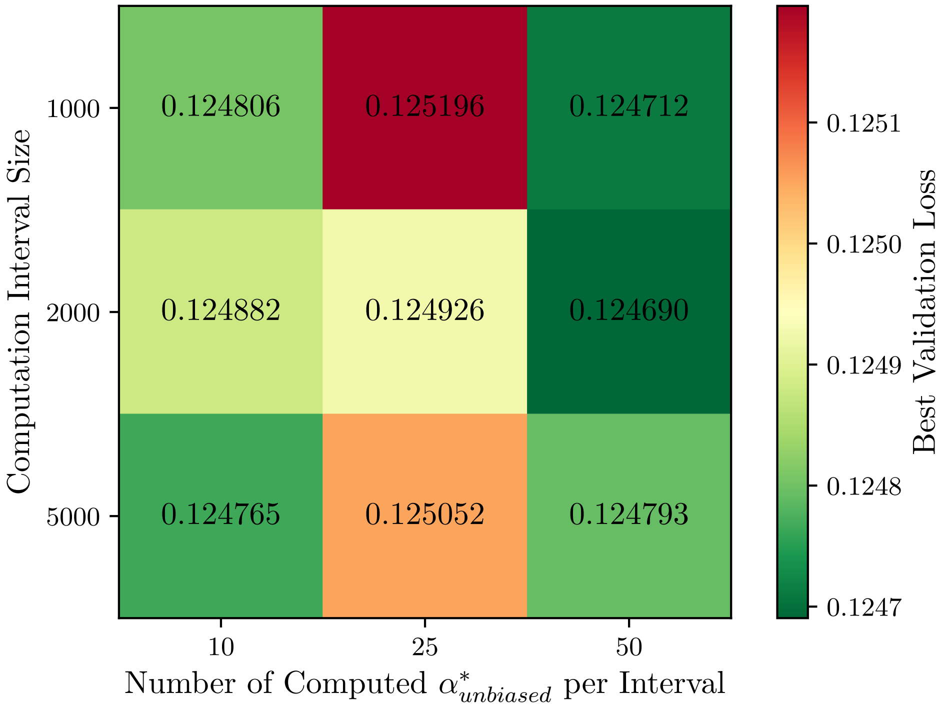


Figure B.1: Best Validation Split Loss: Half Schedule. Shows the best validation split loss for each combination of interval size and number computed within the budget of the half of the full schedule of 85,328 steps.

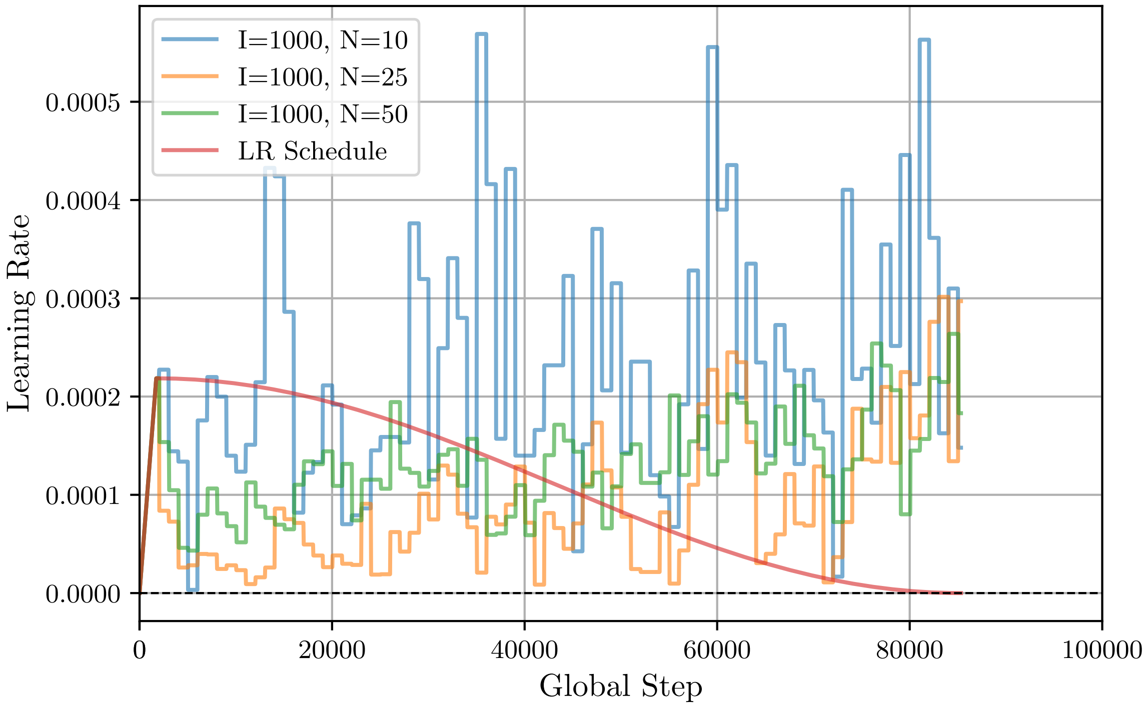


Figure B.2: Applied Learning Rates During Grid Search vs. LR Schedule Shows the applied learning rates during the grid search for the full schedule of 85,328 steps. I indicates the size of the interval, and N the number of consecutive $\alpha^*_{unbiased}$computed within the interval.

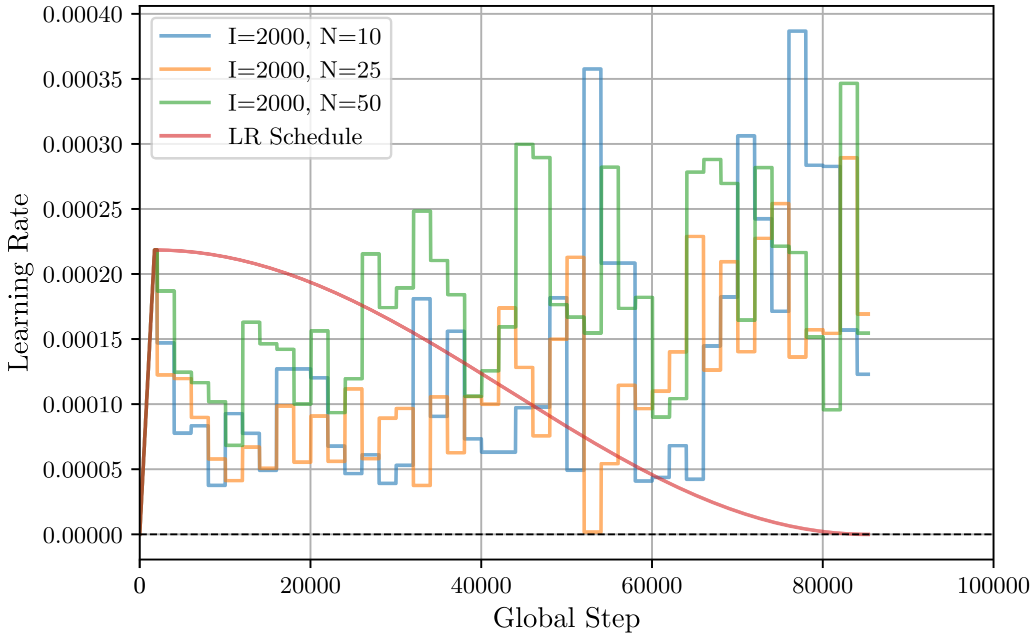


Figure B.3: Applied Learning Rates During Grid Search vs. LR Schedule Shows the applied learning rates during the grid search for the full schedule of 85,328 steps. I indicates the size of the interval, and N the number of consecutive $\alpha^*_{unbiased}$computed within the interval.

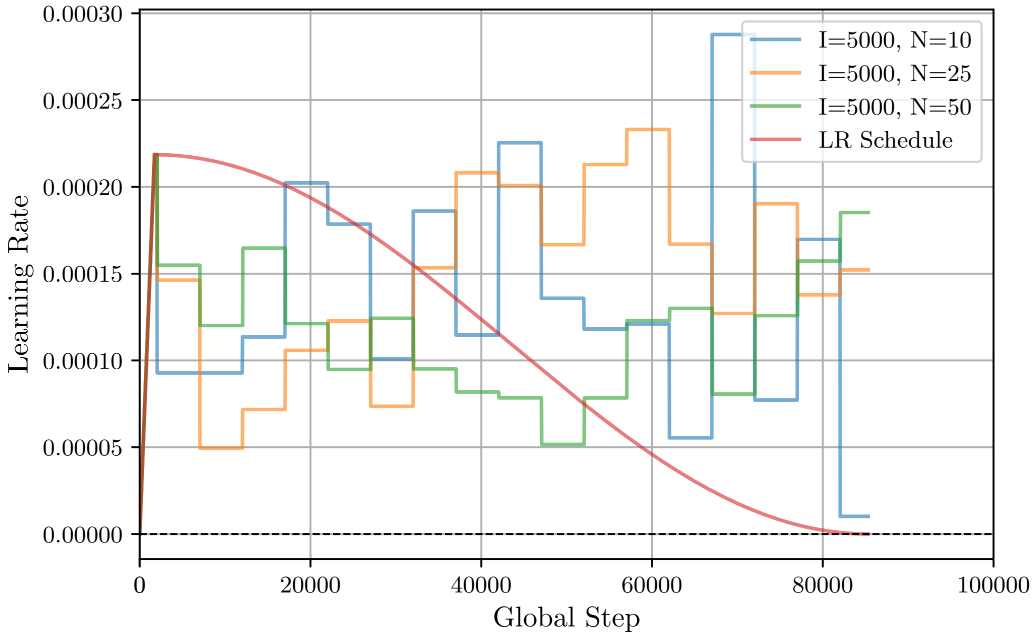


Figure B.4: Applied Learning Rates During Grid Search vs. LR Schedule Shows the applied learning rates during the grid search for the full schedule of 85,328 steps. I indicates the size of the interval, and N the number of consecutive $\alpha^*_{unbiased}$computed within the interval.

C Additional Material Exp. 2b

Interval Size	**Number Computed** 10	 25	 50
1000	16,789.0	9,726	5,005
2000	26,009	8,002	4,975
5000	41,885.0	22,023	12,032
LR Schedule	85,328		

Table C.1: Steps within Time of LR Schedule (13,699 Seconds Budget).

Interval Size	**Number Computed** 10	 25	 50
1000	72,002	38,005	16,025
2000	104,005	26,006	14,031
5000	207,009	127,005	57,005

Table C.2: Steps within 61,624 Seconds

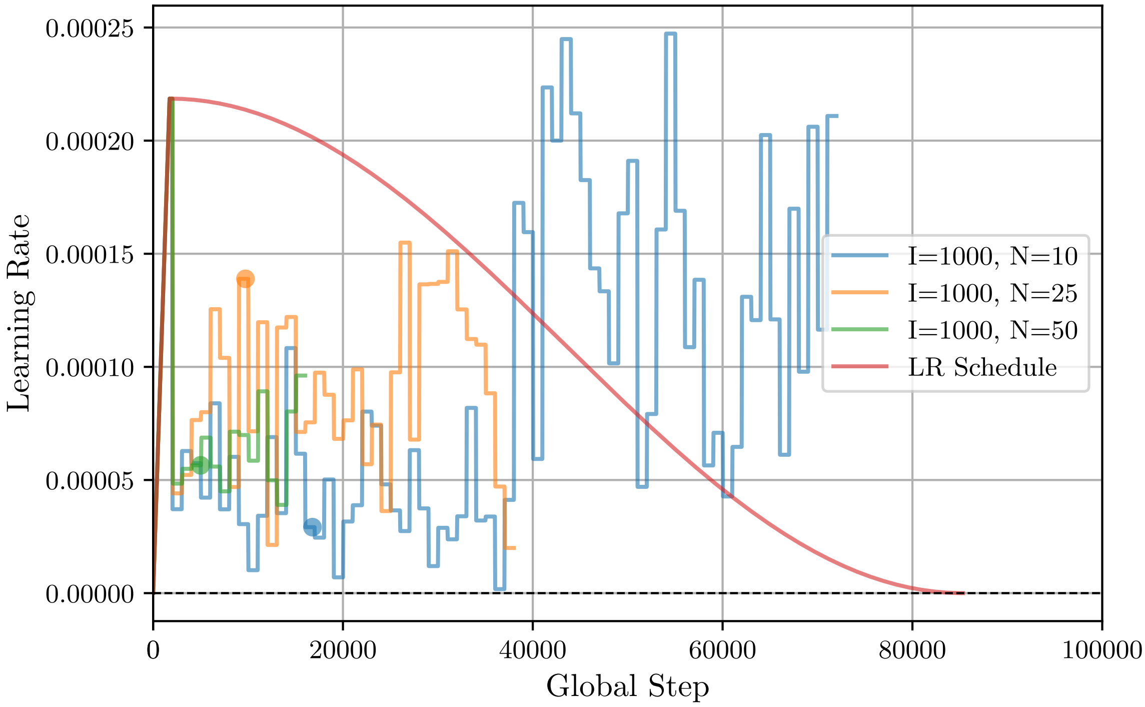


Figure C.1: Learning Rates on 61,624 Seconds Budget.
Shows the learning rates applied in all configurations with an interval size of 1000. I=Interval size, N= number computed. The dot of each line marks the global step where the smaller time budget of 13,699.45 seconds was reached. Each line ends when the larger time budget of 61,624 seconds was reached.

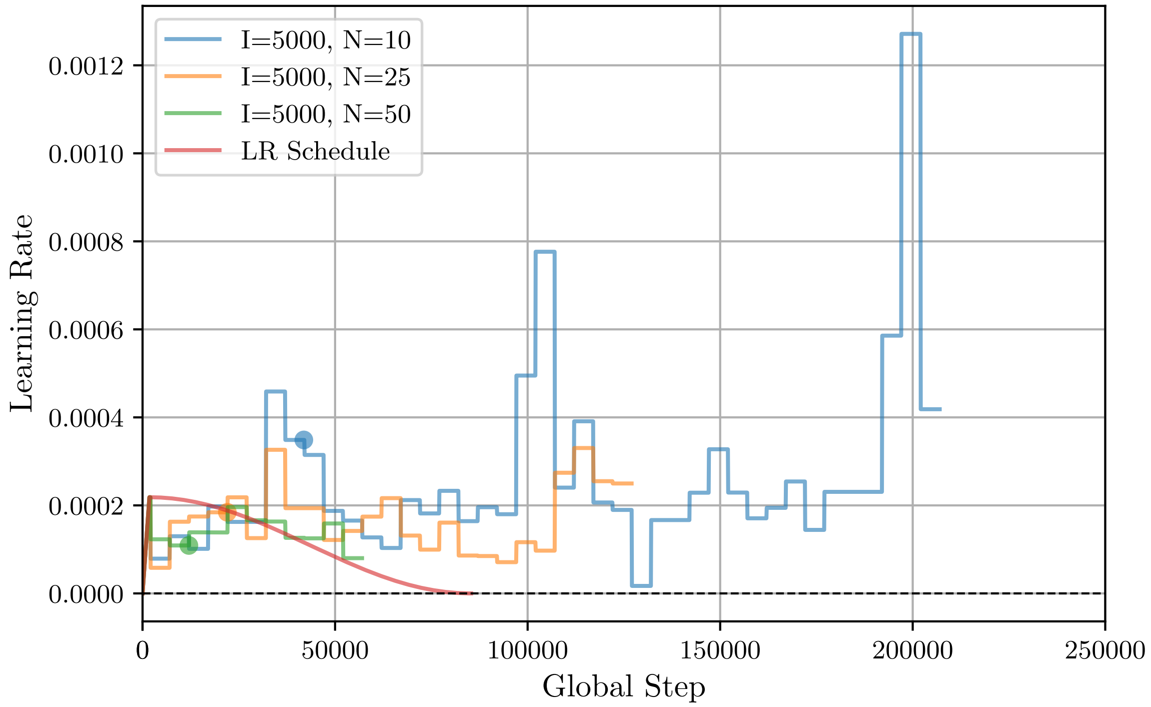


Figure C.2: Learning Rates on 61,624 Seconds Budget.
Shows the learning rates applied in all configurations with an interval size of 5000. The dot of each line marks the global step where the smaller time budget was reached, the lines end when the larger time budget was reached.

	Number Computed		
Interval Size	10	25	50
1000	2.40×10^{-5}	3.27×10^{-5}	3.51×10^{-6}
2000	5.13×10^{-5}	8.16×10^{-6}	1.59×10^{-5}
5000	1.23×10^{-4}	5.07×10^{-5}	6.99×10^{-6}

Table C.3: Standard Deviations of LR's in 13,699 Seconds Budget. The table presents the standard deviations of applied learning rates for different interval sizes and numbers of computations within the smaller time budget. The linear warm-up phase is excluded.

	Number Computed		
Interval Size	10	25	50
1000	9.82×10^{-5}	9.48×10^{-5}	8.07×10^{-6}
2000	1.79×10^{-4}	1.98×10^{-5}	3.43×10^{-5}
5000	3.80×10^{-4}	1.26×10^{-4}	1.40×10^{-5}

Table C.4: Range Values of LR's in 13,699 Seconds Budget. The table presents the ranges of applied learning rates for different interval sizes and numbers of computations within the smaller time budget. The linear warm-up phase is excluded.

Interval Size	Number Computed 10	 25	 50
1000	6.61×10^{-5}	3.60×10^{-5}	1.62×10^{-5}
2000	1.56×10^{-4}	4.56×10^{-5}	2.77×10^{-5}
5000	2.12×10^{-4}	7.40×10^{-5}	3.01×10^{-5}

Table C.5: Standard Deviations of LR's in 61,624 Seconds Budget. The table presents the standard deviations of applied learning rates for different interval sizes and numbers of computations within the larger time budget. The linear warm-up phase is excluded.

Interval Size	Number Computed 10	 25	 50
1000	2.46×10^{-4}	1.35×10^{-4}	5.72×10^{-5}
2000	7.61×10^{-4}	1.50×10^{-4}	7.96×10^{-5}
5000	1.25×10^{-3}	2.72×10^{-4}	1.16×10^{-4}

Table C.6: Range Values of LR's in 61,624 Seconds Budget. The table presents the ranges of applied learning rates for different interval sizes and numbers of computations within the larger time budget. The linear warm-up phase is excluded.

D Environment and Versions

Name	**Version**	**Name**	**Version**
_libgcc_mutex	0.1	_openmp_mutex	4.5
absl-py	1.4.0	algorithmic-efficiency	0.1.0
array-record	0.4.0	astunparse	1.6.3
backpack-for-pytorch	1.6.0	bzip2	1.0.8
ca-certificates	2024.8.30	cached-property	2.0.1
cachetools	5.5.0	certifi	2024.8.30
charset-normalizer	3.4.0	chex	0.1.7
click	8.1.7	cloudpickle	3.1.0
clu	0.0.7	contextlib2	21.6.0
contourpy	1.1.1	curvlinops-for-pytorch	2.0.0
cycler	0.12.1	decorator	5.1.1
dm-tree	0.1.8	docker	7.0.0
docker-pycreds	0.4.0	einconv	0.1.0
einops	0.8.0	etils	1.3.0
filelock	3.16.1	flatbuffers	24.3.25
flax	0.6.10	fonttools	4.54.1
fsspec	2024.10.0	gast	0.4.0
gitdb	4.0.11	gitpython	3.1.43
google-auth	2.36.0	google-auth-oauthlib	1.0.0
google-pasta	0.2.0	googleapis-common-protos	1.65.0
gputil	1.4.0	grpcio	1.67.1
h5py	3.8.0	idna	3.10
imageio	2.35.1	importlib-metadata	8.5.0
importlib-resources	6.4.5	jax	0.4.10
jaxlib	0.4.10	jinja2	3.1.4
joblib	1.4.2	jraph	0.0.6.dev0
keras	2.12.0	kiwisolver	1.4.7
lazy-loader	0.4	ld_impl_linux-64	2.43
libclang	18.1.1	libffi	3.4.2
libgcc	14.2.0	libgcc-ng	14.2.0
libgomp	14.2.0	libnsl	2.0.1

Table D.1: Installed Python packages (Part 1).

Name	**Version**	**Name**	**Version**
libsqlite	3.47.0	libuuid	2.38.1
libxcrypt	4.4.36	libzlib	1.3.1
markdown	3.7	markdown-it-py	3.0.0
markupsafe	2.1.5	matplotlib	3.7.5
mdurl	0.1.2	ml-collections	0.1.1
ml-dtypes	0.2.0	mpmath	1.3.0
msgpack	1.1.0	ncurses	6.5
nest-asyncio	1.6.0	networkx	3.1
numpy	1.23.5	nvidia-cublas-cu12	12.1.3.1
nvidia-cuda-cupti-cu12	12.1.105	nvidia-cuda-nvrtc-cu12	12.1.105
nvidia-cuda-runtime-cu12	12.1.105	nvidia-cudnn-cu12	8.9.2.26
nvidia-cufft-cu12	11.0.2.54	nvidia-curand-cu12	10.3.2.106
nvidia-cusolver-cu12	11.4.5.107	nvidia-cusparse-cu12	12.1.0.106
nvidia-nccl-cu12	2.18.1	nvidia-nvjitlink-cu12	12.6.77
nvidia-nvtx-cu12	12.1.105	oauthlib	3.2.2
openssl	3.3.2	opt-einsum	3.4.0
optax	0.1.5	orbax-checkpoint	0.2.3
packaging	24.2	pandas	2.0.3
pillow	10.4.0	pip	24.3.1
platformdirs	4.3.6	promise	2.3
protobuf	4.25.3	psutil	5.9.5
pytz	2024.2	requests	2.32.3
scikit-learn	1.2.2	scipy	1.9.1
tensorflow	2.12.0	torch	2.1.0
torchvision	0.16.0	tqdm	4.67.0
urllib3	2.2.3	wandb	0.18.7
werkzeug	3.0.6	wheel	0.44.0
wrapt	1.14.1	xz	5.2.6
zipp	3.20.2		

Table D.2: Installed Python packages (Part 2).