Eberhard Karls University Tübingen

Department of Computer Science

Methods of Machine Learning

Bachelor Thesis

Accuracy-Based Line Search

Osane Hackel

Supervisor:
Lukas Tatzel

Examiner:
Prof. Philipp Hennig

October 1, 2023

Abstract

The performance of most optimization strategies such as SGD strongly depends on the chosen learning rate. While there exist approaches for scheduling the learning rate during training, they still require grid searches to find optimal settings. One way to possibly mitigate this issue has previously been suggested by Vaswani, Mishkin, Laradji, *et al.* [1]. The authors propose to perform a line search on the loss landscape at each optimization step to choose the learning rate. However, since usually the overarching goal is to maximize accuracy, this begs the question whether that line search could also be performed on the accuracy landscape. In this work, we explore this avenue and find that performing the line search on the accuracy directly is not feasible. Instead, we introduce smooth accuracy-based stochastic line search (sa-SLS) which uses a smoothed, approximate, version of the accuracy. We test our newly-introduced optimizer on MNIST, CIFAR-10, and CIFAR-100, and find that it achieves comparable, and in some cases even better, results compared to SGD, Adam, and loss-based stochastic line search. Although, on average, sa-SLS takes 1.75x as long as SGD, we believe cumulative training time can still be reduced greatly as no additional hyperparameter tuning is required.

Contents

1 Introduction **3**

2 Background **5**
2.1 Neural Networks 5
2.2 Deep Learning 6
2.3 Training 7

3 Related Work **10**
3.1 SGD 10
3.2 Other Optimizers 11
3.3 Learning Rate Scheduling 12
3.4 Armijo SGD 14
3.5 Loss and Accuracy 17

4 Pre-Study: Loss and Accuracy **20**

5 Method **25**
5.1 Accuracy-Based Line Search 25

6 Experiments **29**
6.1 Preliminar Experiments 29
6.2 Ablation 1: Slope Factor Search 31
6.3 Ablation 2: Search Batch 32
6.4 Main Experiments 34

7 Discussion **36**

8 Appendix **43**
8.1 Data Sets 43
8.2 Figures and Tables 44

1 Introduction

In machine learning, it is all about performance. Thus, if the model, for instance, gets a set of pictures we want it to return the right class for as many pictures as possible. The accuracy measures how many of the data points are classified correctly and usually, this is the metric we aim to maximize. In practice, this can be achieved by changing the weights in the model during training until the optimal setting is found. In the case of a classification task, a parameter setting can be considered good if the output of the network for the real class is higher than for all the other classes.

Finding the optimal weight setting is done by minimizing a loss function. The loss function defines a metric for the distance between the model output and the real label. For real-world problems, however, this minimization problem is not analytically solvable and numerical approaches are required.

Optimizers find the minimum by applying iterative changes to the weight setting. One of the most prominent optimizers is SGD [2] which updates the parameters by taking a step in the direction of the negative gradient of the loss function. The magnitude of this step is called learning rate (lr). However, choosing a good lr is crucial for performance. There exist some approaches that adjust the lr during training [3]–[5]. One newer approach is SGD Armijo [1] which uses an Armijo-conditioned line search to find a reasonable lr in each optimization step. This Armijo condition ensures a sufficient reduction of the loss in each optimization step. This is done by testing the loss at various lrs. As long as using the proposed lr does not achieve a lower loss value as a scaled approximation, the learning rate gets reduced. This chooses in each optimization step a lr that is optimal with regard to the current loss landscape.

But since the main goal in machine learning is to maximize accuracy, we had the idea of an optimizer that performs this line search to find the optimal lr on the accuracy instead of the usually used loss. To test this idea, we design an Armijo conditioned accuracy-based line search (a-SLS). In practice, this idea does not work due to the step-wise behavior of the accuracy landscape which leads to diminishing lrs. As a result, we use a smoothed accuracy instead. This smooth accuracy is computed by using the difference between the prediction of the model for the correct label and the highest prediction for the wrong labels. These data points are scaled by a factor and smoothed by a sigmoid function. Using this new metric, we develop a well-working optimizer: smooth accuracy-based SLS (sa-SLS). This optimizer conducts a line search to find a lr that increases the smooth accuracy more than a scaled approximation of the smooth accuracy of the preceding parameter setting. We benchmark our optimizer on MNIST, CIFAR-10, and CIFAR-100, and find results comparable to SGD and Adam. The lr converges to zero for some seeds and as a result, the training stagnates at a lower accuracy.

Furthermore, we qualitatively explore the loss and accuracy landscape. Here, we find a systematic behavior of the accuracy in the direction of the negative gradient. Since the loss shows a similar behavior, this validates our idea of performing the line search on the accuracy. We additionally find

evidence for a systematic bias with regard to the current mini batch. Specifically, if the gradient is computed on the same batch, the accuracy is overestimated and the loss is underestimated. Building on these results, we conduct an ablation study to investigate strategies for avoiding this bias in our line search. To do this, we conduct the line search on different batches than the one we compute the gradient on. However, in practice, none of our tested strategies work. We assume this could be due to interactions between the gradient norms of the different batches.

Additionally, we conduct a second ablation study to analyze the robustness of the scaling factor for the smooth accuracy. We find surprisingly robust results for most of the different scaling factors. Yet, smaller ones tend to lead to better results on average. This is surprising since larger scaling factors lead to a more closer approximation of the real accuracy function.

Due to the line search, lr tuning can be reduced, but on the other hand, each update step needs more computation. This leads to an average increase in computation time of 1.75x compared to Adam. Since we find our hyperparameters to be relatively robust, we hope they do not require tuning on novel tasks. If this is the case, using our optimizer could drastically reduce the overall needed computation time by avoiding grid searches. Even though the results look very promising, future research is needed to avoid diminishing learning rates and make the optimizer more reliable.

The code for replicating the results can be found in our GitHub repository (`https://github.com/OsaneHackel/accuracy-SLS`)

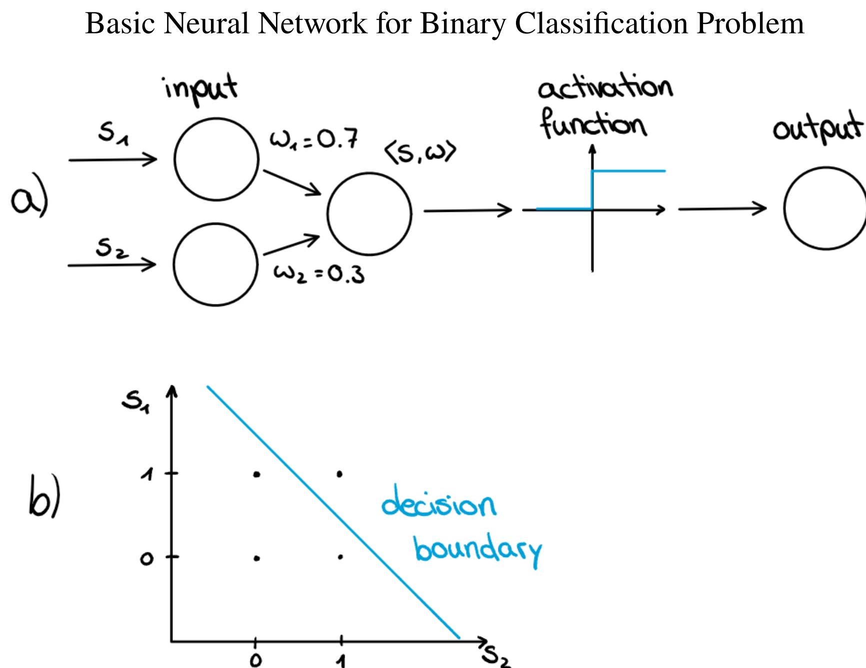

Figure 1: a) Two-layer perceptron, that decides if both stimuli are 1. Takes a two-dimensional stimulus s as input and computes the scalar product of s and weight vector w. These weights can be adjusted during training and an activation function decides based on these values what the output of the model should be. In this case, accepting if both of the stimuli are one and rejecting otherwise. In this case, a decision boundary of 0.8 would work, because this value is passed only if both weight values 0.3 and 0.7 are multiplied with the stimulus value 1 in the scalar product.
b) Feature space of s and visualization of decision boundary. Different combinations of values for the entries in s are shown as black dots. The linear decision boundary for this feature space is shown by the blue line and separates the feature space in an accepting and rejecting area. The accepting area corresponds to both stimulus dimensions being equal to one.

2 Background

Deep learning originated in simulating biological neural networks but has evolved significantly since then. Nowadays, the algorithms get optimized for specific tasks in which only performance matters. This leads to high dimensional parameter spaces, which require more and more complex optimization strategies for finding the optimal parameter setting and achieving good performance.

2.1 Neural Networks

The following will describe some central aspects of deep learning but for more details, we refer to Goodfellow, Bengio, and Courville [6] and LeCun, Bengio, and Hinton [7]. Neural networks can, for instance, be used for classification, where the network would get a picture and should return the object class of the essential part of the picture. A simple neural network consists of an input layer, weights, an activation function, and an output layer. The goal is to give the network some data as input and to receive a reasonable output depending on the task. The binary example

visible in Figure 1 is an extreme simplification of what neural networks nowadays are because the input normally is high-dimensional (e.g., image pixels). Therefore, the weight vector also needs to be high-dimensional. The operation that happens in one layer is an affine transformation

$$f(x) = Ax + b \tag{1}$$

with $A \in \mathbb{R}^{m \times n}, x \in \mathbb{R}^n, b \in \mathbb{R}^m$. A and b are called the weights, or parameters, of the layer. For $b = 0$ this is a linear transformation.

Some problems are not solvable with the linear decision boundary created by an affine transformation, so a nonlinear operation, called the activation function, is added. The most prominent activation function used nowadays is the rectified linear unit $a(x) = \max\{x, 0\}$, also called ReLU [8]. The ReLU function leaves positive values unaffected, but negative values are clamped to zero.

These transformations, conducted in the layers, are concatenated such that the following transformation uses the result of the preceding one as input. In the case of classification tasks, the model needs to return a distribution over the different classes in the last layer of the model. This allows an interpretation of the output and can be done by using $A \in \mathbb{R}^{m \times c}$ with c being the number of classes. In order to change the output of the model into a distribution, commonly a softmax operation is applied.

2.2 Deep Learning

Deep learning is a subsection of machine learning in which the representations for extracting the relevant features are learned by the model. These extracted features are then used by a classifier to determine the correct class. The usage of deep learning led to very impressive results for example in image recognition [9]–[11] but also in speech recognition [12]–[14]. In contrast, classical machine learning requires explicit feature-extracting methods to bring the raw data into a state where a learning subsystem could perform a classification task on. This kind of preprocessing to extract the relevant information is central to the performance of the classifier. If the representation of the data does not contain enough information to distinguish two classes, not even the best classifier will be able to separate them. Emerging from this problem, representation learning started to play a huge role in machine learning. In representation learning, the models find representations of the raw data by themselves. It was not until 2006 though [15] that it became tractable to train these deep neural networks.

A deep neural network has multiple hidden layers, which extract more and more abstract representations of the input data [16]. This allows the model to recognize discriminative features which enables a decision boundary to classify the object. The remarkable advantage of deep learning is rooted in the combination of representation and classification in one network. Hereby, the representation can be adjusted depending on the task and the classifier can specialize in discriminating certain features. No handcrafted feature extraction method is needed as the models learn this by

themselves. This saves time and money but increases the amount of needed training data which can be expensive to generate.

The development of a machine learning solution for a problem involves a lot more steps than only training the neural network. First of all a good data set needs to be found and preprocessed, which enables the network to learn the required representations. Also, the choice of an appropriate network is important, because if the network is not expressive enough, it won't be able to learn and if it has too many layers, it tends to be difficult to optimize [17]. Furthermore, a loss function as a metric for the network performance and an optimizing strategy need to be chosen and tuned.

CNNs For image recognition tasks, commonly convolutional neural networks (CNNs) are used. CNNs use convolution and pooling layers to extract the relevant features of an image. Convolution layers are oriented on the complex cells of the visual cortex and find for instance edges by including multiple feature maps and looking for differences in them. This convolves many inputs into one output. Pooling layers on the other hand are used to merge features with similar properties into one. This helps the network to become shift invariant for small distortions. This process allows the network to receive pixels and change their representations until an expressive feature map evolves.

2.3 Training

Commonly, neural networks are trained using supervised learning. That means, during training, the network is presented with both an input datum X, as well, as a target y. Thus a single training sample consists of the tuple (X, y). The target y, also called label in case of classification, has to be manually annotated by a human. This is in contrast to unsupervised learning, where a network is solely trained from data alone without any additional targets. The data set is commonly divided into two parts, the training set, which the network uses to learn the optimal weight vector, and the test set, which only is used for evaluation and can give insight into the generalizability of the network.

Based on the information contained in the training set the model learns based on optimization strategy values for its weights. In image classification, the goal is to find weights that achieve a high accuracy for classifying objects in images. This means the correct class should have the highest value in the output of the model. When the network gets initialized the weights are normally chosen randomly, so the expected accuracy rate is on a random level.

Loss For training the model, a metric is needed that gives feedback about the quality of the output. In image classification, commonly, the cross-entropy loss [18] (CE-loss) is used, which is defined as the Kullback-Leiber divergence between two categorical distributions. The correct label is represented as a δ function and then this distribution is compared to the distribution of the

output from the model. For a single sample, this results in

$$\ell(x, y; w) = -\log\,[f_w(x)]_y \tag{2}$$

where $[f_w(x)]_i$ denotes the output of the model f with weights w for the true class i. Notice that w is usually extremely high dimensional and consists of millions of parameters. Therefore, the loss landscape is high dimensional as well. Because the goal is to get an output that is close to the actual label, the loss (e.g. the distance between the predicted and the actual class) should be as small as possible. The goal is to find a parameter setting, for which the expected loss for elements of the given distribution is minimal:

$$w^* = \operatorname*{argmin}_w \mathbb{E}_{x,y\sim P(x,y)}\ell(x, y; w) \tag{3}$$

Here, $P(x, y)$ denotes the distribution of input data x and corresponding labels y. However, in practice, it is often impossible to compute this expected value directly. Thus, it gets approximated using a Monte-Carlo estimate using a fixed number of samples $x_i, y_i \sim P(x_i, y_i)$:

$$w^* \approx \operatorname*{argmin}_w \frac{1}{N}\sum_{i=1}^{N}\ell(x_i, y_i; w) \tag{4}$$

This procedure is known as empirical risk minimization. Equation 4, assumes that the training samples are drawn from the real world and follow its distribution. Loss in the following will be used as the empirical mean computed of the loss values for a subset of the data:

$$\mathcal{L}(X, Y; w) = \frac{1}{N}\sum_{i=1}^{N}\ell(x_i, y_i; w) \tag{5}$$

Softmax The prediction of the model is modified by the softmax operation before the loss is computed.

$$\begin{aligned} \sigma &: \mathbb{R}^K \mapsto (0, 1)^K \\ \sigma(x)_i &= \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \end{aligned} \tag{6}$$

The softmax function σ transforms a k-dimensional vector x with arbitrary values to a vector with values between 0 and 1. Furthermore, it ensures that all values sum to 1. Consequentially, it is a valid discrete distribution. This modified vector can then be used as a distribution for the cross-entropy loss.

Optimizer Finding a minimal point in this loss landscape is the task of the optimizer. One of the most popular approaches is gradient descent, which uses the gradient of the loss function that is computed via backpropagation through the network at the current parameter setting. Afterward, the weights are updated by moving them in the direction of the negative gradient. The gradient

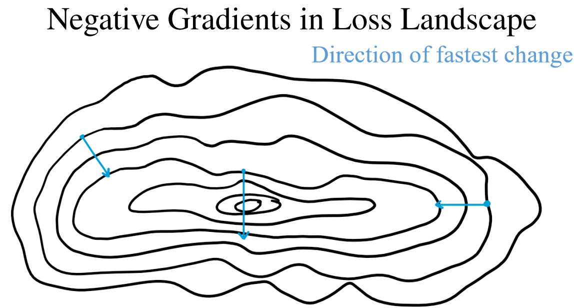

Figure 2: Schematic loss landscape shown by contour lines. The faster the change the closer the contour lines are together. Arrows resemble negative gradients that point in the direction of the fastest decrease. These always are orthogonal to a tangent of the contour line at this point.

always points in the direction of the steepest rise, so the negative gradient points in the direction of the steepest slope as can be seen in Figure 2. Since we look for the maximal reduction of the loss landscape it is straightforward to update the weights in this direction. After knowing the direction in which the weights should be changed, there is still the magnitude of the change to determine. This lr is a hyperparameter and has a large influence on performance.

Because the data sets used in machine learning are extremely large, it would be intractable to always compute the loss for the whole data set. One solution for this is stochastic gradient descent (SGD). Therefore, the data set is partitioned into so-called mini batches which contain random samples from the training set. One epoch includes as many mini batches, sampled without replacement, as needed to see every training item once. Now, in the case of SGD, the update step changes the weights based on the loss computed on one mini batch.

Regularization A common issue of neural networks is overfitting. This means that the model performs well on the training data but generalizes poorly. This happens if the model learns the classifications in training data by heart instead of learning abstract relationships. One way to deal with overfitting, and thus increase overall performance, is to introduce regularization [19]. In the case of L_1 regularization, the optimizer will prefer solutions with sparse weights. In the case of L_2 regularization, solutions are preferred that keep the values of the weights small to avoid the explosion of single weights. Specifically, this is done by modifying the loss.

$$\begin{aligned} L_1 \text{ Regularization:} \mathcal{L}(X, Y; w) &= \mathcal{L}(X, Y; w) + \lambda ||w|| \\ L_2 \text{ Regularization:} \mathcal{L}(X, Y; w) &= \mathcal{L}(X, Y; w) + \lambda ||w||^2 \end{aligned} \tag{7}$$

L_1 regularization corresponds to choosing a zero-centred Laplace distribution as prior for the model weights. This prefers solutions, which for example use fewer features for the classification and lead to sparser parameter settings. More commonly used is L_2 regularization, which, on the other hand, corresponds to choosing a zero-centered Gaussian prior on the model weights. This results in more zero-centered balanced weights, because the quadratic influence of the parameter norm punishes high values in the parameter space. This prevents overfitting and therefore leads to better generalisation.

Difference Between SGD and GD

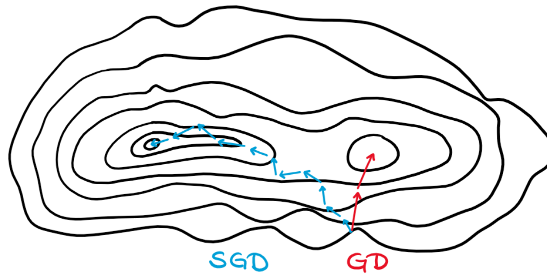

Figure 3: Schematic visualization of the difference between gradient descent (red) and SGD (blue). Because of stochasticity in the mini batches more of the parameter space is explored and SGD can avoid getting stuck in a poor quality minima.

3 Related Work

3.1 SGD

SGD is an optimization strategy for finding a minimum in the loss landscape of different parameter settings. As described earlier, SGD updates the weights of a network based on the loss landscape of a mini batch. Therefore, the gradient of the loss is computed and the weights are updated by a certain lr in the direction of the negative gradient:

$$w_{i+1} = w_i - lr \cdot \nabla_w \mathcal{L}(X, Y; w) \tag{8}$$

where lr denotes the adjustable learning rate and N denotes the size of the mini batch. The gradient of the loss function for the current weight setting is computed by backpropagation. Hereby, the chain rule is used to calculate the derivative of the performed transformation with respect to the input. This strategy is repeated until the first layer is reached and therefore the derivative of the whole network is computed. To save computation time the single activations already are tracked during the forward pass.

In the case that N equals the size of the data set, we get full gradient descent instead of SGD. For small N the approximation of the gradient becomes noisy because less of the training items are used for the computation. Noisy gradients can lead to a weak form of regularization since the different approximations of the loss function prevent the optimization from getting stuck in a poor quality local minima [20] as shown in Figure 3. Even though SGD is a well-working optimizer, it still has some disadvantages. One is the relatively slow convergence and another is the dependence of the performance on the lr [1].

Momentum A common way to increase convergence speed is the usage of a momentum term [21] given in Equation 9. This term regularizes the influence, the previous gradient has on the current

update step.

$$\begin{aligned} v_i &= \beta v_{i-1} + (1-\beta)\nabla_w \mathcal{L}(X,Y;w) \\ w_{i+1} &= w_i - lr \cdot v_i \end{aligned} \tag{9}$$

For $\beta = 0$, regular SGD is recovered. As β increases, the influence of the earlier gradients increases as well. This can be interpreted physically as modeling the inertia of a ball rolling down a valley. If several gradients point in the same direction, the changes become larger, e.g. the ball rolls faster, and if the gradients point in different directions the learning is slowed down. Qian [21] showed, that using momentum increases the convergence rate.

3.2 Other Optimizers

In recent years, numerous different optimizers have been proposed to improve upon SGD. Notably among those are, Adam [22] and AdamW [23].

Adam The most prominent of these is Adam, which incorporates the mean and the uncentered variance of the gradient in the weight update. Let g_i be the gradient of the loss function at the update step i. Then the estimate of the mean is computed and bias-corrected in the following way:

$$\begin{aligned} m_i &= \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_i \\ \hat{m_t} &= \frac{m_t}{1-\beta_1^t} \end{aligned} \tag{10}$$

Afterward, the estimate of the uncentered variance of the gradient and its bias correction are computed:

$$\begin{aligned} v_i &= \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_i^2 \\ \hat{v_t} &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{11}$$

Here β_1 and β_2 are values between 0 and 1, which define the influence of the mean and variance of the prior step on the current step. Notice that β_1 corresponds to the momentum term β used in SGD with momentum. The influence increases, as the values increase. Afterward, the parameters are updated separately

$$w_i = w_{i-1} - lr \cdot \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{12}$$

where m_t and v_t are initialized with 0. To achieve a good convergence rate, fine-tuning of the hyperparameters β_1, β_2, and the lr is needed. Algorithms, that use this strategy of changing the gradient during the training are called adaptive gradient algorithms.

AdamW A modification of Adam's adaptive gradient algorithm is AdamW, which uses decoupled weight decay instead of L_2 regularization to improve the generalization of Adam. Weight decay [24] exponentially decays the weights of a neural network during the training:

$$w_{i+1} = (1 - \lambda)w_i - lr\nabla_w \mathcal{L}(X, Y; w) \tag{13}$$

The factor of the weight decay in combination with the lr gives the L_2 regularization factor for SGD. This is not the case for Adam. Consequentially, the authors propose to decouple the weight decay from the lr with a changed update rule:

$$w_i = w_{i-1} - lr \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda w_{i-1} \tag{14}$$

Adam performs better with decoupled weight decay than with standard L_2 regularization [23]. This is a further example for the thesis, that the lr has a noteworthy influence on the performance. Both Adam and AdamW indirectly change the lr, by changing the influence, the parameters have on the update step depending on the variance of the overall gradients.

3.3 Learning Rate Scheduling

Another ongoing development focuses on directly adjusting the lr during the training [25]. Here, the main idea is to start training with a large lr and gradually decay it to a lower lr during the course of the training. This is motivated by the fact that as training proceeds smaller step sizes are necessary to make meaningful progress. Yet, in the beginning, such small step sizes would only make little progress.

Adaptive Step Size A well-working strategy for lr scheduling on SGD adapts the lr whenever a plateau is reached [26], [27]. Here the main difficulty is to assert when a plateau in the loss landscape is reached. This can be done by observing the change in the loss over a certain amount of update steps. Reducing the learning rate whenever a plateau is reached significantly reduces the loss.

Cosine Annealing For Adam and Adam-related optimization strategies, cosine annealing [28] is used commonly, e.g. in Liu, Lin, Cao, *et al.* [29]. In cosine annealing, during the training, the lr is gradually reduced to lr_{min} according to

$$lr_i = lr_{min} + \frac{1}{2}(lr_{max} - lr_{min})\left(1 + cos\left(\frac{\text{Epoch}_{curr}}{\text{Epoch}_{max}}\pi\right)\right) \tag{15}$$

There also exist different optimizers, which adjust the lr during the training [3]–[5], [30].

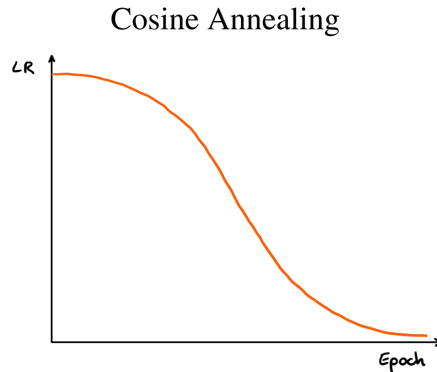

Figure 4: Decrease of the lr when using cosine annealing.

AdaGrad Duchi, Hazan, and Singer [4] propose AdaGrad, which uses first-order information for a dynamic lr adaption over the training:

$$\Delta w_t = -\frac{lr}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t \tag{16}$$

The global lr is divided separately for each weight by the sum of the L_2 norm of all the previous gradients of this weight, and the result of this is used as the size of the update step in the direction of the current gradient for this specific weight. Therefore, step sizes get smaller, if the norm of the gradient becomes larger. In the beginning, fewer gradients influence this sum, so the lr is potentially high and decays over the training similar to cosine annealing. The main idea is to equalize the changes between the different weights, by giving each an individual lr.

AdaDelta Some problems, that arise with this method, are tackled by AdaDelta [3]. The first problem is that AdaGrad is sensitive to the initial parameters and especially the gradients. If one dimension starts with a high gradient norm, this will stay in the sum during the whole training and keep the lr small. This is solved by using an exponential moving average to compute the sum of the gradient norms:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2 \tag{17}$$

In this way, present gradients have a larger influence on the current lr, than the preceding ones. Another problem in AdaGrad is the selection of the global lr because this has a large influence on performance. To solve this, the authors suggest a dynamic lr that is computed for each dimension in each step separately.

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{18}$$

where the ϵ is added for more numerical stable results and to start learning in the initial condition. The change of weight is, computed without setting a general lr:

$$\Delta w_t = -\frac{RMS[\Delta w]_{t-1}}{RMS[g]_t} g_t \tag{19}$$

These changes result in an optimizer, which adjusts the lr during the training without introducing a further hyperparameter.

SALeRA SALeRA [5] is an optimization approach that claims to de- and increase the lr in a way such that learning happens as fast as possible but not too fast. For this, they introduce two statistical tests. The first one compares the norm of the normalized gradients to the sum of randomly chosen unit vectors. Both of the sums are computed as exponential moving averages. This helps to find out, if they are correlated, which is the case if the norm of the cumulative path of the gradients is larger than the sum of the random unit vectors. If they are correlated, there exists a global learning direction and the lr can be increased. In the case of anti-correlation, the lr decreases. The second one tests the mini batch loss on catastrophic changes, meaning large increases in the loss or gradient norm, and halves the lr in these cases. This optimizer claims to increase the lr whenever possible to speed up training. The second strategy reduces the lr whenever needed.

3.4 Armijo SGD

Another approach for finding an optimal lr via line search, called SGD with Armijo line search (SGD Armijo), has been suggested by Vaswani, Mishkin, Laradji, *et al.* [1]. This optimizer performs a line search to find a lr that fulfills the Armijo condition. The found lr is then used for updating the parameters.

Armijo Condition The strategy of the optimizer is built on the Armijo condition introduced by Armijo [31] for functions that have a Lipschitz continuous first partial derivative. The condition states, that a concatenation of steps in the direction of the gradient, converges to a minimum if the step size is chosen correctly. Each step therefore needs to reduce the function more than a negative multiple of the steepness of the function.

$$S^*(x, \delta) = \{x_\lambda : x_\lambda = x - \lambda \nabla f(x), \lambda > 0, f(x_\lambda) - f(x) \leq -\delta |\nabla f(x)|^2\} \tag{20}$$

Here, λ needs to be chosen in a way, that the change of x following the update rule reduces the value of the function more than the scaled steepness of the function. Therefore, δ needs to be larger than zero and smaller than $\frac{1}{4}$ of the Lipschitz constant. The Lipschitz constant is an upper bound for the steepness of the function. This Armijo condition is known as the first wolf condition. The second wolf condition also called Goldstein condition, takes the curvature into account for choosing a good lr [32]. A lr is declared to satisfy the Wolfe conditions, if the following inequalities are

Algorithm 1: SGD Armijo
T stands for the number of batches per epoch.

Data: f, w_0, lr_{max}, b, c, β, γ, opt
Result: updated weights

```
1  for k=0,..,T do
2  |  i_k ← sample mini batch of size b
3  |  lr ← reset(lr, lr_max, γ, b, k, opt) / β
4  |  armijo ← False
5  |  while armijo = False do
6  |  |  lr ← β · lr
7  |  |  w'_k ← w_k − lr · ∇f_ik(w_k)
8  |  |  armijo ← f_ik(w'_k) ≤ f_ik(w_k) − c · lr||∇f_ik(w_k)||^2
9  |  end
10 |  w_{k+1} ← w'_k
11 end
12 return w_{k+1}
```

fulfilled:

$$\text{Armijo: } f(w_i + lr_i \cdot p_i) \leq f(w_i) + c_1 lr_i p_i^T \nabla f(w_i) \tag{21}$$

$$\text{Goldstein: } -w_i^T \nabla f(w_i + lr_i \cdot p_i) \leq -c_2 p_i^T \nabla f(w_i) \tag{22}$$

where p_k is the direction in which the optimization step is performed. The Armijo condition ensures a sufficient reduction of f and the Goldstein condition ensures a sufficient reduction of the curvature when optimized with the chosen lr.

Line Search SGD Armijo now uses this strategy in the stochastic setting in which mini batches are used. This is done by performing a line search on the loss landscape for each optimization step until an Armijo condition is fulfilled. A detailed description of the optimization procedure can be found in Algorithm 1.

The current mini batch is drawn without replacement so that in every epoch each training item is seen once. In each step, the lr is reset using one of the different strategies shown in Algorithm 2. Then the line search starts with a while loop, which only stops if the Armijo condition is satisfied. Until this happens, the lr is reduced and the weights are updated with the new lr, following the SGD update rule. These updates are only temporary, except for the one that satisfies the Armijo condition. In the following, the single components of this optimizer are described in more detail.

Before each optimization step, the lr is scaled by a factor $\alpha = \sqrt[N]{2}$ where N denotes the number of update steps in an epoch. This is necessary because the line search is only able to decrease the lr and without increasing, the lr would converge to 0. The reduction of the lr in the line search happens by multiplying it with a factor β, which is smaller than 1. The update of the weights is performed in the direction of the negative gradient with the temporal chosen lr as step size. This temporal update is kept only if the Armijo condition is satisfied.

Algorithm 2: Reset of Learning Rate

Data: lr, lr_{max}, γ, b, k, opt
Result: lr

```
1  if k=1 then
2  |  return lr_max
3  end
4  else if opt = 0 then
5  |  lr ← lr
6  end
7  else if opt = 1 then
8  |  lr ← lr_max
9  end
10 else if option = 2 then
11 |  lr ← lr · γ^(b/n)
12 end
13 return lr
```

They use a modified version (Equation 23) of the Armijo condition:

$$f_k(w_k - lr_k \nabla f_k(w_k)) \leq f_k(w_k) - c \cdot lr_k ||\nabla f_k(w_k)||^2 \tag{23}$$

The left side of the Equation equals $f_k(w_{k+1})$ if the lr is accepted because in the parenthesis the update step is formulated. The loss of this new parameter setting should be smaller than the approximation situated on the right side of the equation. This approximation can be imagined as a scaled linear approximation on the loss of the present parameter setting. The scaling happens via the parameter $c < 0$, which reduces the steepness of the slope.

During one optimization step, the approximation stays the same and the different lrs are sample points on a scaled linear approximation. Therefore, the curve of the real loss for the different lrs is compared against an approximation. As long as the approximation is better than the real loss the lr decreases.

According to the authors, SGD Armijo reaches faster convergence and better generalization performance on classification problems when used with deep neural networks. They report SGD Armijo beating competing optimizers and state that the increased computational cost of their optimizer is worth the benefits of it. According to their statement using SGD Armijo avoids cost-intensive hyperparameter searches for lr scheduling. Furthermore, they prove that the stochastic version of the classical Armijo line search reaches deterministic convergence rates for convex functions, given that the following conditions are met. The loss function f must be differentiable, have a finite sum structure, be lower bounded, and be L-smooth. Additionally, the model needs to be able to interpolate the data, meaning that the gradient of the loss with regards to each sample converges to zero at the optimum.

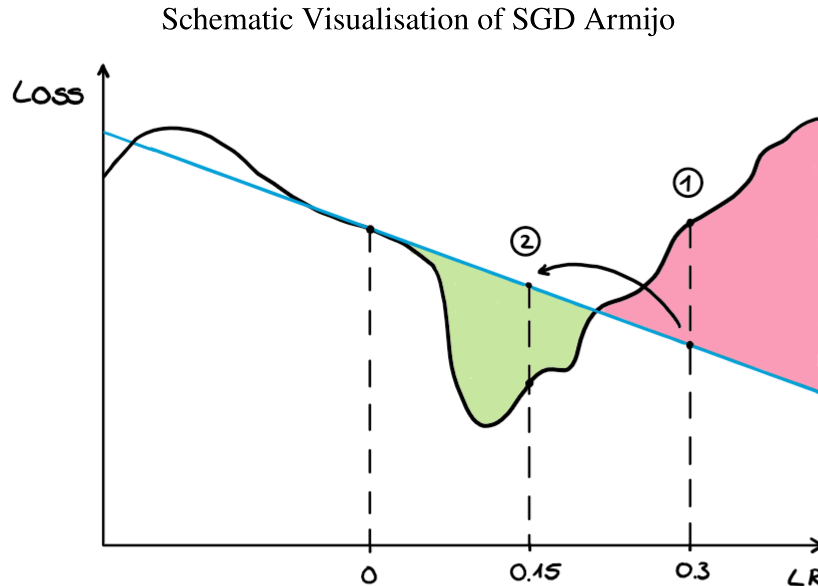

Figure 5: Lr reduction following SGD Armijo. Real loss is shown on the y-axis for the different lrs on the x-axis. Lr 0 gives the loss for the current parameter setting. The scaled approximation (red) is used to evaluate the quality of the update step. 1. A lr of 0.3 is chosen and compared to the corresponding point on the linear approximation. Because the corresponding value is smaller than the real loss, the Armijo condition is not satisfied and the lr gets reduced. 2. The reduced lr is also not able to give a smaller loss than the linear approximation, so the lr is reduced again. All lrs in the red area will be rejected and reduced until the lr falls in the green area. 3. Real loss is lower than scaled approximation, so the update is performed and a new mini batch is drawn.

3.5 Loss and Accuracy

As Schneider, Balles, and Hennig [33] point out, it is not trivial which one of train loss, test loss or test accuracy is the metric that defines the quality of a network best. The user is interested in good test accuracy but training on the accuracy is not possible due to the discontinuity of the accuracy landscape. Thus, instead, the cross entropy loss is usually reduced to increase the accuracy. We are interested if it is beneficial to incorporate the accuracy of a parameter setting in the optimization procedure. Explicitly, by using the accuracy for choosing the lr in a similar way to SGD Armijo. Therefore the following section aims to give an overview of the differences between loss and accuracy.

Loss The commonly used cross-entropy loss (CE-loss) (Equation 2) is a way to describe the distance between the true label and the prediction of the model. The CE-loss $(-log(p))$ is continuous and its derivative $(\frac{1}{p})$ is positive for p between zero and one. Hereby, p stands for the output of the model for the correct class after softmax is applied. Therefore, $p = 0$ corresponds to misclassification and $p = 1$ to correct classification. Due to the logarithmic shape of the CE-loss, this leads to larger values as the output of the model gets worse and to values converging to 0 if the classification becomes better.

The gradient is central for updating the parameters of the model during learning. In the CE-loss,

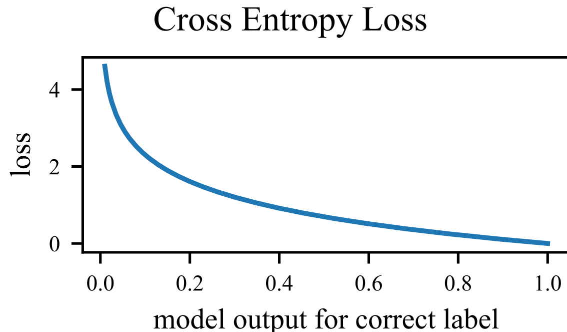

Figure 6: Cross entropy loss on one sample. The x-axis gives the softmax value for the right class. The higher the value, the better the prediction of the model. The y-axis depicts the cross entropy loss for the given prediction of the model.

the gradient converges, for $p = 0$, to infinity and for $p = 1$ to 1. This enables the model to learn faster in case of high loss values (i.e. p close to 0) and still allows learning even if the loss value is already small (i.e. p close to 1).

Accuracy In contrast to the loss, the accuracy is not continuous, because the accuracy only gives the proportion of how many items have been classified correctly. For instance, if the batch size is four, the only possible accuracy values are $0, 0.25, 0.5, 0.75, 1$. Consequentially, small changes in the weights sometimes do not change the accuracy. Because for the accuracy to change, the number of correctly classified items has to change. But for this number to change, the output of the model for at least one item needs to change so much, that another value in the output vector is the largest, and the classification changes. Especially, when the parameter setting is already good, plateaus can emerge that have the same value for close parameter settings.

In a binary classification, the accuracy is zero for $p < 0.5$ and one for $p > 0.5$. For $p = 0.5$ the accuracy is undefined, which makes the accuracy non-continuous. Therefore, the derivative of the accuracy is zero for the constant parts of the function or undefined for the jump points. The absence of a meaningful gradient makes it impossible to use the accuracy as optimization metric.

Interpolations The 0-1 loss is a metric that shares most of the properties with the accuracy function. The only difference is, that the 0-1 loss is one for $p < 0.5$ and zero for $p > 0.5$, so the accuracy is one minus the 0-1 loss. Thus, maximizing the accuracy or minimizing the 0-1 loss leads to identical solutions.

One disadvantage of the CE-loss is that samples that already are classified correctly still have an influence on the training because the gradient of the CE-loss function never becomes zero. To solve this problem, focal loss [34] interpolates between the CE-loss and the constant zero function.

$$\mathcal{F}(X, Y; w) = (1 - p)^{\gamma} \cdot (-log(p)) \tag{24}$$

This zero function is the 0-1 loss for $p > 0.5$ in which the accuracy value would be one. The

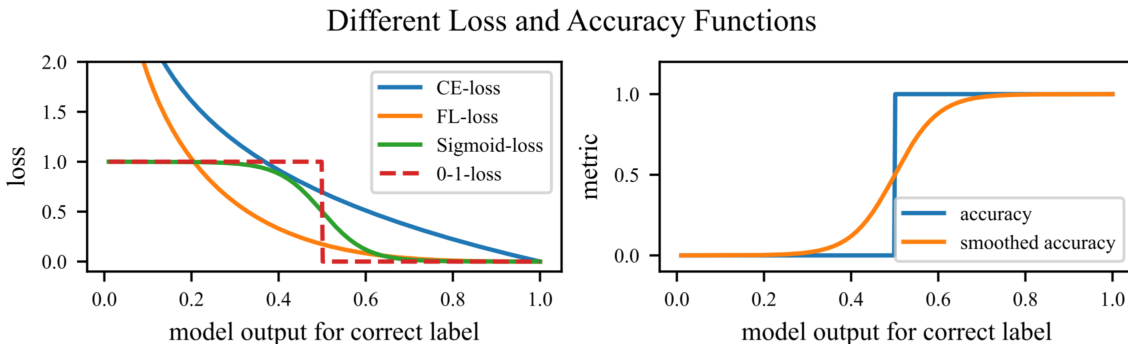

Figure 7: Left: Comparison of CE-loss (blue), focal loss (orange), sigmoidal loss (green) and 0-1 loss (red dashed).
Right: Accuracy (blue) makes the step change in the moment, the output of the model gives the correct class the highest value. Smooth accuracy (orange) emerges from the accuracy smoothed by a sigmoid function and is continuous.

gradient of the focal loss converges for $p \rightarrow 1$ to 0. Therefore, this interpolation gives the misclassified samples more weight than the correctly classified ones. In Equation 24, γ determines how fast the loss converges to zero, with larger values leading to faster convergence. This modified loss works well on very imbalanced data sets that consist mostly of easy samples and only a few difficult ones. The authors also test some slightly different loss functions and observe surprisingly robust results. We can take from that, that the exact shape of the loss function can be changed without destroying the performance. Figure 7 presents the described versions of the loss.

Smooth Acccuracy Similar to the focal loss, one could try to interpolate between the constant parts of the accuracy function. Therefore, a sigmoid function can be used to smooth the steps and make the accuracy continuous. In this way, the accuracy is replaced by a more tolerant smooth accuracy function. Therefore, if the network is close to the right classification, the accuracy value is not as low, as if it is further apart. Therefore, the smooth accuracy function also gives information about the certainty of the model, similar to the loss. These different metrics are compared in Figure 7.

Loss Landscape Because CE-loss and accuracy are correlated, most optimization strategies avoid using the accuracy during optimization and minimize the loss instead. This works, because if the distance between the right and the predicted labels gets minimized for all the training examples the number of correct classified objects increases.

Because the update step in SGD happens in the direction of the negative gradient of the loss, it is interesting how the loss and accuracy landscape behave on this slice. Mutschler [35] describes the loss landscape as being u-shaped (Figure 8) in direction of the gradient. Since we could not find similar statements about the accuracy landscape, we conduct a pre-study to analyze the accuracy landscape in the direction of the negative gradient of the loss.

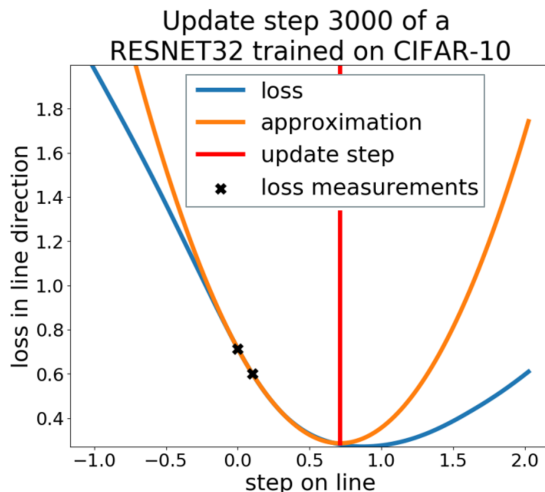

Figure 8: Loss in the direction of the negative gradient. The x-axis gives the size of the update step in the direction of the negative gradient and the y-axis gives the loss value for this fictional parameter setting of the real loss (blue) and the quadratic approximation (orange). It is visible that the real loss shows a quadratic shape close to the current parameter setting. Figure taken from Mutschler [35] (with permission).

4 Pre-Study: Loss and Accuracy

Because there is little known about the relationship between loss and accuracy landscape, we perform a pre-study to compare these. As SGD is the optimization strategy we want to improve, we are only interested in the direction of the gradient. We choose a plotting strategy similar to Mutschler [35]. During the training, we evaluate the loss and accuracy landscape by computing these metrics for different lrs in the direction of the negative gradient. This is done because the update step of SGD will be in this direction and our optimizer aims to find the optimal stepsize for this parameter update.

In case of loss, we try to find a lr, which gives a smaller loss than the one we have at the moment. Vaswani, Mishkin, Laradji, *et al.* [1] show that it is possible to find this lr by line search, which operates only in the direction of the negative gradient. Because we would like to transfer this line search to the accuracy landscape, we are interested in the shape of the accuracy function in the direction of the negative gradient.

We choose the evaluation steps by dividing the time of the training until convergence is reached into 10 parts. At these steps, we analyze the development of loss and accuracy when trained with plain SGD. This strategy is chosen, because we expect most of the qualitative changes to happen in this time period. We sample the landscapes in the direction of the negative gradient with the following lrs:

$$lr = [-1, -0.5, -0.1, -0.01, -0.001, -0.0001, 0.0, 0.0001, 0.001, 0.01, 0.1, 0.5, 1]. \tag{25}$$

We also investigate the stochasticity of the behavior of loss and accuracy by evaluating the metrics

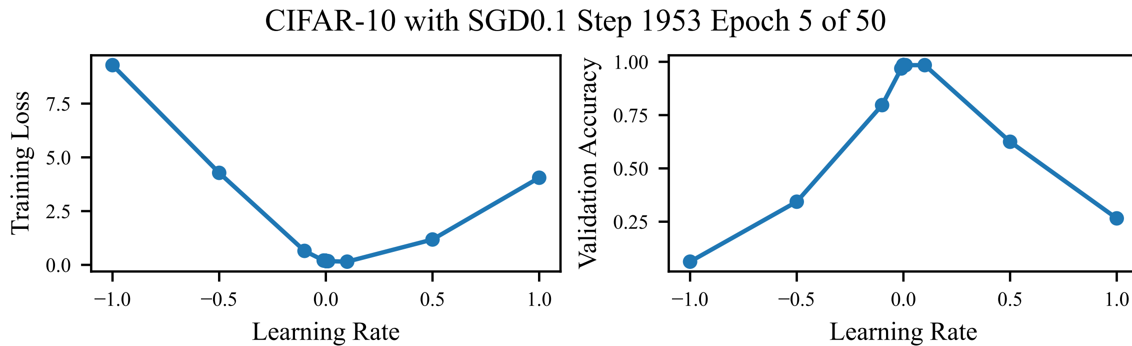

Figure 9: Loss (left) and accuracy (right) landscape in the direction of the negative gradient sampled by different lrs. For the loss landscape, a nearly quadratic shape is visible. Large negative and positive lrs lead to higher values in loss and all curves only show one minimum lying between those extreme values. An inverted quadratic shape in the middle and bounded accuracy values for the edge lrs are visible. This bounding of the values comes from the probability of guessing the right class without any knowledge. If there are for example 10 classes and the network has no idea, which one is the right one, it still has a probability of 10 % to guess the right one.

on different batches than the one we computed the gradient on. As the ground truth of loss and accuracy, the whole validation set is used. In addition to the batch, with batch size 128, which gives the gradient, we evaluate two different batches with the same size and one with batch size 1000. This is inspired by Mutschler [35] because he uses larger batch sizes for finding the optimal lr.

Figure 9, replicates the quadratic shape of the loss, which has already been observed by Mutschler [35].

Since we find qualitatively similar results for all the data sets, we only present results for CIFAR-10 trained with SGD and a constant lr of 0.1 as representative examples. We refer to our GitHub repository for additional plots and the code for replicating the plotted results.

Systematic Behavior There exists a systematical relationship between lr and accuracy value for the accuracy landscape. Figure 9 shows the hat-shaped accuracy landscape in the direction of the negative gradient. Accuracy values are bounded by the probability of guessing the right class which limits the extreme values. In contrast to this, the loss is not bounded and therefore can become large for bad parameter settings. Even though the shape of the accuracy and loss landscape varies between the different experiments and depends on the point in training, the systematic relationship stays the same.

Widening At the beginning of the training, accuracy curves look chaotic but after a while, the described hat shape emerges. The curve first has a narrow peak close to the current parameter setting. As the training proceeds, this peak widens (Figure 10). Reasons for this widening could be more stable optima or smaller gradients at the end of training.

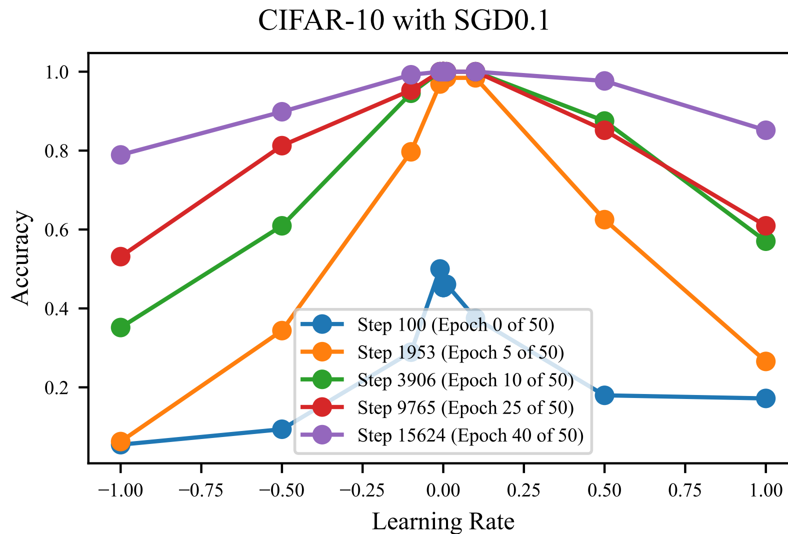

Figure 10: CIFAR-10 on ResNet34. Widening of the same batch curve. Accuracy landscape becomes more tolerant regarding larger lr as the training proceeds. For interpretation of the size of the update step, the gradient norm needs to be taken into account because lr and gradient norm define together the size of the update step. For evaluations of the accuracy landscape on different experiments, we refer to our GitHub repository.

Overfitting We also found a slower widening of the validation accuracy curve. This means, that even though the found minima is wide on the train set, it is still narrow on the test set. Following this, we assume overfitting on the seen data. Additionally, the emerging gap between the performance on the train and the test set speaks for overfitting. Initially, the validation accuracy curve is close to the accuracy curves of the other batches, but as the training proceeds, a gap emerges (see Appendix Figure 19).

Different Batches The accuracy landscape also shows a systematic behavior for the different batches (see Figure 11 and for the loss Appendix Figure 20). As described, the validation batch shows an overall lower accuracy and a higher loss compared to the other batches due to overfitting. The big batch curves are relatively smooth and can be seen as an approximation for the average of the different small batches. This visualizes the stochastic behavior of drawing batches since the accuracy landscape depends on the samples in the batch. The smaller the batches, the larger the expected differences between the computed values. As a consequence, one could use larger batches to find the lr, because the results are prone to give more reliable results.

Batch Bias Furthermore, it can be seen, that bias is a real problem. Specifically, if the gradient is computed on the same batch, the accuracy is overestimated. When updating the weights of the model in the direction of the negative gradient on the current batch, we can see a huge gain in accuracy. Compared to this, we observe smaller improvements for the same lr on different batches. The opposite behavior emerges if the update happens in the direction of the positive gradient. Here, the evaluation on the same batch underestimates the accuracy values of the different batches. This can be explained by the fact that the gradient gives the direction of the steepest change for

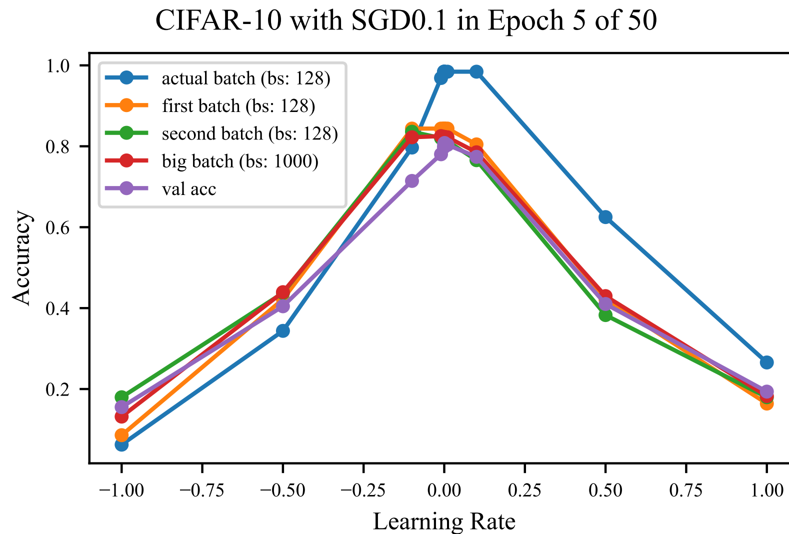

Figure 11: CIFAR-10 on ResNet34. Accuracy landscape in the direction of the negative gradient sampled by different lrs. Same batch (blue), different, but same-sized batch (orange, green), and the validation batch (violet) all show systematic differences. Larger batches (red) lead to smoother behavior. Different small batch curves fluctuate around the big batch curve. The same batch shows bias by having higher accuracy values for positive lr and lower values for negative lr as the other batches. The validation set also shows asymmetry in the accuracy landscape.

this specific data set. As a consequence leads to updating the parameters in this direction to larger changes. The described effect speaks for the usage of different batches for computing the gradient and finding the lr. If this is not done, one would get an lr that is only optimal for this specific batch.

Step Wise Accuracy Taking a closer look at the accuracy values for small lrs shows the step-wise behavior of the accuracy landscape. For example, lrs between -0.001 and 0.1 can lead to the same accuracy value (Appendix Figure 3). This occurs because changing the weights that little, does not lead to a different classification for any of the images. Thus, even though the values in the model output change, the classifications and therefore the accuracy stay the same.

Different Learning Rates We also compare the landscapes that emerge if different lrs are used in the SGD update. It is visible that choosing a smaller lr leads to landscapes on the validation set that are more sensible to smaller lrs. Consequentially, the window in which the accuracy value stays the same is smaller. This can be interpreted in the way that bigger lrs lead to wider minima, in which also a larger step size can be tolerated. An alternative explanation is, that the gradient norm is larger in these cases and therefore the effect of the smaller lr becomes larger. An explanation for wider minima as a result of larger lrs could be that it is more difficult to find narrow valleys with a large lr. This is because larger lrs can jump over narrow valleys or jump out of them, when inside. This raises the question of whether training with a larger lr leads to more robust minima or smaller gradients that reduce the effect of the update step.

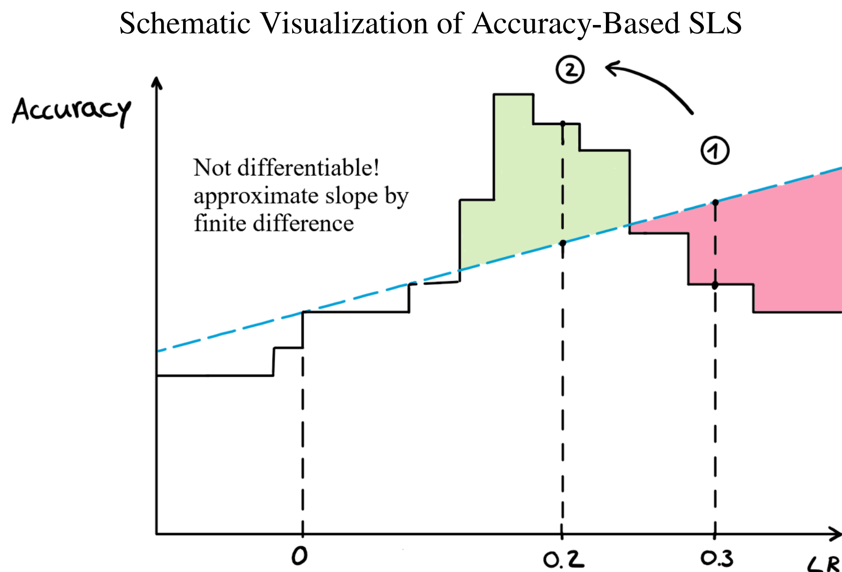

Figure 12: Lr reduction following acc-SLS. Accuracy y-axis for the parameters updated by the different lrs x-axis in the direction of the negative loss. Lr 0 depicts the accuracy for the current parameter setting. The scaled approximation (blue) is used to evaluate the quality of the update step. 1. A lr of 0.3 is chosen and compared to the corresponding point on the linear approximation. Because the corresponding value is larger than the real accuracy, the Armijo condition is not satisfied and the lr gets reduced. All lrs in the red area will be rejected and reduced until the lr falls in the green area. 2. Real accuracy is larger than a scaled approximation, so the update is performed and a new mini batch is drawn.

Overall, we are able to show dependencies between loss and accuracy. Additionally, the accuracy in the direction of the negative gradient is relatively well-behaved. This makes it plausible to transfer the lr search to the accuracy landscape even though only the gradient of the loss is available. The following section describes how we perform this line search on the accuracy and use it for training deep neural networks.

Algorithm 3: SLS for One Epoch
T stands for the number of batches per epoch.

Data: f, data, w_0, b, β, option, γ, $batch$, c, version
Result: updated weights
1 **for** k=0,..,T **do**
2 $i_k \leftarrow$ sample mini batch of size b from data
3 $lr \leftarrow$ reset(option, lr, γ, T)
4 $armijo \leftarrow$ False
5 **while** *armijo = False* **do**
6 $lr \leftarrow \beta \cdot lr$
7 $w'_k \leftarrow w_k - lr \cdot \nabla f_{ik}(w_k)$
8 d $\leftarrow$ random sample of size $batch$ from data
9 $armijo \leftarrow$ArmijoCondition(f, d, w_k, lr, version, c)
10 **end**
11 $w_{k+1} \leftarrow w'_k$
12 **end**
13 **return** w_{k+1}

5 Method

5.1 Accuracy-Based Line Search

Our optimizer is based on SGD Armijo, with the main difference being that the line search is performed on the accuracy instead of the loss. The reason therefore is, that the overall goal is to maximize accuracy and we aim to include this metric in our training. A schematic visualization of our approach can be found in Figure 12. We additionally introduce smooth accuracy, which is a continuous version of the accuracy. Even though we use this metric to compute the line search on, we still use the CE-loss for the optimization. Algorithm 13 gives details about the computations in one epoch. Before the update step is conducted, the lr is reset in the same way as in SGD Armijo [1].

Approximation of the Slope Since the accuracy is not differentiable, we have to approximate the slope in order to compute the linear approximation. We do so by using finite differences for approximating the directional derivative. To verify this, we also compute the slope for loss-based SLS (l-SLS) like this. As later will be shown, there is no significant difference between l-SLS and Armijo SGD. The approximation of the directional derivative follows Equation 26 for l-SLS and Equation 27 for a-SLS.

$$h'(0) \approx \frac{h(0) - h(\epsilon)}{\epsilon} = \frac{loss(w_k) - loss(w_k - \epsilon \cdot \nabla f_B(w_k))}{\epsilon} \tag{26}$$

$$h'(0) \approx \frac{h(\epsilon) - h(0)}{\epsilon} = \frac{acc(w_k - \epsilon \cdot \nabla f_B(w_k)) - acc(w_k)}{\epsilon} \tag{27}$$

Algorithm 4: Armijo Condition

Data: f, d, w_k, lr, version, c

Result: Armijo condition fulfilled?

1 $g \leftarrow$ function defined by version (accuracy, smooth accuracy or loss)
2 $m_0 \leftarrow g(w_k, d)$
3 $m_\epsilon \leftarrow g(w_k - \epsilon \cdot \nabla f_{ik}(w_k), d)$
4 $m_{lr} \leftarrow g(w_k - lr \cdot \nabla f_{ik}(w_k), d)$
5 **if** *version* = *loss* **then**
6 slope $\leftarrow \frac{m_0 - m_\epsilon}{\epsilon}$
7 ArmijoCondition $\leftarrow m_{lr} \leq m_0 - lr \cdot c \cdot slope$
8 **end**
9 **else**
10 slope $\leftarrow \frac{m_\epsilon - m_0}{\epsilon}$
11 ArmijoCondition $\leftarrow m_{lr} \geq m_0 + lr \cdot c \cdot slope$
12 **end**
13 **return** *ArmijoCondition*

We perform a grid search for ϵ and choose it to be 10^{-4}. For this value, the directional derivative approximates the gradient norm of the loss best. Equation 28 shows the equality of directional derivative in direction of the gradient of the loss and the gradient norm.

$$\nabla_g f(x) = \nabla f(x) \cdot \frac{g}{|g|} = \frac{g^T \cdot g}{|g|} = \frac{|g|^2}{|g|} = |g| \tag{28}$$

We choose the same value for ϵ to approximate the slope of the accuracy.

Armijo Condition The main difference between a-SLS and l-SLS is the metric, the line search is computed on. Since a-SLS performs the line search on the accuracy instead of the loss, the known minimization task becomes a maximization task instead. Specifically, the line search now aims to find lrs that lead to larger values than the linear approximation of the accuracy. Therefore, the Armijo condition is changed. Equation 29 depicts the difference between the Armijo condition used in a-SLS and l-SLS. The details of the computation of the Armijo condition for our SLS optimizer can be found in Algorithm 4. Here, the attribute *version*, determines which metric is used to compute the line search on.

$$\begin{aligned} \text{Loss: } & h(lr_\star) \leq h(0) - c \cdot h'(0) \cdot lr_\star \\ \text{Accuracy: } & h(lr_\star) \geq h(0) + c \cdot h'(0) \cdot lr_\star \end{aligned} \tag{29}$$

where $h(x)$ stands for the used metric (e.g. loss or accuracy). x determines the step size of the parameter update in direction of the negative gradient, which is used to compute the value of the corresponding metric.

In both, accuracy and loss, the slope is approximated in the direction of the negative gradient. It would be more exact to compute the norm of the gradient of the accuracy function, but because the accuracy is not differentiable this gradient does not exist. Therefore, we chose to use an

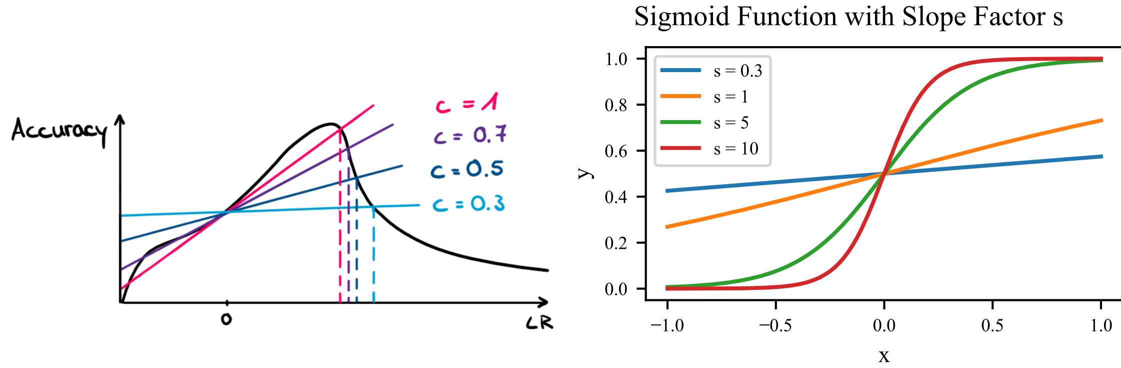

Figure 13: Different scaling for the linear approximation (left), and sigmoid functions with different slope factors (right).

Algorithm 5: Smooth Accuracy

Data: f, images, labels, slope

Result: accuracy value of given data

1 $saArr \leftarrow$ empty list
2 **for** *image, label in images, labels* **do**
3 $\quad \pi \leftarrow softmax(f(image))$
4 $\quad \hat{\pi} \leftarrow \pi$ without π_{label}
5 $\quad diff \leftarrow \pi_{label} - \max_i(\hat{\pi}_i)$
6 $\quad saArr_i \leftarrow sigmoid(diff \cdot slope)$
7 **end**
8 $sa \leftarrow mean(accArr)$
9 **return** sa

approximation by using the gradient of the loss for this computation. This approximation of the derivative in the direction of the gradient is then scaled by a factor c in the Armijo condition.

Figure 13 (left) shows, how c influences the steepness of the linear approximation. A smaller c flattens the approximation, making the condition less restrictive. As a consequence, the improvement by the current lr does not need to be as big for the lr to still get accepted.

Smooth Accuracy Since the accuracy has a derivative of 0, we run into computational difficulties. To solve these, we introduce smooth accuracy-based SLS (sa-SLS) that operates on a continuous version of the accuracy.

$$
\begin{aligned}
\delta &= \pi_{label} - \underset{max}{i}(\pi_k | l \neq label) \\
sa &= \sigma(\delta \cdot s)
\end{aligned}
\tag{30}
$$

Where $\sigma(x) = \frac{1}{1-e^{-x}}$ denotes the standard sigmoid function.

Algorithm 5 shows how this smooth accuracy is computed in sa-SLS. Here, the difference between the model output for the real label and the highest value of the non-matching labels is computed.

This difference is larger than zero if the model classifies correctly and smaller than zero otherwise. In the case of a binary classification task, if both classes get a similar value, the difference is close to 0. In this way, the insecurity of the model is captured in the values. Afterward, the resulting values are weighted by s and smoothed by the sigmoid function, as shown in Equation 30.

Since the difference values only lie between -1 and 1, the scaling factor s defines the size of the codomain. Larger values increase the codomain up to $[0, 1]$ and approximate the step-wise character of the real accuracy function, see Figure 13 (right). We conduct an ablation study to find a good slope and found out, that no scaling is needed, so 1 can be chosen as weighting.

The overall smooth accuracy value for a batch is computed as the mean over the values for the single items. Consequentially, this value is again between 0 and 1 and therefore in the same range as the accuracy. On the other side, the resulting smooth accuracy function also shares properties with the loss function. Changes in the probabilities the model returns for single classes directly affect the resulting value.

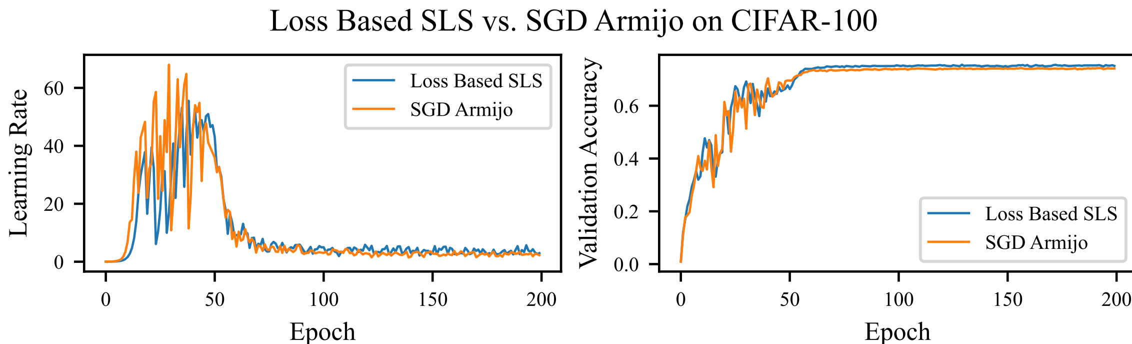

Figure 14: Development of lr (left) and accuracy (right) over the epochs for l-SLS (blue) and SGD Armijo (orange). At the beginning of training the lr reaches high values for both optimization strategies, whereas the values for l-SLS are slightly lower. At around epoch 50, the lrs become smaller and stabilize to smaller fluctuations. This similar behavior of lr size speaks for the comparability of the optimizers. It is visible that the development of the accuracy value is connected to the chosen lr. During the high lr phase, high fluctuations in accuracy are visible. After epoch 50 lr and accuracy stabilize. The behavior of the validation accuracy looks similar for both optimizers with the one of l-SLS being slightly higher. This speaks for an at least even quality of l-SLS and SGD Armijo, which means approximating the gradient norm via directional derivative is no disadvantage for computational stability.

6 Experiments

Building on the promising results of the pre-study, we implement the described l-SLS and a-SLS and benchmark them on different data sets and models. Following Vaswani, Mishkin, Laradji, *et al.* [1], we use the model-data set combinations plain MLP on MNIST [36], ResNet34 [27] on CIFAR-10 [37], and ResNet34_100 on CIFAR-100. To bring in another model architecture, we add DenseNet121 [38] as a model for ResNet34. A detailed description of the used data sets and models can be found in the Appendix. During training, we use random crops and random horizontal flips on CIFAR-10 and CIFAR-100, but no augmentations on MNIST. In opposite to Vaswani, Mishkin, Laradji, *et al.* [1] we do not use the augmented version of the data set for evaluation to ensure comparability with other benchmarking results. For all data sets we use a batch size of 128 and train for 200 Epochs.

6.1 Preliminar Experiments

Loss-Based SLS We start with a sanity check of our implementation by performing l-SLS. Therefore, we use the described strategy to adjust the lr in every optimization step. To enable comparison, we additionally train the same setting with SGD Armijo on the same seed.

The results for this comparison on CIFAR-100 with ResNet34_100 can be seen in Figure 14. The only difference between these optimizers is that in l-SLS the gradient norm is approximated by the directional derivative. To validate the computations in the experiment, we compare our computation of the gradient norm with the real gradient norm computed by the torch implementation. We only find very small computational differences (< 0.001) in the computation of the gradient norm

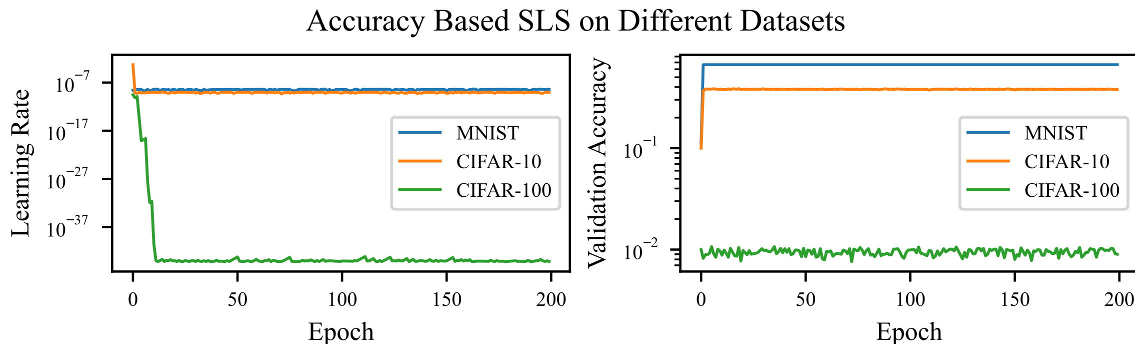

Figure 15: The lr of a-SLS optimizer over the training (left) and corresponding accuracy (right). The lr starts for all experiments at 1.0 and converges to the depicted curves during the first epoch. MNIST (blue), CIFAR-10 (orange), and CIFAR-100 (green) all show the same convergence of the lr. We find it interesting, that after a sufficiently small value is reached, the lr stays at a relatively constant value.

and the rest of the optimization procedure is the same as for SGD Armijo. Consequentially, it is not surprising that we find very similar results for l-SLS and SGD Armijo. This replication ensures that our optimizer works as intended and our computations and approximation strategies are correct.

Accuracy-Based SLS Based on the verified implementation for l-SLS, we make as few changes as possible to perform the line search on the accuracy instead. Therefore, the slope computation is changed as described in Equation 27 and the Armijo condition as described in Equation 29. Unfortunately, this does not work. As visible in Figure 15, we observe the lr to converge to zero for all tested problems. Because the lr converges to zero, accuracy and loss stagnate after a while, due to the insufficient parameter changes.

We find in a closer analysis of the optimization step that the reason for the convergence of the lr is the step-wise change in the accuracy function. At the beginning of the training, the accuracy value changes for the different lrs. In those cases, the line search finds a lr that fulfills the Armijo condition without reducing the lr too much. However, as the training performance increases, the results become more robust to changes in the parameter setting. The step-wise character of the accuracy function leads to the problem, that the reduction of the lr does not change the accuracy value. Therefore, the Armijo condition (Equation 29) is not fulfillable by increasing $h(lr_\star)$ because the function is constant in this area. $h(0), c$ and $h'(0)$ also are constant and therefore the only changing variable is $lr_\star$. This leads to the problem, that the Armijo condition only can be satisfied if the lr is small enough to reduce the right side of the inequality sufficiently. As a consequence, if a parameter setting and mini batch are reached, in which larger parts of the accuracy function are constant, the lr converges to zero. Now the problem of a small lrs emerges. Small lrs can only lead to a small change in the parameter setting and therefore lead to stagnation in accuracy and loss, see Figure 12. Because we only double the lr over one epoch the lr is unable to grow to its old size again.

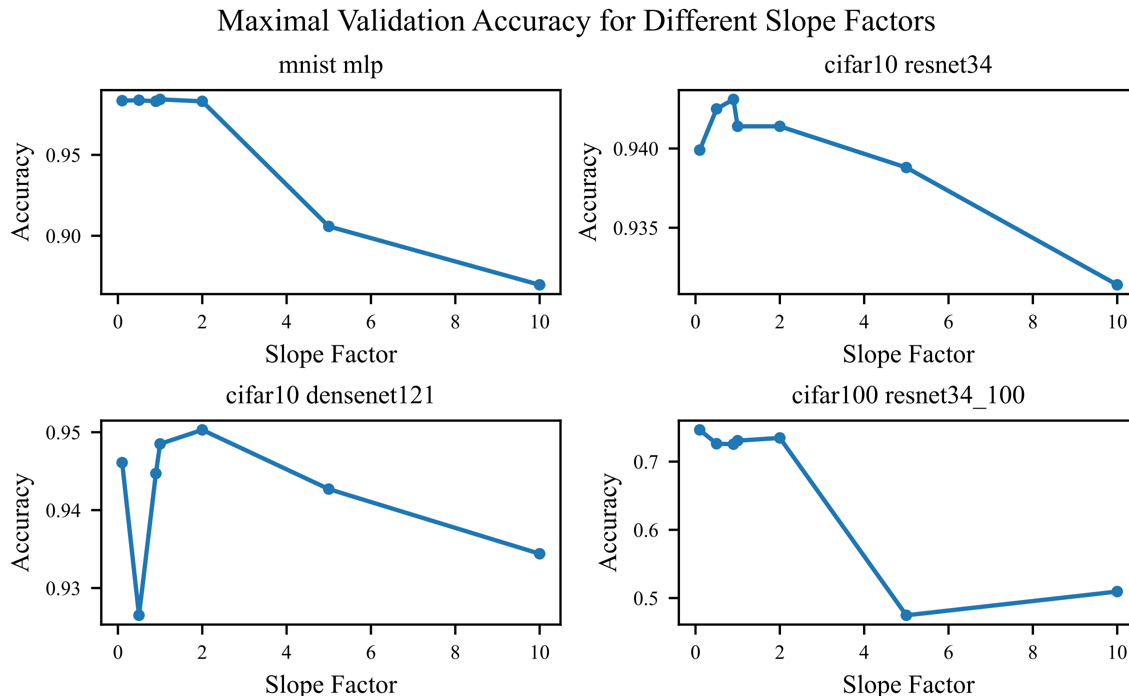

Figure 16: Maximal accuracy over the training with sa-SLS for different slope factors for the smooth accuracy. The slope factors determine how steep the steps in the accuracy landscape stay, with a lower slope factor leading to smoother curves. The results for MNIST (upper left), CIFAR-10 (upper right), CIFAR-10 on DenseNet (lower left), and CIFAR-100 (lower right) show, that the maximal reached accuracy decreases as the slope factor increases. On CIFAR-10 the maximal accuracy values do not change as much as in MNIST and CIFAR-100. Generally, the training converges for most of the different slope factors with worse results for extremely large slope factors than for small slope factors. For a slope factor of 1, all the tested problems show good performance, so we choose 1 as the slope factor for the smooth sa-SLS.

Because the lr of a-SLS converges to 0 for all our test problems due to its step-wise behavior, we use the continuous smooth accuracy for the line search instead. We conduct two ablation studies to find the best parameter setting for sa-SLS. One to find an adequate slope factor for the weak accuracy and one to find out which batch the line search should be conducted on. Afterward, we take the results from the ablation studies and implement our optimizer. This optimizer is benchmarked and compared to various commonly known optimizers on different classification problems. For the ablation studies, we train our model for 100 epochs, whereas we train in the main experiments for 200 epochs.

6.2 Ablation 1: Slope Factor Search

Our first ablation study intends to find a well-performing slope factor for sa-SLS. Therefore, we perform a grid search over different slope factors for sa-SLS. The results for the grid search on MNIST, CIFAR-10, CIFAR-10 on DenseNet, and CIFAR-100 are visible in Figure 16. As can be seen, a slope factor of 1 leads in all experiments to one of the highest accuracy, so we choose a slope factor of 1 for all further experiments.

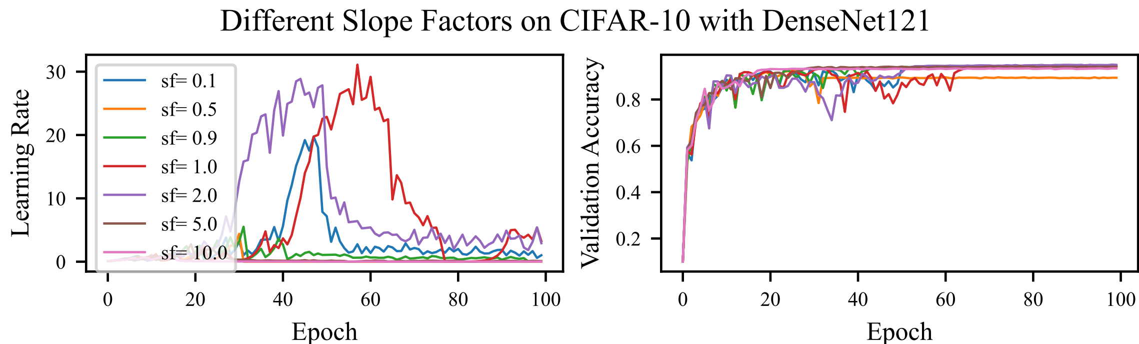

Figure 17: Lr (left) and validation accuracy (right) over the training for the different slope factors on CIFAR-10 with DenseNet121. For a slope factor of 0.1, 1, and 2 the lr increases around epoch 50 to surprisingly high values. We do not know if this is a systematic behavior for lrs on DenseNet121, but we found it interesting because no scheduling strategies use lrs around 30. The accuracy values in the epochs with large lrs are lower than in the ones with lower lr. After a few epochs, the lr becomes smaller again and the accuracy values increase. We found it surprising and promising that the SLS method tolerates temporal divergence of the lr and can stabilize the training again.

In detail, larger slope factors approximate the real accuracy function. When using a large slope factor, we more often find a convergence of the lr to zero, especially if the task is as simple as MNIST. This can be explained by the constant behavior of the accuracy function. On the other side, lead slope factors smaller than 1 to relatively good results. Therefore, smoother functions with smaller codomains do not seem to be a problem for the optimization. However, because further smoothing does not improve the results, we kept a slope factor of 1. This result can be interpreted in the way that similar to the focal loss, different versions of the accuracy can be used. Thus, as long as the function contains information about the quality of the given output and is continuous, we think, it most likely can be used for optimization.

We additionally look at the relationship between lr and accuracy. As shown in Figure 17, we find out that our optimization method tolerates the lr to diverge during training. The optimizer can reduce the lr and stabilize the training when the learning rate diverges to high values. Further research is needed to determine if choosing a large lr in the middle of training leads to improvements in maximal accuracy and how these increasing lrs are connected to the gradient norm.

6.3 Ablation 2: Search Batch

One of the results of our pre-study is about inducting bias in the training, by using the same batch for computing the gradient and performing a line search. As shown in Figure 11, the improvement of the accuracy in the direction of the negative gradient is much larger, if the evaluation happens on the same batch as the one the gradient is computed on. Following this result, we perform the line search on a different batch, to avoid this bias and get a more representative result. This would be beneficial because by choosing an adequate lr we want to increase the performance on the whole data set and not only on the current batch. To test this approach, we sample in every optimization step a second batch with the same size as the original. We use this new sample for the line search

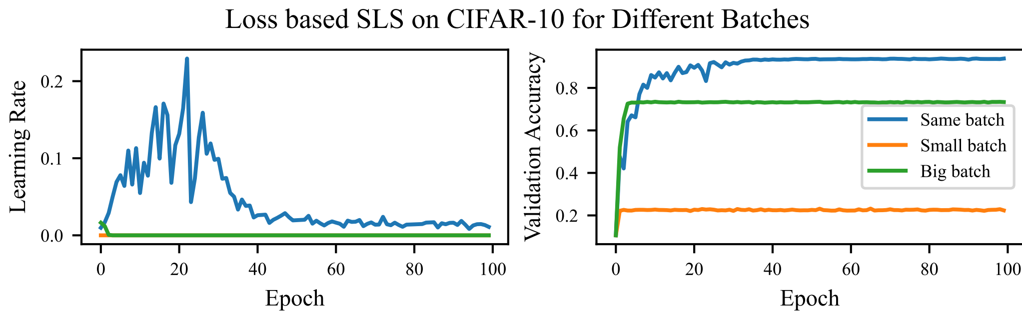

Figure 18: Lr (left) and validation accuracy (right) for training CIFAR-10 on ResNet34 with l-SLS which performs the line search on a new sample. The lr adjusts over training if the gradient is computed on the same batch (blue) but not if the gradient is computed on a different small (orange) or big (green) batch. Validation accuracy is highest for the same batch, lower for the big batch, and small for the small batch gradient computation. The reason therefore is diminishing lrs for small and big batch gradient computation. The validation accuracy for the big batch only is higher, because the lr converges slower to 0.

but compute the gradient on the old one. Additionally, following our results of more stable results on large batches, we test a second approach. Here, we use big batches containing 1000 samples to perform the line search. Because this leads to very time-consuming evaluations, we use the same approach as Mutschler [35] and only conduct the lr update every 1000th step. This can be done because the optimal lr changes slowly and so a finer tuning of the lr is not worth the extreme computational cost of evaluating the whole batch in every optimization step.

Performing the line search on a different batch did not work. We observe diminishing lrs for all data set and model combinations. Figure 18 shows the accuracy and lr for conducting the line search on different batches for l-SLS. Additional results for sa-SLS can be found in Appendix Figure 21. It is visible, that only conducting the line search on the batch the gradient is computed on leads to good results. When the line search is conducted on a small batch, the lr converges to zero very fast, and therefore the accuracy stays low. This is because the small lr is not able to make changes that have a real impact. On the large batch, this convergence to 0 is slower, because the update is only conducted every 1000th step. Consequentially, the lr gets reduced in a coarse grid and therefore the convergence is slower. The accuracy of SLS with line search on the large batch reaches higher values than on the small batch because more updates are performed with a larger lr. Despite this, the problem of diminishing lrs is still the same.

We don't know the reasons for this phenomenon jet, but one could be, that the gradient is not normed. When the line search is conducted on another batch, the gradient and its norm may not fit the data and therefore the update is applied in a direction in which maybe no lr is available that satisfies the Armijo condition. If there exists no lr that satisfies the condition, the lr gets reduced in the same way as in a-SLS. Norming the gradient could help because then the size of the update step only depends on the lr and not on the gradient norm. Another source of mistakes could be the computation of the slope. Therefore, the gradient of the first batch is used to approximate the directional derivative of the second batch. Especially, if the batches are of different sizes, the

gradients could be very different. As a result, the Armijo condition is not performing the expected test, because there are computational artifacts in the results.

In the following experiments, we use the same batch for computing the gradient and conducting the line search. This could lead to biased lrs, but we find it to be relatively good working and well-behaved.

6.4 Main Experiments

The following experiments compare our sa-SLS optimizer with other standard optimizers. We use MNIST, CIFAR-10, and CIFAR-100 with the models described above. As baseline, we use Adam, SGD0.1, SGD0.1 (lr-sched.), SGD Armijo, l-SLS and sa-SLS. Each experiment is conducted on four different random seeds.

Adam We use Adam with cosine annealing and linear ramp-up. For the momentum terms, we use the default values and for the lr, we scale the default values to adjust them to our batch size. We start with $\frac{1}{4}$ of the lr $4 \cdot 10^{-3}$ and increase it over 7 epochs until we reach this lr. This ramp-up strategy for Adam is commonly used and can be found in the literature [29]. Afterward, we decay the lr following traditional cosine annealing and additionally use a weight decay of 10^{-4}. Scheduling Adam in this way improves the performance on all data sets compared to the implementation of Adam used by Vaswani, Mishkin, Laradji, *et al.* [1]. Our scheduling of Adam leads to the same results on MNIST, gaining 1% validation accuracy on CIFAR-10 and 2.5% on CIFAR-100.

SGD SGD0.1 is plain SGD with a constant lr of 0.1. Additionally, we train with SGD0.1 with learning rate scheduling. Hereby, the lr is reduced by 0.1 at the stages 60, 120, and 160.

Line Search Optimizers SGD Armijo is the optimizer proposed and tuned by Vaswani, Mishkin, Laradji, *et al.* [1]. The SLS optimizers are self-implemented and only differ in the metric on which the line search is conducted. We choose the hyperparameters for all line search optimizers corresponding to SGD Armijo to allow comparability between the approaches. For our optimizers, the slope factor is 1 and the line search is performed on the same batch as the one the gradient is computed on.

Results Results for CIFAR-10 on ResNet34 are presented in Table 1 and show that sa-SLS competes with Adam. Results for the other experiments can be found in Appendix Figure 4, 5. These show, that as the task gets harder the performance of sa-SLS seems to increase. One can see, that sa-SLS reaches comparable accuracy values to the standard optimizers. It outperforms SGD with and without lr scheduling and reaches higher accuracy values as l-SLS and SGD Armijo. Especially considering the fact, that after a well-working version is created, no tuning between different tasks seems to be necessary, this looks very promising. Therefore, we can conclude, that performing the line search on accuracy instead of loss works and can even lead to higher accuracy values.

Optimizer	Accuracies in %	Mean Acc	Sd Acc	Mean Runtime in h
Adam	[94.37, 94.42, 94.42, 94.48]	**94.42**	0.039	2.34
SGD0.1	[93.14, 93.26, 93.45, 93.57]	93.36	0.166	2.14
SGD0.1 (lr-sched.)	[93.39, 93.46, 93.48, 93.68]	93.50	0.108	2.15
sa-SLS	[94.08, 94.20, 94.31, 94.33]	94.23	0.100	3.76
l-SLS	[93.42, 93.72, 93.77, 93.91]	93.70	0.179	3.78
SGD Armijo	[93.49, 93.63, 93.67, 93.85]	93.66	0.128	2.75

Table 1: Accuracy of ResNet34 on CIFAR-10, trained with the different optimizers (rows) for four different seeds. Columns two and three give mean accuracy and standard deviation. Column four gives the mean runtime. sa-SLS reaches after Adam the highest mean accuracy. The runtime for SLS optimizers is significantly higher than the runtime of the other optimizers due to the additional forward pass and the conducted line search.

Optimizer	Accuracies in %	Mean Acc	Sd Acc	Mean Runtime in h
Adam	[98.31, 98.37, 98.37, 98.39]	98.36	0.030	0.43
SGD0.1	[98.33, 98.35, 98.40, 98.41]	98.37	0.033	0.38
SGD0.1 (lr-sched.)	[98.33, 98.35, 98.39, 98.41]	98.37	0.032	0.49
sa-SLS	[58.80, 98.15, 98.31, 98.37]	88.41	17.094	0.54
l-SLS	[98.34, 98.38, 98.40, 98.41]	**98.38**	0.027	0.51
SGD Armijo	[98.35, 98.36, 98.37, 98.41]	98.37	0.023	0.68

Table 2: MNIST on plain MLP trained with different optimizers. For as-SLS one of the runs reaches significantly lower validation accuracy (red).

However, this increase in performance comes at the cost of higher computational costs during optimization. The runtime comparison shows that the SLS optimizers need significantly more computation time than Adam and SGD. In opposite to our optimizer, the runtime of SGD Armijo is relatively close to the runtime of Adam even though it also performs line searches during training. This difference in runtime, between SGD Armijo and SLS comes from the additional forward pass in each optimization step needed for approximating the slope via the directional derivative. To verify this, we compare the number of conducted forward and backward passes for sa-SLS, l-SLS, and SGD Armijo. We find out that our optimizers perform the same amount of backward passes but twice as many forward passes as SGD Armijo (see Appendix Table 6). It is also interesting to see that sa-SLS and l-SLS have similar runtime and number of forward passes. This only occurs if they conduct a similar number of line search steps during optimization, which speaks for the comparability of loss and accuracy landscape and a well-behaved line search.

Whereas most of the experiments show well-behaved results, some of the runs with as-SLS suffered from diminishing lrs. As it can be seen in Table 2, for MNIST on plain MLP one of the runs stopped early and reached a significantly lower validation accuracy. We find that the lower validation accuracy is due to diminishing lrs in this run (see Appendix Figure 22).

7 Discussion

We developed an SGD-oriented optimizer that automatically chooses its lr by conducting a line search on a smoothed accuracy function. Therefore only lrs that increase the smoothed accuracy more than a scaled linear approximation at this point are used for training. We performed ablation studies to find out which slope should be used in the smoothed accuracy function and on which batch this line search should be conducted. We tested our opinion on standard benchmark tasks (MNIST, CIFAR-10, and CIFAR-100). Our results showed that our optimizer can compete against commonly used optimizers, like Adam and SGD.

Smooth Accuracy Performing a line search to find the lr on the accuracy function directly did not work due to the discontinuity of the accuracy function. To solve this issue, we developed a smooth version of the accuracy. This smooth accuracy includes the certainty of the model to find a continuous interpolation between correct and wrong classification by using a sigmoid function. Using the original accuracy function was not possible, because in constant areas reducing the lr did not change the accuracy value. If the accuracy value stayed the same, the only way to fulfill the Armijo condition was to reduce the lr. This led to diminishing lrs and therefore to stagnation of the training.

Diminishing Lr Although we solved this problem by using smooth accuracy, we still found diminishing lrs in the course of some experiments which led to a drop in performance. This happened in 3 of the performed 40 runs. This phenomenon occurred on different tasks, but always only for specific seeds when optimizing with sa-SLS. The reason for this behavior is open for further research and questions the reliable applicability of the proposed optimizer. Generally, it speaks against the stability of the method if small changes can lead to unexpected maleficent behavior.

Bias As we found out in the pre-study, using the same batch for computing the gradient and performing the line search leads to bias. This means, that, in theory, performing the line search on the whole batch should lead to a smaller optimal lr. We designed two strategies to avoid this bias, both computing the line search on a new drawn batch. In contrast to our assumptions, performing the line search on a different batch to compensate for the batch bias did not increase overall performance. We assume an interaction with the gradient norm of the batch the gradient is computed on. In the future, it would be interesting to investigate these approaches further by using normed gradients for performing the line search and updating the weights.

Hyperparameters Adaptive lr methods aim to reduce hyperparameters and fine-tuning needed for scheduling lrs. Unfortunately, line searches or other strategies for adaptively setting the lr also introduce new hyperparameters. Though, in line search, the initial lr influences mostly the beginning of the training. It is open for further research to analyze how stable SLS can deal with different lr start points. Since we already observe that sa-SLS can deal with momentarily diverging

lrs during training, we are confident that the initial lr won't have that big of an impact.

Related to this, the decrease in the lr for every failed line search step β and the number of line searches performed per optimization step n need to be determined. These influence how fast the lr can decrease, as a smaller β leads to faster changes, and therefore determines how flexible the lr can be adapted and how easily it can converge to zero. The number of performed line search steps per optimization step plays a central role in the lr diminishing. In the cases where reducing the lr is not able to satisfy the condition, an exhaustive search can occur which leads to a lr reduction of β^n.

Since these parameters only allow the lr to be reduced, there is also the need for a reset function that increases the lr. This function is in our case chosen in a way that the lr is doubled over each epoch. However other increment methods and sizes could work and potentially lead to better performance. These hyperparameters influence how the lr de- and increases. Our setting seems to be a good balance between some restrictions to avoid diminishing lrs and still being flexible enough to learn with an adequate lr.

In addition to that the Armijo condition for accepting a certain lr plays a prominent role in our optimizer. The scaling factor c is a hyperparameter that qualitatively determines how large the improvement needs to be for a lr to get accepted. Smaller c lead to a less restrictive condition and, because the lr starts with a large value and gets reduced, to larger lrs. Choosing a different c could lead to different optimization dynamics and eventually avoid the observed lr divergence.

Another hyperparameter that influences the Armijo condition more indirectly is the chosen ϵ for computing the approximation of the directional derivative. An ϵ that is chosen too large can lead to wrong values due to ignorance against changes in a smaller grid. But also choosing ϵ very small can lead to problems because numerical instabilities can emerge. We compare the result of different ϵ values for approximating the gradient norm via the directional derivative with the current gradient norm of the loss. This leads to our ϵ of 10^{-4}.

Using line search for choosing the lr, therefore, creates a lot of hyperparameters that need computational resources for fine-tuning. But as we could show in our benchmarking results, where we used the same values for sa-SLS as for SGD Armijo [1], even a not perfectly tuned setting leads to good results that can compete with other scheduled lr strategies. This speaks for the stability of the parameters and we hope that the same settings can be used on different problems. Avoiding the adjustment of the lr scheduler for every new problem could justify the effort of fine-tuning needed to construct the optimizer. To evaluate if the parameter setting is stable enough to overall save computation time further experiments on a variety of tasks would be needed.

Different Metrics We believe that the SLS strategy could also work on slightly different continuous accuracy-oriented functions. These could for instance use other slope factors for scaling the sigmoid function, choose a different function for the interpolation, or make changes in computing the differences between the real label and the model output. We found our setting to work suffi-

ciently but other implementations could lead to unexpected gain or drop in performance. It is also possible, that the robustness problem could be fixed by choosing a different strategy to smooth the accuracy function. The optimal smooth accuracy function can also depend on the used data set and model. We already observed different relationships between the size of the slope factor and the maximal achieved accuracy on different data set model combinations 16. Therefore, investigating different metrics and performing stochastic line searches on them remains an interesting topic for the future.

Additionally, our research raises the question of how strictly cross-entropy loss is set as a metric for optimization. We developed a continuous function built on accuracy that allows backward passes. This and the fact, that performing line searches on it worked, makes the smooth accuracy a potential metric for optimization. We think that other kinds of interpolations between accuracy and loss would also work as a metric. Using these functions could weigh in the accuracy during the training and therefore avoid some problems that can occur in training with cross-entropy loss.

Large Learning Rates Observations of the lr chosen by SLS optimizers showed surprisingly high lrs for some tasks during the training. Mutschler [35] observed a similar behavior. This raises the question if these large lrs might have a beneficial effect on the training. For making meaningful conclusions about this, the gradient norm of the loss needs to be taken into account because it stands in direct interaction with the lr and the performed update step.

However, it should hold that large lrs lead to bigger update steps than small lrs on average. This could lead to finding minima in a larger area and because the lrs are large the found minima would probably be relatively wide. Furthermore, we assume this because training with large lrs probably jumps over or out of narrow minima. It would be interesting to conduct further research on the structure of accuracy and loss landscape when different kinds of optimization strategies are used. This could be done by creating random projection plots during the training for the completely trained networks. We hypothesize the minima found by sa-SLS to be wider than the ones found by competing optimization strategies. Wider minima, in general, are preferred because they are hypothesized to yield higher generalizability [39]. Further research needs to be conducted to find out if this behavior occurs and if it is beneficial for training.

Computational Cost Related to this the question emerges if the higher computational cost is worth the performance of the optimizer. As visible in the runtime analysis, the update step based on line search strategies took longer to perform. But since the lr is adjusted during training, grid searches for tuning the lr become obsolete, and therefore it can be argued, that the overall computational cost is similar or even lower. Especially, if sa-SLS can be used without tuning on different tasks, this would save a lot of computation time. sa-SLS takes less than twice the computation time as Adam. Thus, if two or more different hyperparameter settings need to be tested in a grid search, time could actually be saved by using sa-SLS. Apart from this consideration, there might be possibilities to reduce the computational cost of sa-SLS. Mutschler [35] proposes

to perform the lr update on larger batches in a coarser grid and to use a fixed lr in between. This could also be applied to sa-SLS. Fewer lr adjustments would lead to a reduction of computation time and could maybe even lead to more stability in the training. It stays open for further research if this could lower the computational cost and increase performance.

Limitations There are some limitations to the conclusions we drew because we only used a small set of test problems. The used tasks are MNIST, CIFAR-10, and CIFAR-100, whereas MNIST is generally seen as being too easy to reveal qualitative differences in optimizers. Even SGD without any lr tuning reaches extremely high accuracy values within a few epochs. Therefore MNIST is not a good task for evaluating a lr adjusting strategy. The remaining tasks CIFAR-10 and CIFAR-100 are relatively similar as they only differ in the number of classes the pictures contain. To achieve more convincing results for image classification the optimizer should be tested on ImageNet [40]. To test the quality of the optimizer it should also be tested in completely different domains and tasks such as speech recognition.

Another aspect that influences the interpretability of our results are the optimizers we used as a benchmark. Since we mostly concentrated on developing sa-SLS we can not guarantee an optimal hyperparameter tuning for SGD and Adam. But since Adam tuned with our hyperparameter setting achieves higher accuracy values as the Adam implementation tuned by Vaswani, Mishkin, Laradji, *et al.* [1], we think the comparison is fair.

The most concerning problem of sa-SLS lies in the robustness of the optimizer. For some seeds, the training stopped early due to diminishing lrs. We were not able to detect the source of this behavior but we think gradient clipping or using normalized gradients could help to avoid it. Overall, we developed a working optimizer that chooses its lr based on a line search performed on a smoothed accuracy function.

References

[1] S. Vaswani, A. Mishkin, I. Laradji, M. Schmidt, G. Gidel, and S. Lacoste-Julien, "Painless stochastic gradient: Interpolation, line-search, and convergence rates", *Advances in neural information processing systems*, vol. 32, 2019.

[2] H. Robbins and S. Monro, "A stochastic approximation method", *The annals of mathematical statistics*, pp. 400–407, 1951.

[3] M. D. Zeiler, "Adadelta: An adaptive learning rate method", *arXiv preprint arXiv:1212.5701*, 2012.

[4] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.", *Journal of machine learning research*, vol. 12, no. 7, 2011.

[5] A. Schoenauer-Sebag, M. Schoenauer, and M. Sebag, "Stochastic gradient descent: Going as fast as possible but not faster", *arXiv preprint arXiv:1709.01427*, 2017.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks", in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", *Advances in neural information processing systems*, vol. 25, 2012.

[10] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling", *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1915–1929, 2012.

[11] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler, "Joint training of a convolutional network and a graphical model for human pose estimation", *Advances in neural information processing systems*, vol. 27, 2014.

[12] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černockỳ, "Strategies for training large scale neural network language models", in *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*, IEEE, 2011, pp. 196–201.

[13] G. Hinton, L. Deng, D. Yu, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups", *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[14] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks", in *2013 IEEE international conference on acoustics, speech and signal processing*, Ieee, 2013, pp. 6645–6649.

[15] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks", *Advances in neural information processing systems*, vol. 19, 2006.

[16] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks”, in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, Springer, 2014, pp. 818–833.

[17] K. He and J. Sun, “Convolutional neural networks at constrained time cost”, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 5353–5360.

[18] S. Kullback and R. A. Leibler, “On information and sufficiency”, *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[19] J. Kukačka, V. Golkov, and D. Cremers, “Regularization for deep learning: A taxonomy”, *arXiv preprint arXiv:1710.10686*, 2017.

[20] S. Ruder, “An overview of gradient descent optimization algorithms”, *arXiv preprint arXiv:1609.04747*, 2016.

[21] N. Qian, “On the momentum term in gradient descent learning algorithms”, *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[22] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization”, *arXiv preprint arXiv:1412.6980*, 2014.

[23] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization”, *arXiv preprint arXiv:1711.05101*, 2017.

[24] S. Hanson and L. Pratt, “Comparing biases for minimal network construction with back-propagation”, *Advances in neural information processing systems*, vol. 1, 1988.

[25] Y. Wu, L. Liu, J. Bae, *et al.*, “Demystifying learning rate policies for high accuracy training of deep neural networks”, in *2019 IEEE International conference on big data (Big Data)*, IEEE, 2019, pp. 1971–1980.

[26] P. Tseng, “An incremental gradient (-projection) method with momentum term and adaptive stepsize rule”, *SIAM Journal on Optimization*, vol. 8, no. 2, pp. 506–531, 1998.

[27] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition”, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[28] I. Loshchilov and F. Hutter, “Sgdr: Stochastic gradient descent with warm restarts”, *arXiv preprint arXiv:1608.03983*, 2016.

[29] Z. Liu, Y. Lin, Y. Cao, *et al.*, *Swin transformer: Hierarchical vision transformer using shifted windows*, 2021. arXiv: `2103.14030 [cs.CV]`.

[30] J. Dean, G. Corrado, R. Monga, *et al.*, “Large scale distributed deep networks”, *Advances in neural information processing systems*, vol. 25, 2012.

[31] L. Armijo, “Minimization of functions having lipschitz continuous first partial derivatives”, *Pacific Journal of mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

[32] P. Wolfe, “Convergence conditions for ascent methods”, *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[33] F. Schneider, L. Balles, and P. Hennig, “Deepobs: A deep learning optimizer benchmark suite”, *arXiv preprint arXiv:1903.05499*, 2019.

[34] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection", in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[35] M. Mutschler, "Empirics-based line searches for deep learning", Ph.D. dissertation, Universität Tübingen, 2023.

[36] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: 10.1109/MSP.2012.2211477.

[37] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images", 2009.

[38] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[39] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur, "Sharpness-aware minimization for efficiently improving generalization", *arXiv preprint arXiv:2010.01412*, 2020.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[41] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1998.

8 Appendix

8.1 Data Sets

We conduct several experiments to test the developed optimizer. We use the dat sets MNIST, CIFAR-10, and CIFAR-100 and different model architectures. These are plain MLP, ResNet, and DenseNet.

MNIST contains pictures of 20×20 pixels with handwritten numbers from 0 to 9 on it. The pictures are centered and the corresponding label contains the number that is written on the picture. The numbers are written by over 500 different persons and converted to a grey scale. The data set is divided into 60.000 training images and 10.000 test images with their corresponding labels.

CIFAR-10 consists of colored pictures with the size 32×32 which show objects of 10 classes. These are airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Their labels are given as numbers between 0 and 9, which give the index of the right class in the array of classes. The training set contains 50.000 images and the test set contains 10.000, each with its corresponding label. CIFAR-100 shares the properties of CIFAR-10 but consists of 100 different classes and therefore labels between 0 and 99. The 100 classes are divided into 20 superclasses, which contain umbrella terms like flowers, which then contain the ultimate classes like orchids.

MLP On MNIST, we use a plain multi-layer perceptron (MLP) [41] which is one of the simplest deep neural networks because it consists of three or more layers that are fully connected and feedforward. This means each node in layer i is connected to each node in layer $i + 1$ in a way that allows information only to flow upward.

ResNet Because networks with more layers have in general a higher capacity, it seems logical to add more and more layers. Unfortunately, if more than the needed layers are given, the optimization of the network becomes more difficult. One problem is vanishing gradients which can lead to a performance decrease as He and Sun [17] observed. To solve this problem, Residual Networks (ResNets) [27] with added identity connections are used. These building blocks are formally defined as:

$$x_i = H_i(x_{i+1}) + x_i \tag{31}$$

whereas the residual mapping, which should be learned is represented as $H_i(x_{i+1})$. These skip-connections allow easy propagation of the gradient and therefore tackle the vanishing gradient problem. ResNet34, for instance, describes a residual network with 34 layers. We use ResNet34 on CIFAR-10 and ResNet34_100 on CIFAR-100.

DenseNet Another approach to solving this problem is DenseNet [38], which has connections between an underlying layer to all following layers with the same feature-map size. In opposite to ResNet, these skipping connections are not added to the output but are used as additional input.

$$x_i = H_i([x_0, x_1, ..., x_{i-1}]) \tag{32}$$

This concatenation similarly reduces the vanishing gradient problem by enabling an easy gradient propagation.

DenseNet121 and ResNet34 are known to work well on CIFAR-10 [38] [27] so we use DenseNet121 as a further comparison model on CIFAR-10.

8.2 Figures and Tables

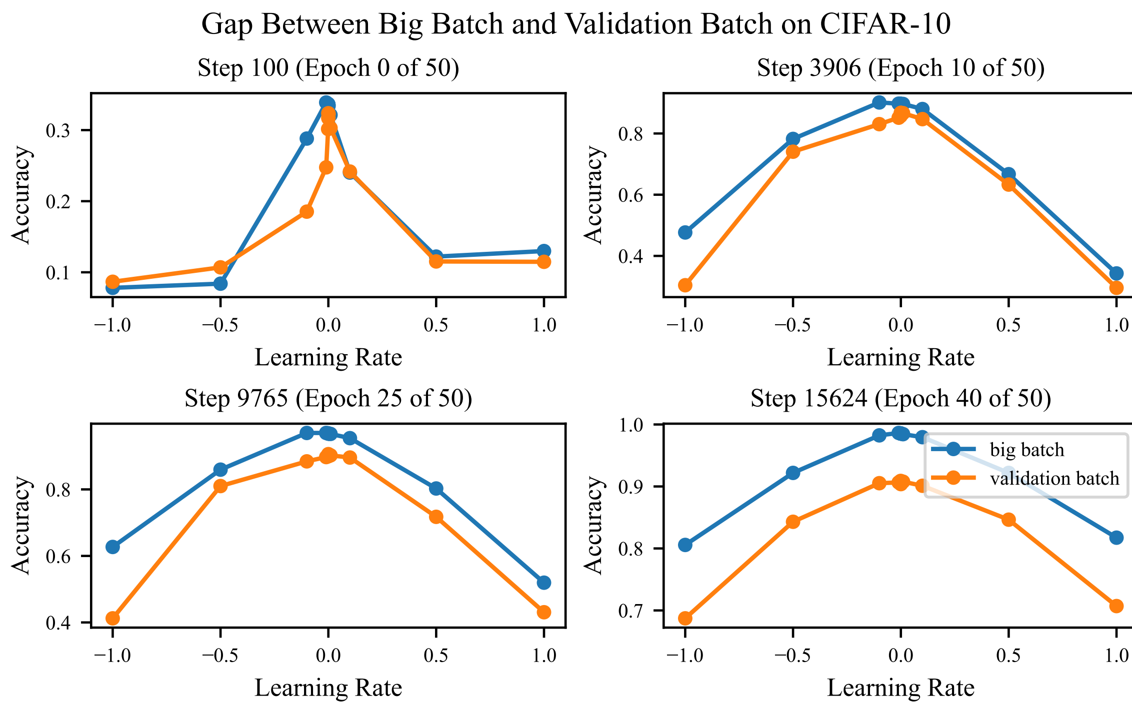

Figure 19: CIFAR-10 on ResNet34. Widening gap between big batch and validation batch due to overfitting. The same behavior can also be found in the other data sets and models.

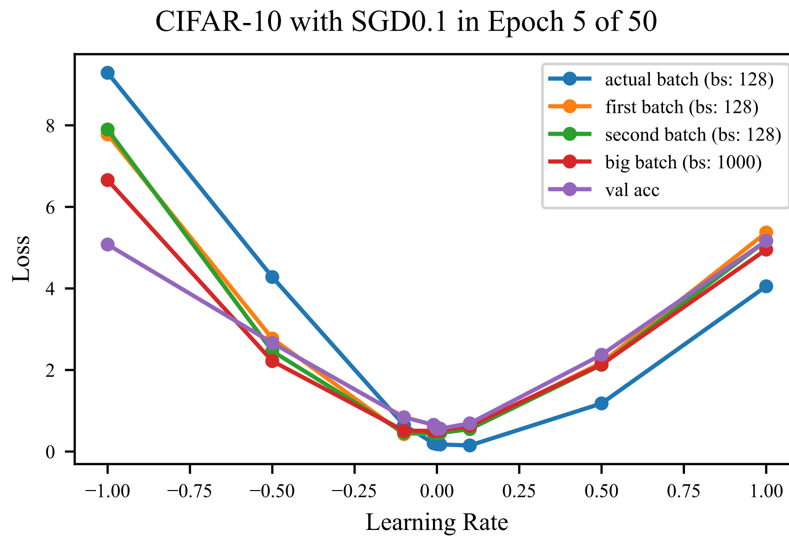

Figure 20: Representative example: CIFAR-10 on ResNet34. Loss landscape in direction of the negative gradient sampled by different lrs. Same batch (blue), different, but same-sized batch (orange, green), and the validation batch (violet) all show systematic differences. Larger batches (red) lead to smoother behavior. Different small batch curves fluctuate around the big batch curve. The same batch shows bias by having lower loss values for positive lr and higher values for negative lr as the other batches. The validation set also shows asymmetry in loss landscape.

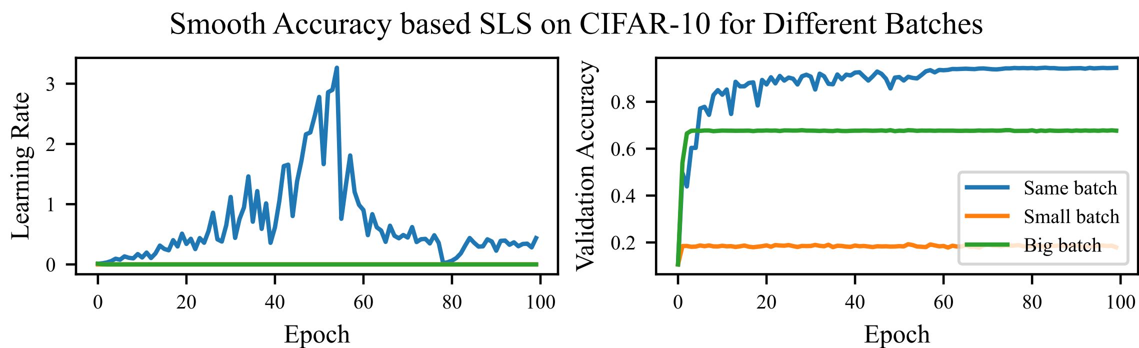

Figure 21: Lr (left) and validation accuracy (right) over the training for the different batches. The lr adjusts over training if the gradient is computed on the same batch (blue) but not if the gradient is computed on a different small (orange) or big (green) batch. Validation accuracy is highest for the same batch, lower for the big batch, and really small for the small batch gradient computation. The reason therefore is diminishing lrs for small and big batch gradient computation.

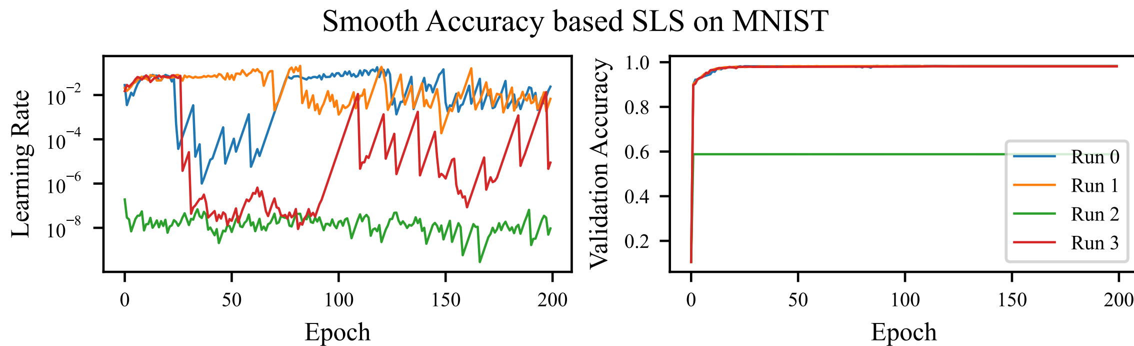

Figure 22: Lr of sa-SLS, when used for training MNIST on plain MLP. The lr (y-axis) shows similar behavior over the epochs (x-axis) for most of the runs(0,1,3) but diminishes for one of the seeds (2).

lr	acccuracy
-1.0000	0.062500
-0.5000	0.343750
-0.1000	0.796875
-0.0100	0.968750
-0.0010	**0.984375**
-0.0001	**0.984375**
0.0000	**0.984375**
0.0001	**0.984375**
0.0010	**0.984375**
0.0100	**0.984375**
0.1000	**0.984375**
0.5000	0.625000
1.0000	0.265625

Table 3: Accuracy values for different lrs on CIFAR-10 trained with SGD0.1 in step 1953, epoch 5 of 50. The same accuracy values for lrs between -0.001 and 0.1 due to step wise character of the accuracy landscape.

Optimizer	Accuracies in %	Mean Acc	Sd Acc	Mean Runtime in h
Adam	[94.45, 94.61, 94.63, 94.90]	94.65	0.162	5.25
SGD0.1	[93.40, 93.55, 93.72, 93.84]	93.63	0.167	6.04
SGD0.1 (lr-sched.)	[93.77, 93.84, 93.93, 93.95]	93.87	0.072	5.70
sa-SLS	[94.53, 94.79, 95.18, 95.24]	**94.94**	0.291	8.88
l-SLS	[94.15, 94.26, 94.26, 94.29]	94.24	0.053	9.77
SGD Armijo	[94.04, 94.10, 94.24, 94.25]	94.16	0.090	6.58

Table 4: CIFAR-10 trained on DenseNet121. Maximal accuracy in the benchmarking experiments for the different optimizers. The second and third columns give the mean and standard deviation for the maximal accuracy. The last column gives the mean runtime in hours for the different optimizers. Much longer runtime for optimization on DenseNet as for other networks.

Optimizer	Accuracies in %	Mean Acc	Sd Acc	Mean Runtime in h
Adam	[71.86, 72.19, 73.03, 73.11]	72.55	0.536	2.30
SGD0.1	[73.61, 73.68, 73.78, 74.19]	73.81	0.225	2.18
SGD0.1 (lr-sched.)	[73.21, 73.40, 73.55, 74.01]	73.54	0.296	2.29
sa-SLS	[69.63, 74.15, 75.26, 75.44]	73.62	2.356	3.70
l-SLS	[75.28, 75.33, 75.55, 75.71]	**75.47**	0.173	3.78
SGD Armijo	[74.32, 74.89, 75.03, 75.38]	74.91	0.382	2.76

Table 5: CIFAR-100 trained on ResNet34_100. Maximal accuracy in the benchmarking experiments for the different optimizers. The second and third columns give the mean and standard deviation for the maximal accuracy. Runs with diminishing lrs are marked red. The last column gives the mean runtime in hours for the different optimizers.

Optimizer	fw passes	fw per steps	bw passes	bw per steps
sa-SLS	315156.00	4.040462	78000.0	1.0
l-SLS	315198.00	4.041000	78000.0	1.0
SGD Armijo	157537.75	2.019715	78000.0	1.0

Table 6: CIFAR-10 with ResNet34. Mean number of forward and backward passes performed during the whole training and per optimization step. Training on CIFAR-10 has 390 optimization steps per epoch. Our SLS optimizer performs twice as much forward passes as SGD Armijo.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln und Quellen angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder veröffentlicht. Zudem versichere ich, dass das in Dateiformat eingereichte Exemplar mit dem dem eingereichten gebundenen Exemplar übereinstimmt.

Tübingen, October 1, 2023

OSANE HACKEL