Institute of Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Investigating the Influence of Learning Rates on the Learning Speed of Neural Networks

Robin Sasse

Course of Study: Informatik B.Sc.

Examiner: Prof. Dr.-Ing. Andrés Bruhn

Supervisor: M.Sc. Jenny Schmalfuß

Commenced: April 14, 2021

Completed: October 1, 2021

CR-Classification: G.1.6

Abstract

This Bachelor's Thesis investigates the effects of learning rates on the learning speed of Residual Neural Networks, training on the CIFAR-10 and CIFAR-100 data sets. Besides the optimal constant learning rate setting, we discuss the option of learning rate scheduling and calculating the learning rate. Cyclical schedules with large maximum learning rates are used to recreate a phenomenon called super-convergence, which speeds up the training procedure by as much as orders of magnitude and leads to better generalization capabilities of the network. We present an intuition as to why cyclical learning rates lead to better regularization of the network. We show that super-convergence can be reproduced for the optimizer Adam by introducing cyclical learning rates. Lastly, a method which calculates the learning rate, rather than requiring it as a hyper-parameter, is investigated. This algorithm promises to use statistical element-wise curvature information to automatically tune the learning rate for each iteration and each parameter separately. We show that while the approach of calculating the learning rate is valid, it neither leads to super-convergence nor to a higher validation accuracy achieved by the network when compared to the ones trained with cyclical learning rates.

Kurzfassung

In dieser Bachelorarbeit untersuchen wir die Effekte von zyklischen Lernraten auf die Konvergenzgeschwindigkeit Residualer Neuronaler Netze, welche auf dem CIFAR-10 und dem CIFAR-100 Datensatz trainiert werden. Neben der optimalen Einstellung konstanter Lernraten, werden dynamische ("scheduled") und berechnete Lernraten analysiert. Wir rekreiren ein Phenomän, genannt Super-Convergence, welches durch zyklische Lernraten, mit außergewöhnlich großen Maximal-Lernraten, hervorgerufen wird und das Training um ein vielfaches beschleunigt. Als positiver Seiteneffekt generalisiert das trainierte Netzwerk außerdem besser. Wir liefern eine intuitive Erklärung der Ursachen von Super-Convergence und der besseren Regularisierung der Netze. Des Weiteren zeigen wir, dass Super-Convergence auch für den Optimierer Adam emergiert wenn zyklische Lernraten verwendet werden. Zusätzlich analysieren wir eine Methode, welche die Lernrate berechnet, anstatt sie als Parameter übergeben zu bekommen. Dieser Algorithmus verspricht über statistische, elementweise Kurveninformationen die Lernrate, für jeden Parameter des Netzes separat, zu bestimmen. Wir zeigen, dass dieser Ansatz valide ist. Jedoch zeigen wir ebenfalls, dass die Berechnung der Lernrate auf diese Art, im Gegensatz zur Nutzung zyklischer Lernraten, weder zu einer höheren Validation Accuracy, noch zu Super-Convergence führt.

Contents

1. Introduction 7

2. Optimization Methods for Neural Networks 11
   - 2.1. ResNet Architectures . . . 11
   - 2.2. Stochastic Gradient Decent . . . 15
   - 2.3. Momentum-Based Descent Methods . . . 17
   - 2.4. Hyper-Parameter Settings in Training Neural Networks . . . 21
   - 2.5. Super-Convergence . . . 26
   - 2.6. AdaSecant . . . 28

3. Implementations and Extensions of the Theoretical Foundations 35
   - 3.1. Standard Implementations and Libraries . . . 35
   - 3.2. Combining Momentum and Cyclical Learning Rates . . . 36
   - 3.3. Implementing AdaSecant . . . 38

4. Experiments 43
   - 4.1. Set-Up of Experiments and Inter-Dependencies of Hyper-Parameters . . . 43
     - 4.1.1. Choice of Data Sets . . . 43
     - 4.1.2. Choice of Networks . . . 44
     - 4.1.3. Choice of Optimizers . . . 45
     - 4.1.4. Epoch Length Settings . . . 45
     - 4.1.5. Learning Rate Schemes and Ranges . . . 46
     - 4.1.6. Choice of Batch Size . . . 47
     - 4.1.7. Decay Rate Settings . . . 49
   - 4.2. Overview of the Results . . . 50
   - 4.3. SGD and General Insights into Super-Convergence . . . 55
     - 4.3.1. CLR Schemes Converge Faster . . . 55
     - 4.3.2. Low Sensitivity to Learning Rate Range . . . 57
     - 4.3.3. Super-Convergence Emerges from Cyclical Nature of CLR . . . 57
     - 4.3.4. Super-Convergence is Independent of ResNet Depth . . . 60
     - 4.3.5. Effects of Super-Convergence Increase when Training Data is Limited . . . 61
     - 4.3.6. Summary . . . 61
   - 4.4. Super-Convergence for Adam . . . 62
     - 4.4.1. CLR Schemes Converge Faster . . . 62

4.4.2. Low Sensitivity to Learning Rate Range . . . 64
4.4.3. Super-Convergence is Independent of ResNet Depth . . . 65
4.4.4. Effects of Super-Convergence Increase when Training Data is Limited . . 65
4.4.5. CUS-Adam . . . 67
4.4.6. Summary . . . 68
4.5. AdaSecant . . . 68
4.5.1. Re-Implementation of AdaSecant in PyTorch . . . 68
4.5.2. Simple AdaSecant Provides Convergence . . . 68
4.5.3. Investigating the Learning Rates of Simple AdaSecant . . . 70
4.5.4. Summary . . . 72

5. Conclusion and Outlook 75

A. Appendix 79
A.1. Relationship between Adam's second momentum vector and the distance covered by the gradients . . . 80
A.2. Results Using Torchvision Models . . . 81

Bibliography 83

1. Introduction

Deep neural networks have achieved tremendous success in recent years [1]. While they find application in many areas, the research field of image classification has seen especially strong advances through deep learning. Residual Neural Networks (ResNets) have been successfully used to achieve high accuracy on data sets such as the CIFAR-10 and CIFAR-100 image classification data sets [2, 3]. These image classification data sets consist of 50,000 training images and 10,000 validation images, each. The CIFAR-10 data set provides 10 distinct labels, i.e. classes, for these images. The CIFAR-100 data set contains 100 distinct classes. While the overall performance of ResNets on these data sets is high, they need well over 100 epochs to fully converge to these results when using conventional training settings [2]. The fine-tuning of hyper-parameters, especially of the learning rate, remains an open topic for research [1]. An optimal learning rate setting can lead to faster learning speeds for neural networks. This speed-up is of interest to us as it means that less valuable computational resources and time are spend on the training of neural networks, which is a very time-consuming and resource-intensive task for most problems. As a secondary effect the resulting networks may even achieve a higher overall performance than those trained with less optimal learning rate settings. This is obviously another desirable effect as it provides better classifiers.

In this thesis we investigate the influence of learning rates on the learning speed of neural networks. For this, a phenomenon called super-convergence, a term which was coined by Smith and Topin [1], is investigated. Super-convergence arises when cyclical learning rates (CLR) are used with stochastic gradient descent (SGD). The learning rate ranges for these cycles use very large maximum learning rates during the midsection of a cycle and small or very small minimum learning rates in the beginning and end of a cycle. It has previously been shown that these CLR schemes lead to a speed-up of training by order of magnitude [1, 4]. We validate this observation and extend its workings to function with Adam [5], a popular momentum-based descent algorithm. In this thesis various reasons for the emergence of super-convergence are discussed and evidence supporting or contradicting these ideas is presented. Furthermore, the sensitivity of the training results to the exact choice of learning rate ranges used is investigated. Lastly, the inter-dependencies of CLR, network depth, and training data available are analyzed.

As an alternative to setting the learning rate, an optimizer which calculates it, called AdaSecant [6], is examined an compared to super-convergence in terms of convergence quality and

speed. As there currently exists no implementation of this algorithm in PyTorch[1], we propose one. As an alternative to AdaSecant, a simpler variation, based on the same underlying concept, is introduced and conveniently named Simple AdaSecant. Both implementations are evaluated with regards to their influence on the training speed and quality of neural networks and the learning rates they calculate. This learning rate is then compared to the optimal cycle settings found for SGD with CLR.

The contributions of this work include:

1. The extension of systematic analysis on super-convergence.
2. The validation of the speed-up of training neural networks with CLR.
3. The proposal and examination of a new explanation for the success of CLR and the examination of alternative explanations.
4. The implementation of a second-order directional Newton Method with a secant approximation of the Hessian for optimization and its evaluation.

Structure of this work

This work is structured as follows:

Chapter 2 – Optimization Methods for Neural Networks: Here the theoretical fundamentals of this work are laid out. Firstly, ResNet architectures and their advantages for this thesis are introduced and explained in *2.1 – ResNet Architectures*. In *2.2 – Stochastic Gradient Decent* the basic concept underlying SGD is explained, followed by the introduction of momentum in *2.3 – Momentum-Based Descent Methods*. We will further discuss the settings of hyper-parameters in *2.4 – Hyper-Parameter Settings in Training Neural Networks*. In *2.5 – Super-Convergence* the speed-up of the training procedure of neural networks caused by CLR is explained. Lastly, *2.6 – AdaSecant* an optimizer that does not require the setting of the learning rate is introduced.

Chapter 3 – Implementations and Extensions of the Theoretical Foundations This chapter discusses implementations and extensions to existing optimizers made for the purposes of this thesis. In *3.1 – Standard Implementations and Libraries* we briefly show the libraries and standard implementations used in this work. Two adaptations to Adam are proposed in *3.2 – Combining Momentum and Cyclical Learning Rates*. These adaptations have the purpose of introducing update step size cycles to the optimizer in order to allow for super-convergent-like behavior with Adam. Lastly, in *3.3 – Implementing AdaSecant* a

[1] https://github.com/pytorch/pytorch/tree/88032d894311e5c0aed8bbc21a4306bc6be4af82

proposal for the implementation of AdaSecant in PyTorch is discussed. Furthermore, a simplification to AdaSecant is proposed as an alternative to using AdaSecant.

Chapter 4 – Experiments This chapter discusses the main part of this work, which is the assessment of different learning rate schemes. *4.1 – Set-Up of Experiments and Inter-Dependencies of Hyper-Parameters* discusses the set-up of the experiments and the reasons behind the choices made, especially regarding the setting of hyper-parameters other than the learning rate. In *4.2 – Overview of the Results* we present a summary of the findings of these experiments. Details of these results are discussed separately, sorted by optimizer, in *4.3 – SGD and General Insights into Super-Convergence*, *4.4 – Super-Convergence for Adam*, and *4.5 – AdaSecant*.

Chapter 5 – Conclusion and Outlook This chapter concludes this thesis and gives an outlook on future work that can be done to complete the findings of this work.

2. Optimization Methods for Neural Networks

The focus of this chapter is on the preceding research that was used as a foundation for any further development of algorithms and experiments. Furthermore, we use this chapter to introduce the most important notations used in the upcoming parts of this work. In Section 2.1 the neural network architecture used in this thesis is laid out. Sections 2.2 and 2.3 discuss classical and momentum-based descent methods for solving the optimization problem (i.e. minimizing the loss function). The algorithms presented are later used in the experiments of Chapter 4. We discuss the hyper-parameter settings in training in Section 2.4 as they have an influence on the results of training. In Section 2.5 a phenomenon that speeds up the training process by using CLR is presented. This phenomenon called super-convergence was a main driver that motivated this thesis. The results are later replicated and compared to other learning rate choices in Sections 4.2–4.5. Finally, Section 2.3 discusses the option to calculate the learning rate for each iteration of the optimization algorithm. This is an interesting approach as calculating the learning rate could potentially replace the necessary task of setting a learning rate if the results are promising.

2.1. ResNet Architectures

Since the main focus of this work is setting the learning rate of a neural network, this introduction to the type of network used in this work is kept brief. A ResNet is a type of (convolutional) neural network which is optimized for very deep learning. It is significantly better suited for high number of layers than plain (convolutional) neural networks [2]. The reason for this lies in the special architecture of the ResNet and the difficulty of learning the identity function.

While a neural network's performance on new data might decrease with more layers due to overfitting, this should not be the case for its performance on the training data. More specifically, if Network A has more layers than Network B, Network A should yield at least the same accuracy on the training data as Network B. The reason being that if Network B has n_B layers and Network A has $n_A = n_B + n_\Delta$ layers, Network A could theoretically learn the first n_B layers just as they were learned by Network B. For the following n_Δ layers Network A could simply learn the identity function. Thus, Network A would produce the exact same results as Network B. But in practice this could not be observed [7].

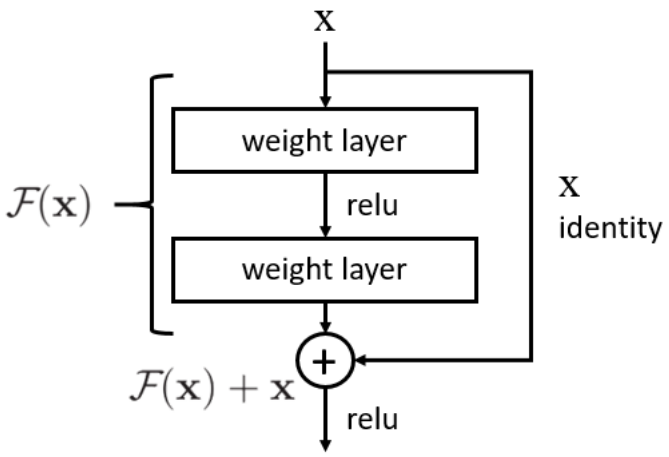


Figure 2.1.: The skip connection as a building block of a ResNet [2].

The problematic assumption here is that the identity function can be learned easily. In reality this is a rather complicated function to learn for a neural network. ResNets therefore use skip connections, as indicated in Fig. 2.1. It is easy to learn a function that maps all inputs to zero, especially when exploiting the activation function. Therefore, the ResNet can easily learn the identity mapping between two non-subsequent layers by finding a zero mapping and adding it to the identity function supplied by the skip connection.

We define $\mathcal{F}_{i'}$ as the function of the i'-th building block of the ResNet's plain counterpart (i.e. the function of the building block without skip connection); see Fig. 2.1 for a visual representation. Then the i'-th building block $\mathcal{H}_{i'}$ of a ResNet can be defined as [2]

$$\hat{y}_{i'} := \mathcal{H}_{i'}(x_{i'}, \{\theta_i\}_{i=l_{i'},...u_{i'}}) \tag{2.1}$$
$$= \mathcal{F}_{i'}(x_{i'}, \{\theta_i\}_{i=l_{i'},...u_{i'}}) + x_{i'}, \tag{2.2}$$

where $x_{i'} = \hat{y}_{i'-1}$, $i' \in \{1, ..., n'\}$ is the input of the building block's first layer. The output of the building block is $\hat{y}_j$. The term θ_i refers to the i-th parameter of the network, with $l_{i'}$ and $u_{i'}$ denoting the first and last index of the parameters found within building block i'. The ResNet consists of n' such residual building blocks $\mathcal{H}_{i'}(x_{i'}, \{\theta_i\}_{i=l_{i'},...,u_{i'}})$. The total network has n parameters θ_i, $i \in \{1, ..., n\}$.

We further define the final output of the ResNet $\mathcal{R}$ with input x as[1]

$$\hat{y} := \mathcal{R}(x) \tag{2.3}$$
$$= \mathcal{F}_{n'+1}(\hat{y}_{n'}) \tag{2.4}$$
$$= \mathcal{F}_{n'+1}(\mathcal{H}_{n'}(\mathcal{H}_{n'-1}(...(\mathcal{H}_1(\mathcal{F}_0(x)))))). \tag{2.5}$$

[1]The parameters as an argument of the function are implied and therefore left out for the purpose of simplicity.

with $\mathcal{F}_{n'+1}$, $\mathcal{F}_0$ being the first and last building blocks of the network, which are non-residual. This then yields

$$x_1 = \mathcal{F}_0(x) \tag{2.6}$$

as the first residual building block's first input. A complete visual representation of a 34-layer ResNet is shown in Fig.2.2

In this thesis we use f to denote the objective function (i.e. the loss function). This notation was chosen over the notations l or L because it is the term used in the three most important papers for this thesis [1, 5, 6]. The respective loss of the ResNet's prediction is, just as for any other network, defined as

$$f(y, \hat{y}) = f(y, x, \theta) \tag{2.7}$$
$$= f(y, \mathcal{R}(x, \theta)) \tag{2.8}$$

with y being the actual target (i.e. the correct label or value) associated with x and f being the loss function (e.g. cross-entropy loss, mean squared error). We adjust the network's parameters θ in order optimize f.

Li et al. [8] argue that using skip connections in neural networks modifies the geometry of the loss function in an important way. The loss function of a ResNet is much smoother than the loss function of a plain neural net, see Fig. 2.3. A smooth surface of the loss function is much more suited for optimization problems. Usually increasing a networks size will make the loss function more rough. Therefore, very deep neural networks might not be well-suited for training as their associated loss functions may be too irregular. This is another explanation for why large ResNets train better than their plain counterparts.

Because ResNets are specialized for very deep learning, we use them as the architecture in the experiments of this thesis. The smoothness of the loss function yields flat saddles and saddle-like structures. This we can exploit when setting the learning rate; the details of which are explained in 2.4 when we discuss CLR. For those two reasons, ResNets are a favorable option for the experiments performed in this work. Furthermore, ResNets have been used in the work of Smith and Topin 2.5 whose results we are trying to replicate and compare to other learning rate choosing techniques. From a stance of comparability, using ResNets is the preferred approach as well.

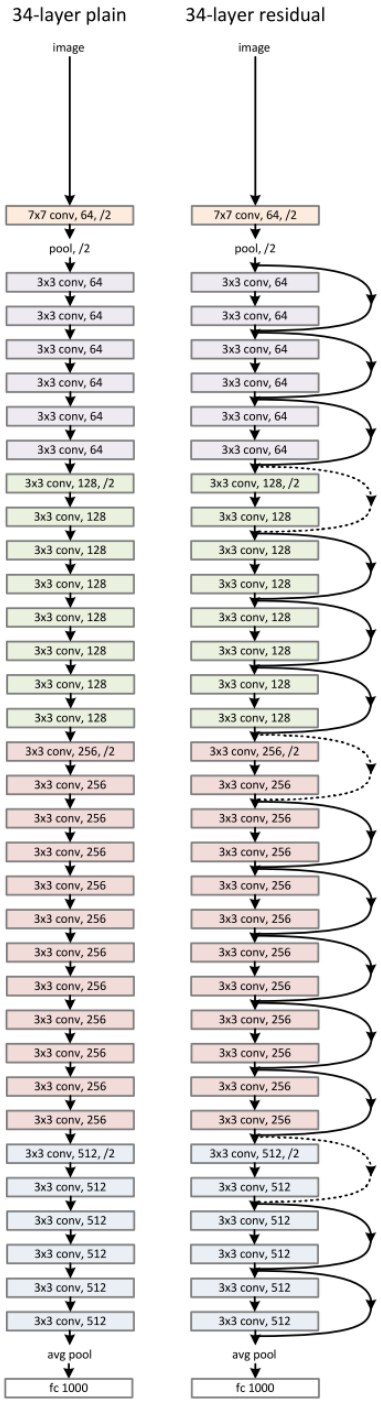
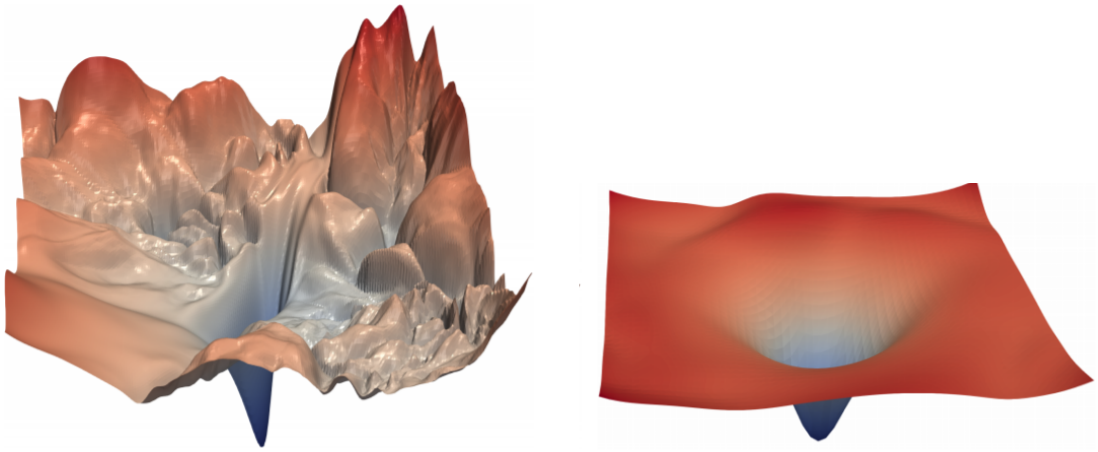


Figure 2.2.: Comparison of a 34-layer ResNet with a 34-layer plain network and the VGG-19 architecture. Image source [2].



(a) Plain 56 layer network (without skip connections).

(b) ResNet56 (with skip connections)

Figure 2.3.: The loss surfaces of a 56 layer neural network on the CIFAR-10 data set with and without skip connections. Image source [8].

2.2. Stochastic Gradient Decent

A crucial step on the path of developing a deep learning model lies in solving the underlying optimization problem. SGD is one of the most popular optimizers due to its simplicity [9]. Nonetheless, weaknesses such as inherent noise have lead to the development of newer optimizers [9]. While nowadays more refined algorithms, such as Adam [5], are gaining more popularity, SGD still remains a useful algorithm in practice and an essential theoretical foundation for more advanced algorithms. SGD is derived from the Method of Steepest Descent [10].

The Method of Steepest Descent is used to find a local minimum of a function by iteratively moving in the direction of the steepest descent d. The distance for which this direction is followed is defined through a scalar α. Suppose we want to minimize a loss function $f(y, \mathcal{R}(x, \theta))$ associated with a ResNet $\mathcal{R}$. The inputs x and targets y are given by the training data and are therefore fixed. Furthermore, the function $\mathcal{R}(x, \theta)$ refers to the forward pass of the input x by the ResNet $\mathcal{R}$ with parameters θ. Therefore, the functionality of $\mathcal{R}$ is defined by the ResNet architecture. Hence, we can rewrite the optimization problem as minimizing the loss function $f(\theta)$. In order to achieve this we must adjust the network's parameters θ towards the direction of the vector d with a scalar step size α. The direction is evaluated using the current parameter settings. As the parameters θ change, so does the direction of steepest

descent d. Therefore, from the current position we can only determine the direction of steepest descent for an infinitesimally small scalar α. We want to find the one d which minimizes

$$\lim_{\alpha \to 0} f(\theta + \alpha d) = d^{\mathrm{T}} \nabla_\theta f(\theta). \tag{2.9}$$

Solving this [10], we obtain the direction of steepest descent:

$$d = -\nabla_\theta f(\theta). \tag{2.10}$$

The update of the network parameters follows in the direction of the negative gradient, evaluated for the current parameter settings. Practically speaking, we obviously cannot limit the algorithm to an infinite amount of infinitesimally small update steps. Therefore, we pick a scalar $\alpha > 0$ as the learning rate. The size of the learning rate conveys how much confidence is put in the update step:

$$\theta^{(j+1)} = \theta^{(j)} - \alpha \nabla_\theta f(\theta^{(j)}), \tag{2.11}$$

with j referring to the j-th iteration of the update routine. We call α the learning rate. Smaller values for the learning rate mean that a step is more likely to be optimal in direction. Larger values for the learning rate mean that less evaluations must be performed to traverse a fixed distance. Finding an optimal setting for the learning rate of an optimizer is the central objective of this thesis.

Goodfellow, et al. [11] describe the difference between SGD and the Method of Steepest Descent (i.e. Gradient Descent) as the introduction of a separation of the training data. In SGD one views the gradient as an expectation over the number of samples. Therefore, instead of using the full data set available for calculating the gradient, the data $D = \{(x_1, y_1), ..., (x_{n_D}, y_{n_D})\}$, which consists of n_D values and corresponding labels, can be separated into smaller batches $B = \{(x_{l_B}, y_{l_B}), ..., (x_{u_B}, y_{u_B})\} \subset D$. Each batch contains a subset of $u_B - l_B$ samples from the full data set. For each batch the (expected) gradient can be calculated as

$$g = \frac{1}{u_B - l_B} \sum_{k=l_B}^{u_B} \nabla_\theta f(y_k, x_k, \theta). \tag{2.12}$$

SGD then follows this stochastic gradient downhill. Thus, we obtain the update step

$$\theta^{(j+1)} = \theta^{(j)} - \alpha g^{(j)}. \tag{2.13}$$

for $j \in \{1, ..., n_{iter}\}$ where $n_{iter} = n_{epochs} \cdot n_{batches}$ denotes the number of total iterations performed.

Algorithm 2.1 shows the complete optimization via SGD.

Algorithmus 2.1 SGD

Require: The parameters of the network θ.
Require: The learning rate α.
Require: The data set D.
Require: The objective function f.
Require: The network architecture $\mathcal{R}$.
  procedure SGD(θ, α, D, f, $\mathcal{R}$)
    while stopping criterion is not met **do**
      for all $B \subset D$ **do** // for batch B in data set D
        $x \leftarrow \text{GETVALUES}(B)$
        $y \leftarrow \text{GETLABELS}(B)$
        $g \leftarrow 1/|B| \cdot \sum_{k=1}^{|B|} \nabla_\theta f(y_k, \mathcal{R}(x_k, \theta))$ // compute average gradient for samples in B
        $\theta \leftarrow \theta - \alpha \cdot g$
      end for
    end while
  end procedure

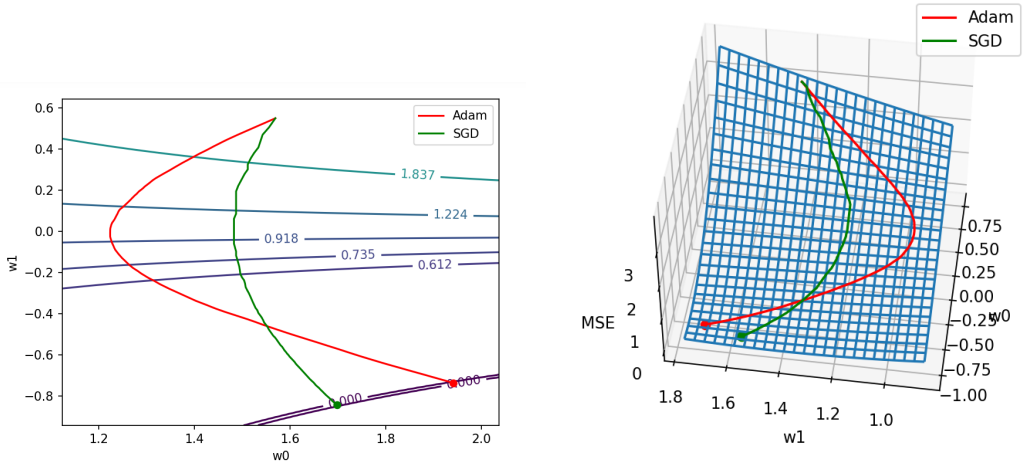
2.3. Momentum-Based Descent Methods

Plain SGD does not perform well in all areas where neural networks are used (e.g. robotics) [9]. Therefore, in this section we discuss momentum-based descent methods as alternative optimizers. We can take SGD as explained above and generalize it by introducing a momentum vector m ($m^{(j)}$ refers to the momentum vector of the j-th iteration). This momentum replaces the gradient in the update step. The resulting descent resembles the trajectory of a heavy ball rolling down the loss function [10]. Introducing momentum can cause the descent to escape small ditches (i.e. local minima) and has been shown to reduce oscillation and provide faster convergence [10]. We define the update routine as [12]

$$m^{(0)} = 0, \tag{2.14}$$
$$m^{(j)} = \beta m^{(j-1)} + (1-\beta) g^{(j)}, \tag{2.15}$$
$$\theta^{(j+1)} = \theta^{(j)} - \alpha m^{(j)} \tag{2.16}$$

Then classical SGD is just a special case of the equation above where $\beta = 0$.

The idea of heavy-ball momentum strongly resembles that of using an expected value over the gradients. Consider $m^{(j)}$ to be a moving average for g. Therefore, $m^{(j)}$ is a good estimator for $E[g]$, the expected value of g. This only holds true for large j (i.e. after many iterations have passed) because we initialize the first momentum vector with $m^{(0)} = 0$ instead of $m^{(0)} = g^{(1)}$ which would yield a true moving average. Nonetheless, since we usually perform many iterations and therefore $m^{(0)}$ becomes exponentially less influential, this intuition still holds true over the major part of training. The stochastic implication of this is a more stable descent



(a) Isolines indicating the values of $f(w_1, w_2)$.

(b) Grid indicating the values of $f(w_1, w_2)$.

Figure 2.4.: Comparison of optimization with Adam and SGD. The slope of the descending curve for Adam shows the effects momentum on the descent line. Each minimum found by each algorithm is equally good.

due to more well-balanced directional vectors and less distortions caused by outliers. This is because all gradients are weighted with $(1 - \beta)$ which is usually rather small [5, 13].

Liu et al. [12] show that momentum-based SGD has the same convergence bound as classical SGD, even if a function has multiple local minima. Better yet, in practice these methods even tend to outperform plain SGD [10, 12]. Fig. 2.4 is a visual comparison of the descent lines formed by Adam (a special case of momentum-based descent – details follow in the next paragraphs), compared to those of SGD. We clearly observe the added momentum in the descent of Adam. Both algorithms terminate on a different minimum. Nonetheless all the minima are equally good solutions to the optimization problem.

Adam

There are many forms of momentum in gradient descent, such as the algorithm proposed by Nesterov [14] and the stochastic optimization methods AdaGrad [15] and AdaDelta [16]. In this work we will focus on a more modern momentum-based descent algorithm called Adam [5]. The name Adam is derived from adaptive moment estimation. Besides the advantages of including momentum, Adam is associated with two properties that form the core idea of the algorithm. Firstly, Adam is invariant to the scaling of the gradients. Secondly, Adam performs some form of step size annealing, meaning that towards later iterations (i.e. larger values for j) the step size decreases. In the following we will explain why these properties are desirable and how they arise.

First of all, we analyze the invariance to the gradients' scale and why other forms of (momentum-based) descent do not yield this property. Since the learning rate α has been a constant scalar, so far differences in the size of the update steps $\Delta^{(j)} = -\alpha g^{(j)}$ between the iterations j only depended on the scale of the gradient g. The possible size of the update step can be described as a region of confidence. For (momentum-based) SGD the region of confidence is determined by the product of learning rate and gradient. The steepness of the gradient and the direction to which it points determine the size and direction of the update step and thus, the confidence in the correctness associated with it. Fig. 2.5(a) illustrates the region of confidence depending on the size and direction of the gradient. Rather than showing a single gradient, the possibilities of what the gradient might look like are indicated. The idea is to show that instead of being uniformly set, the region of confidence depends on the actual stochastic gradient.

Nonetheless, the scale of the gradient is not necessarily an indicator for the optimal scaling of the update step. If we examine Fig. 2.3 from Section 2.1, we actually observe that the steepest gradients correspond to smaller regions that can easily be overshot with large update steps. Flatter regions, with less steep gradients, tend to span larger distances. Therefore, the difference in the approach for Adam is to set a uniform region of confidence which is invariant to the re-scaling of the gradients. The update step $\Delta^{(j)}$ is approximately bounded by the learning rate α [5]:

$$\left|\Delta^{(j)}\right| \lessapprox \alpha. \tag{2.17}$$

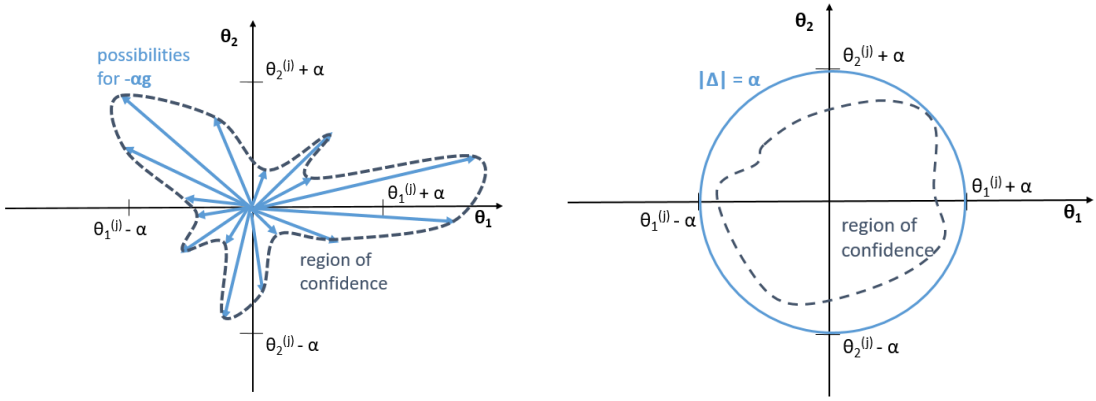
Thus, α forms a trust region around the current parameters' values, beyond which the current gradient estimation does not provide sufficiently reliable information. Further descent within the iteration is therefore discouraged by the learning rate settings. Fig. 2.5(b) illustrates this idea.

In order to achieve this, Adam uses two momentum vectors instead of one. The momentum vector $m^{(j)}$ and its bias-corrected version $\hat{m}^{(j)}$ are closely related to the momentum vector introduced for momentum-based SGD. The second momentum vector $v^{(j)}$ and its bias-corrected version $\hat{v}^{(j)}$ are tied to the distance already traversed by the end of iteration j. The update step is defined as [5]

$$\theta^{(j+1)} = \theta^{j} - \alpha \cdot \frac{\hat{m}^{(j)}}{\sqrt{\hat{v}^{(j)}}} \tag{2.18}$$

$$\approx \theta^{j} - \alpha \cdot \frac{E[g]}{\sqrt{E[g^2]}}. \tag{2.19}$$

The bias-corrected first momentum vector $\hat{m}^{(j)}$ is a good estimator for the expected value of the gradients regardless of the iteration j. This is because the bias correction term undoes the initial bias towards zero, which we observed in momentum-based SGD.



(a) SGD (or momentum-based SGD if gradient g is substituted for momentum m) with arrows indicating possibilities for scaled gradients one might encounter at this position.

(b) Adam with the region of confidence limited by the learning rate.

Figure 2.5.: Regions of confidence indicating how large the update step Δ can be.

The expected value over the gradient g, approximated by the bias-corrected first momentum vector $\hat{m}^{(j)}$, is formed over positive and negative values for g. The expected value over the squares of the gradients g^2, approximated by the bias-corrected second momentum vector $\hat{v}^{(j)}$, is formed over positive values only. Therefore, over time the root of the expected values of the gradients squared becomes larger than the expected value of the gradients. This gives rise to Adam's second advantageous property. Because

$$\lim_{j \to \infty} \Delta^{(j)} = \lim_{j \to \infty} \left(\alpha \cdot \frac{\hat{m}^{(j)}}{\sqrt{\hat{v}(j)}} \right) = 0 \tag{2.20}$$

Adam naturally performs some form of update step annealing. Step size annealing can be advantageous because the distance between the current solution and the optimal solution might become too small towards later iterations [17]. If not treated by reducing the step size towards later iterations, the algorithm oscillates around the optimum [10]. As an alternative to using Adam, update step annealing for (momentum-based) SGD through learning rate scheduling is discussed in Section 2.4.

The pseudocode for Adam is given in Algorithm 2.2. Please note that while $var^{(j)}$ refers to the variable in the j-th iteration, var^j refers to the variable raised to the power of j. This holds true not only for the algorithm but also for any mathematical statement found in this thesis. For additional information on Adam, which is not relevant to understanding the rest of this thesis but might nonetheless be of interest to the reader, we kindly refer to Appendix A.1.

Algorithmus 2.2 Adam

Require: The parameters of the network θ.
Require: The learning rate α. Recommended setting is $\alpha = 0.001$.
Require: The exponential decay rates $\beta_1, \beta_2 \in [0, 1)$. Recommended settings are $\beta_1 = 0.9$, $\beta_2 = 0.999$.
Require: The data set D.
Require: The objective function f.
Require: The network architecture $\mathcal{R}$.

```
procedure ADAM(θ, α, β1, β2, D, f, R)
    m ← 0                                              // initialize first momentum vector
    v ← 0                                              // initialize second momentum vector
    j ← 0
    while stopping criterion is not met do
        for all B ⊂ D do                               // for batch B in data set D
            j ← j + 1
            x ← GETVALUES(B)
            y ← GETLABELS(B)
            g ← 1/|B| · Σ_{k=1}^{|B|} ∇θ f(y_k, R(x_k, θ))   // compute average gradient for samples in B
            m ← β1 · m + (1 − β1) · g                  // update the first momentum estimate
            v ← β2 · v + (1 − β2) · g^2                // update the second raw momentum estimate
            m̂ ← m/(1 − β1^j)                           // compute bias-corrected first momentum estimate
            v̂ ← v/(1 − β2^j)                           // compute bias-corrected second raw momentum estimate
            θ ← θ − α · m̂/(√v̂ + ε)                     // update parameters, recommended setting is ε = 10^−8
        end for
    end while
end procedure
```

2.4. Hyper-Parameter Settings in Training Neural Networks

The aforementioned algorithms require special hyper-parameters to be set by the user. Setting these hyper-parameters will have a non-negligible effect on an algorithm's performance in training the network. These effects are either the object of research of this thesis (learning rate and scheduling and their effects on number of epochs required) or they must be accounted for (batch size and momentum) when performing experiments with these algorithms. In this section we will provide an overview of the most common hyper-parameters required by descent algorithms. Note that these are the most well-known and most widely used parameters. However, there might by other highly specialized training environments requiring additional hyper-parameters that we cannot cover here.

Epochs

Firstly and maybe most importantly, the number of iterations performed by the descent algorithm has to be defined. In Algorithms 2.1 and 2.2 we iterated for as long as the "stopping criterion is not met." In practice, this usually is defined by specifying a number of epochs for which to perform the loop. We do this because it usually cannot be known when a minimum is reached. In other words, we usually do not know what the optimum value $f_{optimum}$ for the loss function f will be. Thus, we cannot define a condition which depends on such a value (e.g. "while $f > f_{optimum}$"). The preferred solution is therefore to perform a predefined number of epochs and take the optimal state of the neural network, found within this period. This optimal state is the released network.

A single epoch is defined as a complete run through the training data. An iteration is a single update step of the iterative optimizer. The number of iterations performed by the optimizer n_{iter} equals the number epochs n_{epochs} times the number of batches $n_{batches}$ in the data set.

Batch Size

In Section 2.2 we explained that we can separate our data set into batches. Technically speaking, the most important aspect of this choice is the advantage we gain on computational power needed [10]. Calculating the loss and gradient for smaller batches uses less GPU memory. This might be an essential modification for us to even be able to perform the necessary calculations since data sets tend to be very large containing at least many tens of thousands of samples, possibly even millions.

However, we also have to acknowledge that our choice of the batch size will directly affect our results. Keskar et al. [18] argue that large batch sizes converge to sharp minima. Rather than giving a theoretical explanation they show their results experimentally. Fig. 2.6 illustrates why sharp minima are problematic. The likelihood of a sharp minimum, evaluated on the training data, coinciding with a near-minimum point of the loss function, evaluated on the test data, is less than when we are at a flat minimum. Training with large batch sizes might yield trained networks which perform just as well on the training data as those that were trained using smaller batches. Nonetheless, the solutions to the optimization problem in Fig. 2.6 do not generalize well on new data. Normally, an optimal batch size lies in the range between 32 and 512 samples [18].

Please note that it is neither certain that sharp minima cannot generalize for deep networks, nor is the definition of a sharp minimum undisputed [19]. Nonetheless, the findings of Keskar et al. [18] are frequently cited and remain an important foundation for explaining the effect of batch sizes on training success [1, 20, 21]. Takase et al. [22] speculate that large batch sizes stabilize the loss function, which makes escaping a poor local minimum less likely. The

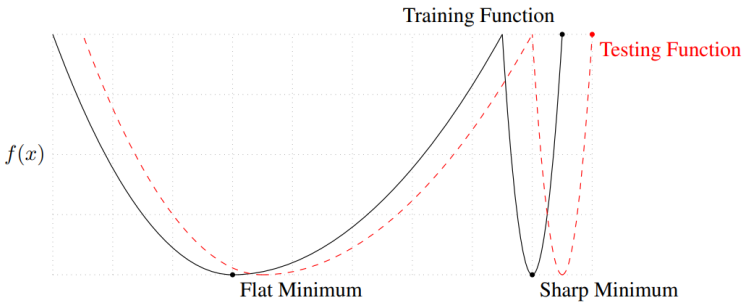


Figure 2.6.: Sketch of both a sharp and a flat minimum found within the loss function. The y-axis shows the values of f, the loss function, and the x-axis indicates the value of the parameter of the model. Image source [18].

quintessential taking from this subsection is that we have to account for the choice of batch size in the upcoming experiments of Chapter 4.

Learning Rate

The learning rate α is the main focus of this work. Initially the learning rate was defined as a constant scalar. Essentially, the learning rate conveys the degree of confidence in the current gradient. When α is large we strongly trust the gradient and take large steps in its opposite direction. When α is small we have little faith in the extent to which this gradient represents any future descent direction. Thus, our steps are small in the direction of the negative gradient $-\nabla_\theta f(\theta)$.

If α is small, we gradually move towards a local minimum. Nonetheless, too small steps mean that the descent algorithm needs to perform many iterations. Each iteration is very costly as it involves large amounts of data to be processed. If α is too large, on the other hand, the optimizer might not converge at all. Fig. 2.7 shows this diverging process.

Later some configurations to the learning rate are introduced. These configurations are scheduling and vectorization – see the next subsection and Section 2.6 respectively.

Learning Rate Scheduling

So far we have considered the learning rate α to be a fixed value across all iterations of the descent algorithm. We can extend the previously introduced algorithms (SGD and Adam) by making $\alpha^{(j)}$ a scheduled variable whose value depends on the iteration $j \in \{1, ..., n_{iter}\}$.

One way a scheduler can be constructed is by introducing a decay rate γ for the learning rate $\alpha^{(j)}$. This decay rate is used as a factor ($\gamma < 1$) which decays the learning rate in

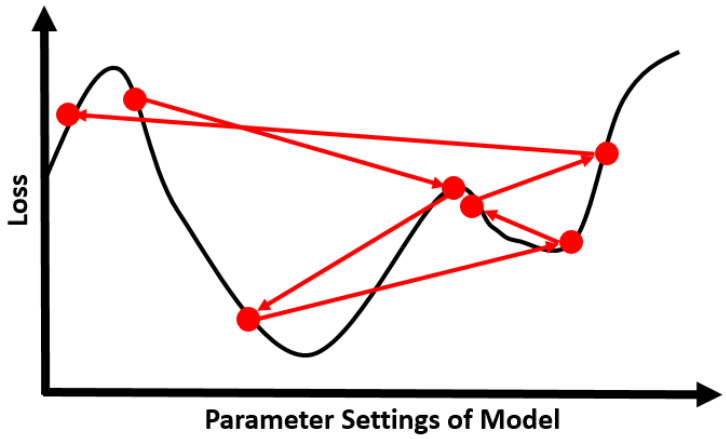


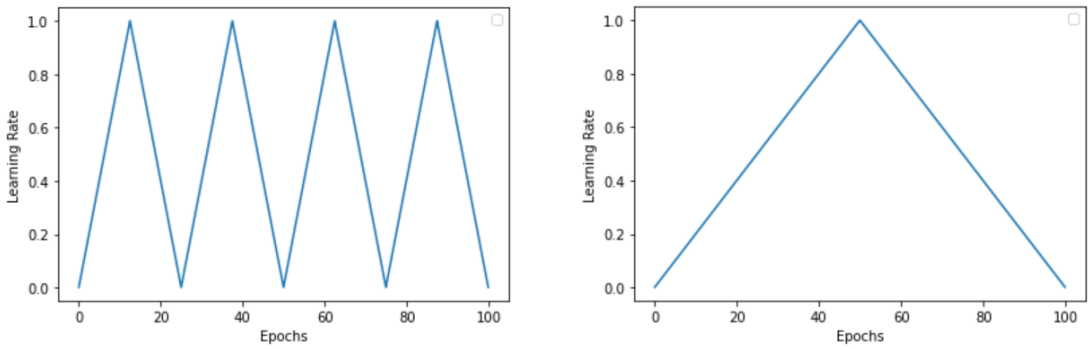
Figure 2.7.: Sketch of a diverging attempt at optimization due to a learning rate that was chosen too large.

each epoch. A decaying learning rate introduces step size annealing. This form of step size annealing differs from the one mentioned in Section 2.3. Instead of decaying the quotient of the momentum vectors it decays the learning rate directly. We can choose between a variety of decay settings such as linear decay [23], exponential decay [23], or piece-wise constant learning rate schedules [1].

The result of a decaying learning rate might be that learning rates, which were initially chosen too large, could still produce a converging descent. Furthermore, the optimizers can descent quickly towards the general direction of the optimum and then slowly find it without constantly overshooting. This process can be thought of as a two part optimization. At first, we optimize quickly but imprecisely. Towards the end we trade speed for precision. Our last steps become smaller and smaller and thus, more accurate. With small steps we then find a precise optimum.

Decaying the learning rate is not the only form of scheduling it. We might also choose to preform a cycle [4]. This CLR scheme might even produce incredible results for the convergence of the algorithms [1, 4]; the details of which are discussed in Section 2.5.

A CLR scheme usually starts with an initially small value α_{min} (minimum learning rate) and continuously increases the learning rate to a larger value α_{max} (maximum learning rate), after which it the decreases the learning rate back down to α_{min}. The update of the learning rate usually happens after each iteration. The ascent and descent of the learning rate parameter itself are usually linear [4]. Nonetheless, alternatives can be thought of and have already been implemented and tested successfully [24]. This cycle can be performed arbitrarily often during the training. If exactly one complete cycle is performed over the predefined number of epochs,



(a) Arbitrary CLR. (b) 1-cycle.

Figure 2.8.: CLR schemes with $\alpha_{min} = 0.001$ and $\alpha_{max} = 1$.

we call that a *1-cycle*. Fig. 2.8 shows two CLR schemes over 100 epochs – 2.8(a) a cycle that repeats itself four times and 2.8(b) a *1-cycle*. The learning rate in a *1-cycle* can be defined as

$$\alpha^{(j)} := \begin{cases} \alpha_{min} + \frac{2(j-1)(\alpha_{max}-\alpha_{min})}{n_{iter}}, & j \in \left\{1, ..., \lceil \frac{n_{iter}}{2} \rceil\right\} \\ \alpha_{max} + \frac{2\left(j-\frac{n_{iter}}{2}\right)(\alpha_{min}-\alpha_{max})}{n_{iter}}, & j \in \left\{\lceil \frac{n_{iter}}{2} \rceil + 1, ..., n_{iter}\right\}, \end{cases} \tag{2.21}$$

yielding the special points

$$\alpha^{(1)} = \alpha_{min} \tag{2.22}$$

$$\alpha^{\left(\frac{n_{iter}}{2}\right)} = \alpha_{max}, \text{ if } n_{iter} \text{ uneven} \tag{2.23}$$

$$\alpha^{(n_{iter})} = \alpha_{min}. \tag{2.24}$$

To conclude this subsection, we showed that the learning rate does not need to be a constant scalar. The value of the learning rate can be adjusted in each epoch or, more commonly, each iteration in training. The way in which the learning rate is scheduled can have a direct influence on the quality of the results [1, 4, 23, 24].

Momentum

We already discussed momentum in Section 2.3. Nonetheless, for the purpose of completeness, we review the momentum decay rate β as a hyper-parameter. Most descent algorithms have a single momentum vector, but algorithms such as Adam might have more. The momentum vector can be described as a moving average. The way we calculate this moving average can be defined by the momentum decay rate $\beta \in [0, 1)$, with $\beta = 0$ yielding a momentum-less descent.

Larger decay rates indicate a greater importance given to past gradients. Smaller decay rates discount the past gradients heavily and thus, lay a stronger emphasis on the current gradient. This importance or emphasis can be seen as a level of trust divided between the past and the present gradients.

2.5. Super-Convergence

The term super-convergence was coined by Smith and Topin [1]. It refers to an emergent phenomenon when using CLR with large maximum learning rates. Essentially, the training of a ResNet [2], Densenet [25], or LeNet [26] on the CIFAR-10, CIFAR-100, Imagenet and MNIST image classification data sets speeds up by orders of magnitude. Furthermore, the finished network generalizes better on data it has not seen before. Lastly, the boost in performance becomes more significant when training data is limited or the network architecture is shallower [1].

Speeding up the network's training is the main focus in this work. Super-convergence therefore provides an interesting insight related to the objective. When it comes to why this speed-up occurs, the reason remains an open topic for research and discussion. Smith and Topin [1] give two possible intuitions to why this may happen.

Their first intuitive explanation is related to the topology of the underlying loss function. Assuming a smooth loss function, we might expect a traversal to occure as follows:

- In the beginning, the loss function has steep descending slopes which yield large gradients.
- During the next phase of training we may experience flat saddles with small gradients.
- Finally, we want to find a sharp optimum within a flat semi-optimum. In other words, within that wide valley there may be another sub-valley with an even better optimal value for $f(\theta)$.

This would mean that in the beginning we would desire small learning rates to counteract the large gradients. Intuitively speaking we can think of a local optimizer seeking the right initial direction to step towards. Too large learning rates in the beginning might lead towards a misguided direction. Having figured out the initial directions, the algorithm traverses the wide saddle. Here we prefer larger learning rates in order to take larger steps through a long, mostly featureless plane. We call this the global optimizer. On the final part of the function we want to avoid overstepping and thus, choose a smaller learning rate to find the exact minimum. Here we return to a local optimizer.

This intuition should be viewed critically as it works well only when one assumes the smooth loss functions observed from ResNets [8], see Section 2.1. Nonetheless, super-convergence is not limited to ResNets but works well with LeNets, too [1]. Another issue with this explanation

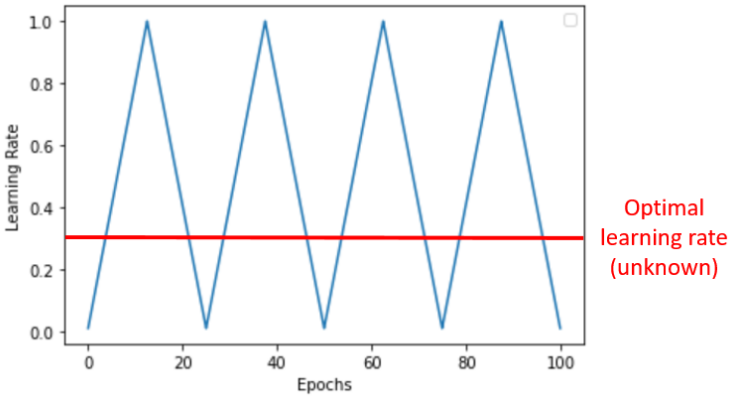


Figure 2.9.: Oscillating around the optimal constant learning rate with a CLR.

is that it favors sharp minima as solutions to the optimization problem. Since training with CLR also results in better generalization, this would be in contradiction to the findings of Keskar et al. [18]. Nonetheless, these findings are not undisputed. Dinh et al. [19] argue that sharp minima can generalize for deep networks. To conclude, this explanation is to be viewed critically but can nonetheless be used as an intuition for the success of CLR.

The second possible explanation given by Smith and Topin [1] is much simpler. Assuming there is an optimal constant learning rate, then we are unlikely to exactly choose it. If we choose a CLR with large enough ranges for the learning rate α, we experience phases of training with a smaller than optimal learning rate and phases with a larger than optimal learning rate, see Fig. 2.9. The assumption is that this yields better performance than always using a smaller or larger than optimal learning rate. The problem with this explanation are the underlying assumptions that (1) there is such an optimal constant learning rate and (2) under- and overestimating it yields better performance. Especially the existence of such an optimal learning rate is to be viewed critical. Super-convergence might just as well be a phenomenon emerging only from CLR.

These two explanations are nothing more than intuitions. They are both problematic in their own way. Nonetheless, together they might explain at least some aspects of why the phenomenon of super-convergence emerges in training. More importantly, the speed-up of training neural networks can be observed experimentally and can be exploited in practice. A positive side effect is the better generalization [1]. Therefore, using CLR is a favorable alternative to constant learning rates or decaying learning rates when training neural networks.

2.6. AdaSecant

So far we have viewed the learning rate α as a scalar, either constant or prescheduled. In this last section of the theoretical foundations we want to explore the possibility of calculating α in each iteration using curvature information. If successful, this method could replace the task of manually setting or scheduling a learning rate.

In addition to being calculated, the learning rate α will be vectorized, meaning instead of being a scalar for the gradient we will introduce a learning rate that scales each component g_i of the gradient g differently. An algorithm which does this is called AdaSecant. AdaSecant uses information about the gradient and the curvature of the Hessian in order to compute the learning rate [6]. In this section we will first explain the numerical method that forms the foundation for AdaSecant. In the next step we will introduce the specific adjustments made to that method.

The Underlying Numerical Method

So far, we have not discussed any descent algorithm that solves for the learning rate. We rather used a fixed or scheduled learning rate for each iteration. This means that there might be better, more efficient solutions for solving the optimization problem (i.e. finding a minimum of the loss function). On the other hand, the problem of finding a root of a function has been solved efficiently with methods such as Newton's Method or the Secant Method [27]. Unfortunately, in almost all cases the loss function will not have any real-valued roots. In fact most loss functions are strictly positive.

Nonetheless, we do know that the first derivative of the loss function has real-valued roots that correspond to the loss function's local minima, local maxima, and true saddle points. Using this information we can use Newton's Method in order to find a root of the loss function's first derivative. How we make sure that we converge towards a root corresponding to a local minimum, instead of one that corresponds to a local maximum, will be explained later in this section. Essentially we will develop a derivation of the Newton Method that follows the negative gradient within a deviation of no more than 90 degrees. But first of all, we must understand how directional Newton Methods work and how we can use them for the optimization problem at hand, which is finding a minimum of the loss f.

The Newton Method is an iterative optimizer for finding the root of a function $\phi(w)$. For this, the point where the tangent line, drawn at the parameter value of the j-th iteration $w^{(j)}$, evaluates to zero is taken. The assumption is that the tangent line is a good enough approximation for the function in the local region. In other words, we use the linear approximation of the function to guess a point closer to the root of the actual function. Fig. 2.10 visualizes the idea behind this underlying assumption. In order to compute this update step, the value $\phi(w^{(j)})$ of the function f with the parameters $w^{(j)}$ of the j-th iteration and the curvature information from

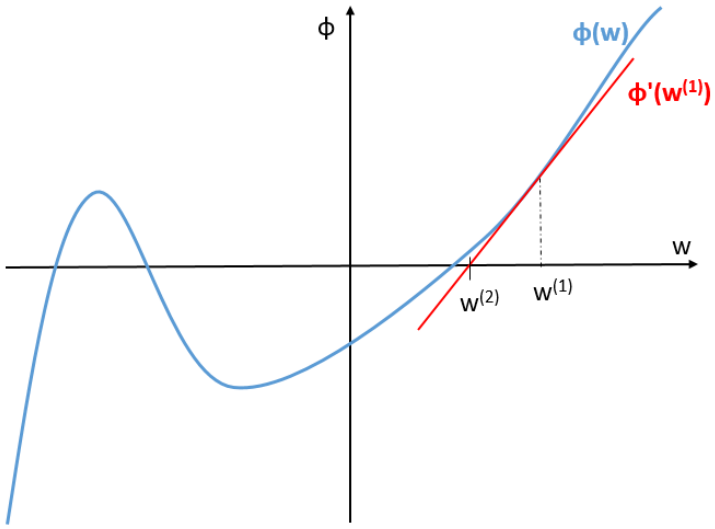


Figure 2.10.: The update step of Newton's Method visualized. Red line indicates the gradient at position $w^{(1)}$. The point where the resulting tangent line crosses the x-Axis is the new value $w^{(2)}$

the first derivative $\phi'(w^{(j)})$ are necessary. The algorithm finds a root by repeatedly calculating the update step [27]

$$w^{(j+1)} = w^{(j)} - \frac{\phi(w^{(j)})}{\phi'(w^{(j)})}. \tag{2.25}$$

In Equation 2.27 we transfer this idea to the second order, meaning that the gradient will be divided by the Hessian. In the n-dimensional setting this requires a matrix inversion [6]. The directional Newton Method, as proposed in [28], avoids this issue by introducing a directional vector $d^{(j)}$. Given the directional vector $d^{(j)}$, we obtain

$$w^{(j+1)} = w^{(j)} - \frac{\phi(w^{(j)})}{\nabla_w \phi(w^{(j)}) \cdot d^{(j)}} d^{(j)}. \tag{2.26}$$

Note that $\nabla_w \phi(w^{(j)}) \cdot d^{(j)}$ denotes the dot product of the gradient and the directional vector. Thus, the denominator evaluates to a scalar and no matrix inversion is necessary. If the directions $d^{(j)}$ are sufficiently close to the gradients $\nabla_w \phi(w^{(j)})$, the method provides quadratic convergence [28].

Because the loss function does not have a real-valued root, we use the second-order directional Newton Method with the Hessian H as the second derivative to find a root of the first derivative:

$$\theta^{(j+1)} = \theta^{(j)} - \frac{\nabla_\theta f(\theta^{(j)})}{H^{(j)} d^{(j)}} d^{(j)} \tag{2.27}$$

$$= \theta^{(j)} - \frac{g^{(j)}}{H^{(j)} d^{(j)}} d^{(j)} \tag{2.28}$$

$$= \theta^{(j)} - \frac{d^{(j)}}{H^{(j)} d^{(j)}} g^{(j)}. \tag{2.29}$$

This can be calculated separately for each parameter θ_i of the network

$$\theta_i^{(j+1)} = \theta_i^{(j)} - \frac{d_i^{(j)}}{h_i^{(j)} \cdot d^{(j)}} g_i^{(j)}, \tag{2.30}$$

with $h_i := \nabla_\theta \frac{\partial f(\theta^{(j)})}{\partial \theta_i}$ referring to the i-th row of the Hessian and $h_i^{(j)} \cdot d^{(j)}$ to the dot product.

The resulting update $\Delta_i^{(j)}$ for parameter $\theta_i^{(j)}$ is thus

$$\Delta_i^{(j)} = -\frac{d_i^{(j)}}{h_i^{(j)} \cdot d^{(j)}} g_i^{(j)}, \tag{2.31}$$

So effectively the learning rate is[2]

$$\alpha_i^{(j)} = \frac{d_i^{(j)}}{h_i^{(j)} \cdot d^{(j)}}. \tag{2.32}$$

Following the direction of the update $\Delta_i^{(j)}$ and then the numerical approximation of the Hessian along that direction $H\Delta \approx \nabla_{\theta_i} f(\theta^{(j)} + \Delta^{(j)}) - \nabla_{\theta_i} f(\theta^{(j)})$, we obtain

$$\alpha_i^{(j)} = \frac{\Delta_i^{(j)}}{h_i^{(j)} \cdot \Delta^{(j)}} \tag{2.33}$$

$$= \lim_{|\Delta^{(j)}| \to 0} \frac{\Delta_i^{(j)}}{\nabla_{\theta_i} f(\theta^{(j)} + \Delta^{(j)}) - \nabla_{\theta_i} f(\theta^{(j)})}. \tag{2.34}$$

[2] In the original paper, the term α refers to the approximation for $\nabla_{\theta_i} f(\theta) - \nabla_{\theta_i} f(\theta)$. Nonetheless, internal consistency is more important than external consistency. We therefore name this approximation a and let α be the learning rate (called η in the original paper).

Using finite differences and the update step of the last iteration, this term can be approximated as

$$\alpha_i^{(j)} \approx \frac{\Delta_i^{(j-1)}}{\nabla_{\theta_i} f(\theta^{(j-1)} + \Delta^{(j-1)}) - \nabla_{\theta_i} f(\theta^{(j-1)})} \tag{2.35}$$

$$= \frac{\Delta_i^{(j-1)}}{g_i^{(j)} - g_i^{(j-1)}} \tag{2.36}$$

$$\implies \theta_i^{(j+1)} = \theta_i^{(j)} - \frac{\Delta_i^{(j-1)}}{g_i^{(j)} - g_i^{(j-1)}} g_i^{(j)}. \tag{2.37}$$

We write the difference of the current and the past gradient as $a_i^{(j)} := g_i^{(j)} - g_i^{(j-1)}$.

What we are left with is a second-order directional Secant Method [6] which corresponds to the one-dimensional first-order variation [27]

$$\theta^{(j+1)} = \theta^{(j)} - \frac{\Delta^{(j-1)}}{f(\theta^{(j-1)} + \Delta^{(j-1)}) - f(\theta^{(j-1)})} f(\theta^{(j)}) \tag{2.38}$$

$$= \theta^{(j)} - \frac{f(\theta^{(j)})(\theta^{(j)} - \theta^{(j-1)})}{f(\theta^{(j)}) - f(\theta^{(j-1)})}. \tag{2.39}$$

Lastly, one more adjustment to Equation 2.37 must be made. The current update will find a root of the first derivative which either corresponds to a local minimum, a local maximum, or a saddle point. If we only update towards the general direction (with a deviation of less than 90 degrees) of the negative gradient, we eventually follow the Secant Method to a local minimum. This works because we either follow the update step exactly as calculated when iterating towards a local minimum or we traverse in the exact opposite direction. Hence, the learning rate should not convey any directional information which deviates more than 90 degrees from the gradient, which is achieved by taking the absolute values only. The final definition of the update step is

$$\theta_i^{(j+1)} = \theta_i^{(j)} - \underbrace{\left| \frac{\Delta_i^{(j-1)}}{a_i^{(j)}} \right|}_{=\alpha_i^{(j)}} g^{(j)}. \tag{2.40}$$

The result is a second-order directional Newton Method with a secant approximation of the Hessian, using the differences of subsequent gradients. The method iterates towards the negative gradients and thus, towards a local minimum. The gradient and curvature is evaluated separately for each parameter and thus, for each dimension of the loss function. Therefore, we have a calculated learning rate that scales each element of the gradient individually.

The Adjustments

While this more or less simple idea represents the underlying concepts of AdaSecant, the actual algorithm is much more sophisticated with elements added that target numerical stability. These differences are [6]

1. substituting current values for their expected value,
2. the introduction of a numerically stable term for the learning rate α,
3. and a variance-reduced gradient $\tilde{g}$.

All central variables var (gradients g and gradients squared g^2, update step Δ and squares Δ^2, curvature information a and squares a^2, product of curvature information and update step $a\Delta$) are substituted with moving averages which approximate the expectation of this variable. Other than most moving averages, the update is defined differently in that it uses a calculated weight $\tau^{(j)}$ for each iteration j. The moving average is calculated as

$$E[var_i]_j = \left(1 - \frac{1}{\tau_i^{(j)}}\right) E[var_i]_{j-1} + \frac{1}{\tau_i^{(j)}} var_i^{(j)} \tag{2.41}$$

$$\tau_i^{(j)} = \left(1 - \frac{E^2[\Delta_i]_{j-1}}{E[\Delta_i^2]_{j-1}}\right) \tau_i^{(j-1)} + 1, \tag{2.42}$$

with $\Delta^{(j-1)} = -\alpha^{(j-1)} g^{(j-1)}$ being the update step from the previous iteration and thus, $E[\Delta]_{j-1}$ the moving average of Δ over the past iterations. The same goes for any variable var. The expected values $E[var]_j$ is calculated as the moving average of a variable over j iterations, as given in Equation 2.41.

The learning rate α in AdaSecant is then substituted for a more stable term:

$$\alpha_i^{(j)} = \frac{\sqrt{E[\Delta_i^2]_{j-1}}}{\sqrt{E[a_i^2]_j}} - \frac{E[a_i \Delta_i]_{j-1}}{E[a_i^2]_j} \approx \left| \frac{\Delta_i^{(j-1)}}{a_i^{(j)}} \right|. \tag{2.43}$$

Lastly, a more robust version is developed by introducing the variance-reduced gradient $\tilde{g}$

$$\tilde{g}_i^{(j)} = \frac{g_i^{(j)} + \gamma_i^{(j)} E[g_i]_j}{1 + \gamma_i^{(j)}}, \tag{2.44}$$

$$\gamma_i^{(j)} = \frac{E[(g_i - g_i')(g_i - E[g_i]_j)]_j}{E[(g_i - E[g_i]_j)(g_i' - E[g_i]_j)]_j}, \tag{2.45}$$

with $g'^{(j)} := \frac{1}{|B'|} \sum_{k=1}^{|B'|} \nabla_\theta f(\theta^{(j)})$ being the gradient calculated with the current parameter settings $\theta^{(j)}$ on the next batch B'.

Algorithm 2.3 shows the full implementation of AdaSecant. The authors of the original paper [6] were unspecific about many details such as the initialization of their moving averages and other variables. Furthermore, their actual implementation[3] seems to diverge from their theoretical design [6]. Ambiguities which could not be resolved were left in, while those that were resolvable were adjusted for in the pseudocode. If the practical implementation solved an ambiguity, it was solved in the pseudocode accordingly. If the practical implementation contradicted or extended the theoretical proposal, the feature was left as in the theoretical proposal.

[3] https://github.com/sotelo/scribe

Algorithmus 2.3 AdaSecant

Require: The parameters of the network θ.
Require: The data set D.
Require: The objective function f.
Require: The network architecture $\mathcal{R}$.
procedure ADASECANT(θ, D, f, $\mathcal{R}$)
while stopping criterion is not met **do**
for all $B \subset D$ **do** // for batch B in data set D
$x \leftarrow$ GETVALUES(B)
$y \leftarrow$ GETLABELS(B)
$B' \leftarrow$ GETNEXTBATCHFROM(D)
$x' \leftarrow$ GETVALUES(B')
$y' \leftarrow$ GETLABELS(B')
$g \leftarrow 1/|B| \cdot \sum_{k=1}^{|B|} \nabla_\theta f(y_k, \mathcal{R}(x_k, \theta))$ // compute average gradient for samples in B
$g' \leftarrow 1/|B'| \cdot \sum_{k=1}^{|B'|} \nabla_\theta f(y'_k, \mathcal{R}(x'_k, \theta))$ // compute average gradient for next batch
compute $E[g]$ using 2.41
$g \leftarrow g/\,||E[g]||$ // normalize gradients
$g' \leftarrow g'/\,||E[g]||$ // normalize gradients
$a = g - g_{last}$ // approximate curvature of Hessian
for $i \in \{1, ..., n\}$ **do** // iterate over all parameters of the network
$\gamma_i \leftarrow \frac{E[(g_i - g'_i)(g_i - E[g_i])]}{E[(g_i - E[g_i])(g'_i - E[g_i])]}$ // compute variance correction term using 2.41
$\tilde{g_i} = \frac{g_i + \gamma_i E[g_i]}{1 + \gamma_i}$ // compute the corrected gradient
$cond1 \leftarrow |g_i - E[g_i]| > 2\sqrt{E[g_i^2] - E^2[g_i]}$
$cond2 \leftarrow |a_i - E[a_i]| > 2\sqrt{E[a_i^2] - E^2[a_i]}$
if $cond1$ or $cond2$ **then**
$\tau_i \leftarrow 2.2$ // reset the memory size for outliers
end if
update moving averages according to 2.41
$\alpha_i \leftarrow \frac{\sqrt{E[\Delta_i^2]}}{\sqrt{E[a_i^2]}} - \frac{E[\alpha_i \Delta_i]}{E[a_i^2]}$ // calculate learning rate
update τ according to 2.42
$\Delta_i \leftarrow -\alpha_i \tilde{g}_i$
$\theta_i \leftarrow \theta_i + \Delta_i$ // update parameters
$g_{last} \leftarrow g$
end for
end for
end while
end procedure

3. Implementations and Extensions of the Theoretical Foundations

In the previous chapter we laid out the theoretical basis for the upcoming experiments of Chapter 4. In this chapter we discuss the implementation of the algorithms in PyTorch[1]. PyTorch was chosen because it is well-suited for research as it makes developing and debugging realtively easy [29]. Another reason was the author's familiarity with it. Section 3.1 is a short summary of the standard implementations and libraries used. Sections 3.2 and 3.3 discuss the implementations that needed more coding and analysis. We discuss how Adam can be adapted for CLR in Section 3.2. Lastly, Section 3.3 is dedicated to the issues that were faced when implementing AdaSecant. Furthermore, the solutions and alternatives that were found to resolve those issues are presented. The final validation of all these approaches can be found in Sections 4.2–4.5. The training environments and other code we developed for the experiments and plots is provided in our GitHub repository[2].

3.1. Standard Implementations and Libraries

PyTorch is a useful deep learning framework for Python. It provides data sets, models, components, an automatic computation of the gradients, and many other useful features. PyTorch uses dynamically generated computational graphs to represent mathematical expressions. Due to its dynamic nature, it is well-suited for research [29].

PyTorch provides a number of neural networks that can be used for any classification problem, requiring only small adaptations. The torchvision package[3] includes the sub-package *models* which contains ResNet architectures. These can be loaded as either pretrained or untrained. Because the objective of this thesis is the optimal setting of the learning rate, which is a hyper-parameter for the training, we chose the untrained version. Nonetheless, the models from the torchvision package did not come close enough to the results that ResNets produce on the CIFAR-10 and CIFAR-100 data sets in other works [1, 2, 3]. We suspect the problem to lie in the first convolutional layer which discards a large portion of the pixels and therefore,

[1] https://github.com/pytorch/pytorch/tree/d5bfdd3dac33dfa84e2a511fa79c4ad4e0e6b822
[2] https://github.com/AverageDude0815/BA
[3] https://github.com/pytorch/vision/tree/5ef75d7bc666c9af3dd05b62b075df4c23ca467c/torchvision

the information available. This layer was probably designed for the ImageNet data set which contains images with a size of 224×224 pixels, compared to the 32×32 images from the CIFAR-10 or CIFAR-100 data sets. Thus, the CIFAR images are simply too small to discard any of the information available.

We found another package[4] that provides ResNets which are better suited for the CIFAR-10 and CIFAR-100 data sets and are in accordance with the designs used by He et al. [2]. The models chosen in this work were partially determined by the models provided from that package since using tested libraries leaves less room for errors. Therefore, different ResNet architectures than in the work of Smith and Topin [1], which is the paper we are validating with this thesis, were chosen. Nonetheless, the difference lies in the slight deviations in the number of layers and therefore, these architectures are comparable with regards to their general behaviour in training. We choose ResNet18 as the smaller network and ResNet50 as the larger network, compared to ResNet20 and ResNet56 in the original paper [1].

The torchvision package further provides the CIFAR-10 and CIFAR-100 data sets via torchvision.datasets.

For the implementation of standard optimization algorithms, which are SGD and Adam, we use the optim[5] package from the PyTorch library. The algorithms in that package are tested and fully functional. Therefore, an own implementation is unnecessary.

The scheduling of the learning rate can be done using the learning rate scheduler from the optim package.

Since PyTorch does not provide a complete training environment, the implementation of the training routine itself was done manually.

3.2. Combining Momentum and Cyclical Learning Rates

In this section we provide an insight into the interdependence between CLR and momentum. This analysis is extended in order to cover Adam. We provide two possible variations of Adam that should in theory produce cyclical update step sizes. This means that we do not only look at CLR but rather the update step term as a whole.

The momentum of momentum-based SGD is biased towards zero which means that smaller step sizes are expected in the early phases of training. Nonetheless, the influence of this bias decreases exponentially. Therefore, the interactions of CLR and momentum should be minimal. Momentum is rather a denoising functionality since it estimates the expected value

[4] https://github.com/kuangliu/pytorch-cifar/blob/49b7aa97b0c12fe0d4054e670403a16b6b834ddd/models/resnet.py

[5] https://github.com/pytorch/pytorch/tree/093a12aaa984bd4a7768bb306157067f7c95b0ec/torch/optim

of the gradients. No obvious reason exists for an undesired interfere with CLR. This is also reflected by the results of Smith and Topin [1] who were able to show that momentum-based descent methods such as Nesterov [14], AdaGrad [15], and AdaDelta [16] allow for super-convergence.

Nonetheless, Smith and Topin [1] were not able to show the same results for Adam [5]. Therefore, we decided to further examine this case. As a first step, we review the naive implementation of scheduling the learning rate without taking into account what happens to the rest of the variables used to compute the update step. Since the update step is approximately bounded by the learning rate, a CLR will affect the region of confidence for the step size, see Fig. 2.5(b) from Section 2.3. However, we do know that Adam performs a type of step size annealing, meaning that the update step size and the region of confidence decrease with the number of iterations, even with constant learning rates. This property, which is a core component to Adam's success [5], could therefore be problematic when CLR are introduced. Since *1-cycle* schedulers decay the learning rate towards later iterations, the additional step size annealing might cause the step sizes to become too small too quickly. Thus, different cycles should be evaluated.

For this naive implementation we had to adjust the Adam algorithm provided by the PyTorch optim package, which does not allow for learning rate scheduling. In order to achieve this, a new learning rate parameter was created for the Adam class[6]. This learning rate is then updated via an own scheduler that was specifically implemented for this task. The updated Adam class passes the learning rate as an argument to the standard Adam function[7]. We henceforth refer to this naive implementation as CLR-Adam or simply as Adam.

Following that line of thought, another approach for introducing a type of cyclical update routine emerges. Consider that Adam performs automatic step size annealing. This means that towards the end of training the effective step size decreases, even when the learning rate stays constant. This already resembles the concept of decaying learning rates as discussed in Sections 2.3 and 2.4. The missing link to creating a cyclical scheme can therefore be found in the beginning of training. We remember that Adam uses a bias-corrected first momentum vector $\hat{m}$. If the bias-correction term $(1 - \beta_1^j)$ is removed, the first momentum vector becomes biased towards zero. Effectively, we replace the first bias-corrected momentum vector with the normal first momentum vector; setting $\hat{m} = m$. This bias towards zero yields smaller update steps in the beginning of training. Thus, this new version of Adam performs a form of automatic cycling. We call this the Cyclical Update Step Adam or simply CUS-Adam.

CUS-Adam is implemented by adapting the Adam code provided by the PyTorch optim package. Both CLR-Adam and CUS-Adam are evaluated in Section 4.4.

[6]https://github.com/pytorch/pytorch/blob/46c886e8a6d3d9f502fa8c0985784436ab7f9543/torch/optim/adam.py
[7]https://github.com/pytorch/pytorch/blob/46c886e8a6d3d9f502fa8c0985784436ab7f9543/torch/optim/_functional.py#L54

3.3. Implementing AdaSecant

In this last section of the implementations and extensions, the implementation of AdaSecant is discussed. As mentioned in Section 2.6, the authors [6] were unspecific about many things, mostly about the initialization of their variables. Furthermore, the actual implementation[8] diverges from their theoretical design. For example, their implementation uses thresholds for the weights $\tau^{(j)}$ used for calculating the moving averages. These thresholds were not mentioned in the paper. Another issue is that the sequence of operations differs from the theoretical pseudocode. Therefore, we must make some decisions on how AdaSecant should be implemented. Furthermore, we present a simpler implementation to be tested in the upcoming experiments. We call this algorithm Simple AdaSecant.

AdaSecant

The implementation decisions for AdaSecant are laid out chronologically by the actual implementation used in this thesis:

1. The gradients are calculated using the autograd package[9] of PyTorch. A prestep is taken to calculate the current gradient on the current batch. With the execution of the step method the gradient calculated on the next batch with current parameters is obtained. Both gradients are passed to the optimizer. The step method triggers the iteration of AdaSecant.
2. Inside each iteration j the moving average for the gradient $E[g]_j$ is calculated first. This is coherent with both the pseudocode and the authors' implementation.
3. The position of the gradient normalization in the sequence is ambiguous. We observed no difference independent of where and if the normalization term was included. Hence, the normalization is left out of the final algorithm.
4. Calculate the term $a^{(j)} = g^{(j-1)} - g^{(j)}$ to approximate the curvature of the Hessian. The exact position of the approximation of the curvature in the code is not of importance because the gradients stay constant over a single iteration. The only requirement is that it is calculated before use.
5. Calculate the corrected gradient. The semantics of the order of statements regarding the corrected gradient is equal in both pseudocode and implementation, even though the syntax slightly differs.

[8] https://github.com/sotelo/scribe
[9] https://github.com/pytorch/pytorch/tree/93d2e5090f9823102debab3845117c8e8208995b/torch/autograd

6. Update *all* moving averages. The pseudocode is unspecific here. The authors state to "update moving averages according to Equation 23" [6] and Equation 23 is related to the moving average for the update step $\Delta^{(j)}$. Nonetheless, their implementation updates all moving averages. Furthermore, the pseudocode does not include the update of the other moving averages specifically. Thus, simply all moving averages are updated in this step. The difference to the pseudocode is that the update of the memory size for the weights $\tau^{(j)}$ is done after this step in our implementation. The authors' implementation is consistent with our's.

7. Update the weights $\tau^{(j)}$ for calculating moving averages. This implementation choice differs from the pseudocode only in that it is applied after calculating the learning rate. This has no effect on the outcome of the learning rate if all moving averages have been updated previously, which they have. Again, the syntax is slightly different but the semantics are equivalent. The sequence is furthermore in accordance with the authors' actual implementation.

8. Apply thresholds to weights $\tau^{(j)}$. This information was left out of the pseudocode. Nonetheless, we observed it to be essential for the algorithm's convergence. Without it the algorithm quickly returns some invalid values (e.g. infinity or nan). We copy the position in the sequence from the actual implementation. The lower threshold is set to 1.5 and the upper threshold to 10^7, just as the authors' implementation suggests.

9. Reset memory size for weights $\tau^{(j)}$. This was the most challenging part. Neither sequence nor necessary condition are the same when comparing pseudocode and implementation. We use the formula as presented in the pseudocode because the objective of changing it was not made clear in the implementation. The sequence is taken from the implementation because we consider it to be more senseful to reset the memory size after setting it. After this, no ambiguities are left.

10. Compute the learning rate.

11. Compute the update step.

12. Update the parameters.

13. Save old values for next iteration.

All variables and moving averages are initialized with zero except for the weights $\tau^{(j)}$ which are initialized with one. This is also in accordance with the implementation of the original authors. Note that the moving averages are not biased towards zero with this implementation strategy. If the weights $\tau^{(j)}$ are set to one, the first update of the moving averages will neglect any initialization bias. Not quite an initialization, but related, the update of moving averages that require the initial update step $\Delta^{(0)}$, should use the negative corrected gradient $-\tilde{g}$ instead. This corresponds to an update step with a learning rate of 1.0.

To summarize, the sequence, initializations, and thresholds applied follow the actual implementation of the authors of AdaSecant. The formulas are interpreted as presented in the paper and shown in Section 2.6 and Algorithm 2.3. The resulting implementation is henceforth simply referred to as AdaSecant since it is the only implementation of the algorithm available to us in PyTorch. The implementation is given with Algorithm 3.1.

Simple AdaSecant

Now, we want to present a different approach to implementing AdaSecant. Because so many more or less arbitrary decisions had to be made, the resulting AdaSecant implementation might not perform as desired. Thus, we propose a Simple AdaSecant that removes all the sophisticated mathematical decisions. It can therefore be used to act as a validation experiment for the underlying idea of AdaSecant. In Section 2.3 we developed a second-order directional Newton Method with a secant approximation of the Hessian which is the update step we use in Simple AdaSecant:

$$\theta_i^{(j+1)} = \theta_i^{(j)} - \underbrace{\left| \frac{\Delta_i^{(j-1)}}{a_i^{(j)}} \right|}_{=\alpha_i^{(j)}} g^{(j)}. \tag{3.1}$$

The advantage is that all decisions for implementation are clear from the start except for the initial values. We choose

$$\Delta^{(0)} = -g, \tag{3.2}$$

$$a^{(1)} = g. \tag{3.3}$$

Thereby, we effectively set the learning rate of the first iteration to one. We choose $\Delta^{(0)} = -g$ (instead of $\Delta^{(0)} = g$) because it resembles a preceding update step with a learning rate of one. Due to the absolute value function, this decision is only of matter for the code's readability. The approximation of the curvature, by the term $a^{(1)}$, is set to equal the gradient $g^{(1)}$ as it resembles a preceding gradient $g^{(0)}$ of zero. Divisions by zero are set to evaluate to zero. This is because they usually occur in settings where a gradient's component g_i evaluates to zero and thus, no update step would be performed. Therefore, initializing either the curvature $a^{(1)}$ or the previous update step $\Delta^{(0)}$ with zero should be avoided as it implies a learning rate of zero which implies an update step of zero. This loop is endless.

In practice we observed that this implementation is numerically unstable. Therefore, a threshold for each component of the learning rate $\alpha_i^{(j)}$ had to be added. Different thresholds are compared in Section 4.5. Interestingly, we observed that the algorithm only performs well when values $\alpha_i^{(j)}$ which surpass this threshold are set to zero instead of being set to the

Algorithmus 3.1 Our implementation of AdaSecant

Require: The parameters of the network θ.
Require: The data set D.
Require: The objective function f.
Require: The network architecture $\mathcal{R}$.
procedure ADASECANT(θ, D, f, $\mathcal{R}$)
$g_{last} = 0.0$ // initialize last gradient
$\tau = 1.0$ // initialize weights for updating moving averages
Initialize moving averages with zero.
while stopping criterion is not met **do**
for all $B \subset D$ **do** // for batch B in data set D
$x \leftarrow$ GETVALUES(B)
$y \leftarrow$ GETLABELS(B)
$B' \leftarrow$ GETNEXTBATCHFROM(D)
$x' \leftarrow$ GETVALUES(B')
$y' \leftarrow$ GETLABELS(B')
$g \leftarrow 1/|B| \cdot \sum_{k=1}^{|B|} \nabla_\theta f(y_k, \mathcal{R}(x_k, \theta))$ // compute average gradient for samples in B
$g' \leftarrow 1/|B'| \cdot \sum_{k=1}^{|B'|} \nabla_\theta f(y'_k, \mathcal{R}(x'_k, \theta))$ // compute average gradient for next batch
compute $E[g]$ using 2.41
$g \leftarrow g / ||E[g]||$ // normalize gradients
$g' \leftarrow g' / ||E[g]||$ // normalize gradients
$a = g - g'$ // approximate curvature of Hessian
for $i \in \{1, ..., n\}$ **do** // iterate over all parameters of the network
$\gamma_i \leftarrow \frac{E[(g_i - g'_i)(g_i - E[g_i])]}{E[(g_i - E[g_i])(g'_i - E[g_i])]}$ // compute variance correction term using 2.41
$\tilde{g}_i = \frac{g_i + \gamma_i E[g_i]}{1 + \gamma_i}$ // compute the corrected gradient
update all moving averages according to 2.41
// use $\Delta = -\tilde{g}$ where Δ is undefined
$cond1 \leftarrow |g_i - E[g_i]| > 2\sqrt{E[g_i^2] - E^2[g_i]}$
$cond2 \leftarrow |a_i - E[a_i]| > 2\sqrt{E[a_i^2] - E^2[a_i]}$
if $cond1$ or $cond2$ **then**
$\tau_i \leftarrow 2.2$ // reset the memory size for outliers
end if
$\alpha_i \leftarrow \frac{\sqrt{E[\Delta_i^2]}}{\sqrt{E[a_i^2]}} - \frac{E[\alpha_i \Delta_i]}{E[a_i^2]}$ // calculate learning rate
update τ according to 2.42
$\Delta_i \leftarrow -\alpha_i \tilde{g}_i$
$\theta_i \leftarrow \theta_i + \Delta_i$ // update parameters
$g_{last} \leftarrow g$
end for
end for
end while
end procedure

Algorithmus 3.2 Simple AdaSecant

Require: The parameters of the network θ.
Require: The data set D.
Require: The objective function f.
Require: The network architecture $\mathcal{R}$.
procedure SIMPLEADASECANT(θ, D, f, $\mathcal{R}$)
$g_{last} = 0.0$ // initialize last gradient
$\Delta = -g$
while stopping criterion is not met **do**
for all $B \subset D$ **do** // for batch B in data set D
$x \leftarrow$ GETVALUES(B)
$y \leftarrow$ GETLABELS(B)
$g \leftarrow 1/|B| \cdot \sum_{k=1}^{|B|} \nabla_\theta f(y_k, \mathcal{R}(x_k, \theta))$ // compute average gradient for samples in B
$a = g - g'$ // approximate curvature of Hessian
for $i \in \{1, ..., n\}$ **do** // iterate over all parameters of the network
$\alpha_i \leftarrow \left|\frac{\Delta_i}{a_i}\right|$ // calculate learning rate
$\Delta_i \leftarrow -\alpha_i g_i$
$\theta_i \leftarrow \theta_i + \Delta_i$ // update parameters
$g_{last} \leftarrow g$
end for
end for
end while
end procedure

threshold's value. This manipulates the direction in an undesired way. From a theoretical stance it is unclear why this choice increases the performance.

The implementation of Simple AdaSecant is given by Algorithm 3.2.

The results for Simple AdaSecant are evaluated in Section 4.5, which is also where we evaluate AdaSecant. Simple AdaSecant can validate the approach if AdaSecant does not perform successfully. Alternatively, Simple AdaSecant could validate the extensions made by AdaSecant (e.g. corrected gradient, moving averages) if AdaSecant's performance is superior to that of Simple AdaSecant.

4. Experiments

In this chapter the experiments with corresponding results are presented and discussed in the context of prior research. In Section 4.1 the experimental set-up is discussed and some intermediate results, which influenced the set-up, are presented preemptively. A new intuition as to why super-convergence emerges is introduced. Section 4.2 functions as an overview for all the experiments conducted in this thesis. In Sections 4.3, 4.4, and 4.5 selected results are presented and discussed within the context of prior research. SGD and fundamental research on CLR and super-convergence are contained within Section 4.3. We also evaluate intuitions to why super-convergence emerges. In Section 4.4 CLR-Adam and CUS-Adam are evaluated in terms of convergence behavior. For the first time it is shown that super-convergence can be applied to Adam as well. Lastly, in Section 4.5 we evaluate the implementations of (Simple) AdaSecant as introduced and discussed in Section 3.3. These results, especially the values for the calculated learning rates, are then compared to the findings on CLR and discussed within that context.

4.1. Set-Up of Experiments and Inter-Dependencies of Hyper-Parameters

In this section we discuss the experiments conducted in this thesis and the choices which had to be made regarding the selection of data sets, networks, optimizers, and hyper-parameter settings. The result of these choices is a tree of combinations, which are tested with regard to their efficiency in training. Because of the rapidly increasing complexity of experiments with multiple levels of independent variables, not all combinations can be evaluated within the scope of this thesis. Therefore, some intermediate results have to be analyzed preemptively in order to determine interesting cases that require further examination. These intermediate results, which influence the structure of the experiments, are presented in this section instead of Sections 4.3–4.5 which contains the major part of the results.

4.1.1. Choice of Data Sets

For training and validation of the networks the CIFAR-10 and CIFAR-100 data sets are selected. These standardized image classification data sets are commonly used in research [1, 2, 3, 4, 8].

The comparability of results is a major quality indicator for this work. Therefore, the CIFAR data sets are good choices for our experiments. The data sets are provided as 32×32 pixel images via the torchvision package[1]. Their relatively small size, which benefits computational efficiency, and the easy access, granted through PyTorch, further confirms the CIFAR data sets as the preferred choice for this thesis.

Both data sets consist of 50,000 images for training and another 10,000 images for testing. The fundamental difference is that the CIFAR-10 data set contains 10 distinct classes and the CIFAR-100 data set contains 100 distinct classes. This effectively means that when learning the classifier for CIFAR-10 the network has 5,000 samples for each class while for CIFAR-100 only 500 samples per class are provided. Smith and Topin [1] claim that the fewer samples per class are provided, the stronger the effects of super-convergence become. We want to verify their observation and thus, compare the effects of CLR on the convergence speed of neural networks for both, CIFAR-10 and CIFAR-100.

At this point it should be mentioned that the results in this thesis are not as good as the state of the art. It was not possible to achieve the same accuracy on the data sets as other researchers did [1, 2, 3, 4]. In these works the converged networks provided above 90% accuracy on the CIFAR-10 data set. Our networks achieved between 80% and 90% accuracy in the settings that provided good training results. Similarly, for the CIFAR-100 data set Smith and Topin [1] achieve up to 69% accuracy, we achieve a maximum of 63.72% accuracy. We suppose that the data sets need some normalization and padding to allow for better performance. Unfortunately, this realization came too late to re-run all the experiments. However, the focus of this work is on the convergence speed and quality compared between different learning rate settings. We are very confident that the results of this work can be transferred to state of the art data preparation, since we conducted earlier runs on the torchvision models[2]. These performed below 80% because they are not suited for smaller images such as contained by the CIFAR data sets. These results are comparable to our later runs presented in Section 4.2 with regards to convergence speed but not maximum accuracy. This similarity in behaviour indicates that the findings in this thesis are independent from the optimal convergence behaviour of the model on the data set. We therefore belief that future researchers can obtain the same convergence behavior when working with normalized and padded data. A selection of the results for the torchvision models is presented in Appendix A.2.

4.1.2. Choice of Networks

In Section 2.1 we already explained the reasons for choosing ResNets. In Section 3.1 we elaborated which networks were chosen for which reasons. To summarize, we choose ResNets because they are well-suited for deep learning [2], they yield a smooth loss function on the

[1] https://github.com/pytorch/vision/tree/5ef75d7bc666c9af3dd05b62b075df4c23ca467c/torchvision
[2] https://github.com/pytorch/vision/tree/5ef75d7bc666c9af3dd05b62b075df4c23ca467c/torchvision

CIFAR-10 data set [8], and they are used in the work of Smith and Topin [1], whose results we want to replicate and extend. We chose ResNet18 and ResNet50 because they were easily available to us and are close enough in number of layers to ResNet20 and ResNet56, which are some of the networks used by Smith and Topin [1].

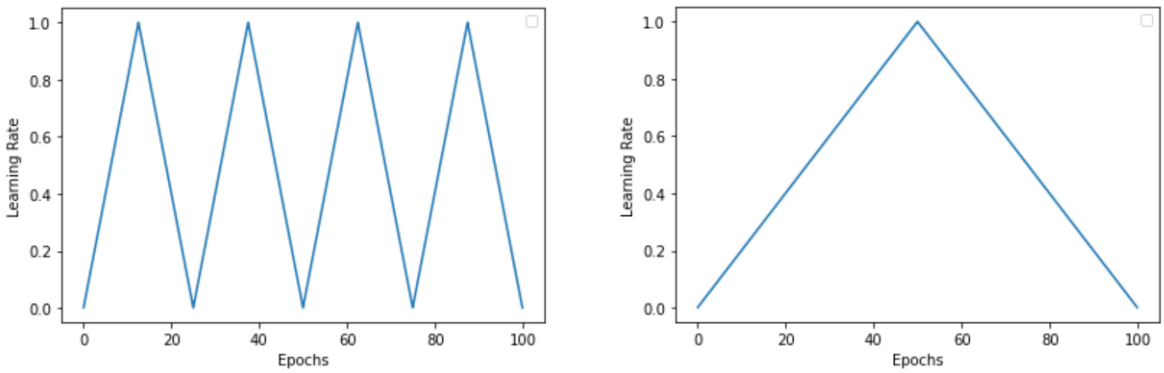
For CIFAR-10 we use both networks in order to compare the effects of CLR on different network depths. For CIFAR-100 we use ResNet50, the results of which are compared to the results of ResNet50 on CIFAR-10 in order to compare the effects of CLR on different number of samples per class available for training. ResNet18 on CIFAR-100 is a combination we did not observe to be tested in previously conducted research [1, 2, 3, 4, 6]. The high number of classes of the CIFAR-100 data set are probably too difficult to classify for the relatively shallow architecture of the ResNet18. Since this combination is not necessary for the comparisons in this thesis, we are not using it either.

4.1.3. Choice of Optimizers

We investigate the influence of learning rates on the training speed of neural networks for SGD, Adam, and AdaSecant. The latter calculates its learning rate rather than requiring it as an argument passed. SGD was chosen to verify the results of Smith and Topin [1] and provide more data on the behavior of super-convergence in general. Furthermore, we investigate the effects of CLR on Adam because Smith and Topin [1] were not able to achieve super-convergence-like behavior with Adam. Lastly, AdaSecant is chosen as an alternative to the CLR scheme and to setting the learning rate in general.

4.1.4. Epoch Length Settings

Firstly, we must decide how long we train the networks. This choice yields a trade-off between computational efficiency and quality of the trained networks. On the one hand, more epochs lead to better results because they give the networks time to fully converge. Furthermore, potential late training effects are accounted for. On the other hand, more epochs mean more time spent training a single network, that is, conducting a single experiment out of many. The longer we train for, the fewer experiments can be conducted. He et al. [2] needed 64,000 iterations for the full convergence of their networks. They used a batch-size of 128 which means their 64,000 iterations correspond to 164 epochs. Smith and Topin [1] show that the effects of super-convergence become noticeable after at most 100 epochs. We therefore choose to train for 100 epochs as well since we are primarily interested in increasing the speed of convergence rather than the effects during late training. Thus, we can conduct more experiments with more variations.



(a) 25-cycle.

(b) 100-cycle.

Figure 4.1.: CLR schemes with $\alpha_{min} = 0.001$ and $\alpha_{max} = 1$ displaying the naming conventions used in the experiments.

4.1.5. Learning Rate Schemes and Ranges

In this part of the thesis we use a different naming convention than the one provided by Smith and Topin [1] used in Sections 2.4 and 2.5. Our CLR schemes are called *100-cycle*, *25-cycle*, and *10-cycle* instead of *1-cycle*. The number refers to the number of epochs needed for the completion of a full cycle (minimum learning rate to maximum learning rate and back to minimum learning rate). Hence, an *N-cycle* is equivalent to a *1-cycle* as introduced in Section 2.4 when simulated for N epochs. We choose this rephrasing of the cycle term to better suit the experiments conducted. A cyclical scheme can thus be referred to for more than one full cycle. The *25-cycle* routine is simulated for four full cycles and the *10-cycle* routine is simulated for ten full cycles. Fig. 4.1 shows a *25-cycle* and a *100-cycle*. We use the same cyclical schemes for SGD and Adam, but with different learning rate ranges.

For SGD we test the learning rates {0.001, 0.01, 0.1, 1, 3}. For Adam we test the learning rates {0.0001, 0.001, 0.01, 0.1, 1}. For SGD the learning rate 0.0001 is left out and for Adam the learning rate 3.0 is left out, as well. This is because the optimizers were already not functioning well when we used the next higher or lower learning rate respectively. Further increasing or decresing the learning rate thus, requires unnecessary computational energy, a valuable resource when training many neural networks. On the constant learning rates an interesting phenomenon emerges. When scaling the learning rate by a factor of 10, SGD behaves similarly to Adam in terms of validation accuracy during training, as shown in Fig. 4.2. Therefore the learning rate ranges for the CLR schemes were chosen accordingly. For SGD we evaluate the lower bounds 0.01 and 0.001 and the upper bounds 1.0 and 3.0. These choices were inspired by Smith and Topin [1] who use a range of [0.01, 3]. Hence, for Adam we use the lower bounds 0.001 and 0.0001 and the upper bounds 0.1 and 0.3. Later we add the

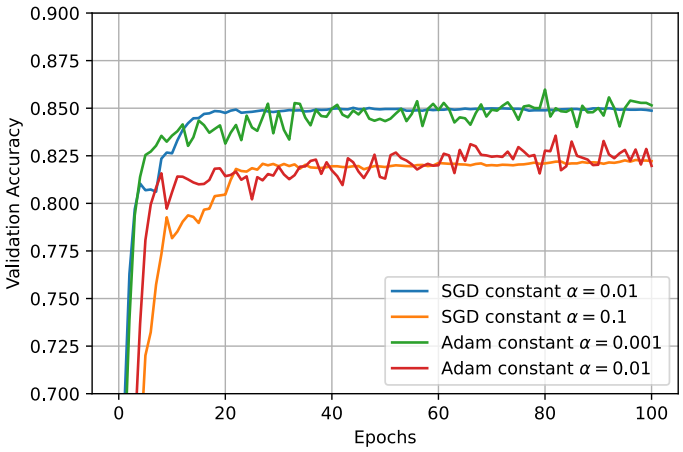


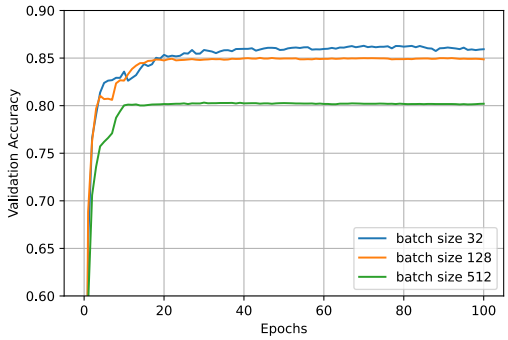
Figure 4.2.: When scaling the learning rate by a factor of 10, SGD behaves similar to Adam in terms of validation accuracy during training.

ranges [0.0001, 0.01] and [0.0001, 0.03] as we experimentally observe better convergence behaviours for Adam when using small upper bounds.

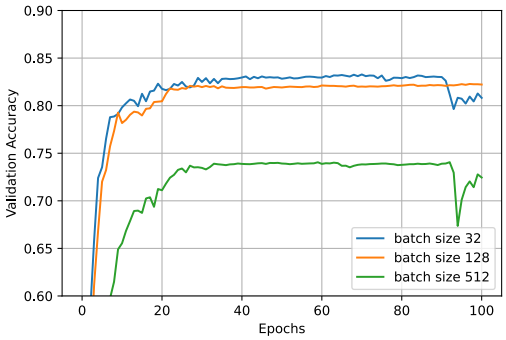
4.1.6. Choice of Batch Size

As discussed in Section 2.4, the selection of the batch size can influence the results in training. We therefore examine different settings and how they influence the results. In other words, we want to find a batch size that does not favor any schemes over others in order to control the final results. The batch sizes tested are 32, 128, and 512, which are all in the commonly used span of 32–512 [18]. Fig. 4.3 shows the influence of the batch size settings on the quality of the training of a ResNet18 on the CIFAR-10 data set for SGD with constant learning rates and CLR, as well as for Adam with a constant learning rate. For SGD the learning rates tested are 0.01 and 0.1 as recommend by Bengio [30]. For SGD with CLR we experimentally found the range of [0.001, 1] to yield good results. For Adam we used a learning rate of 0.001 as recommended by the original proposal [5]. We use two different cycle routines to analyse the effects of batch size. These routines are the aforementioned *100-cycle* and *25-cycle* schemes.

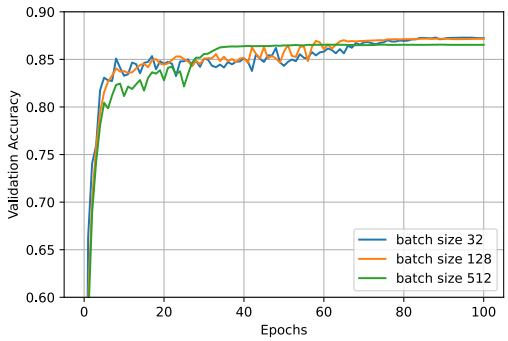
Figures 4.3(a) and 4.3(b) show that SGD with constant learning rates is sensitive to the choice of the batch size. Larger batch sizes tend to yield worse results. We observe batch sizes 32 and 128 to be similarly good measured by the overall convergence quality and speed. Figures 4.3(c), 4.3(d), and 4.3(e) show that CLR schemes and Adam are less sensitive to the choice of the batch size within the recommended range of 32–512 [18]. Fig. 4.3(d) indicates that a cycle of higher frequency destabilizes training when batch sizes are too small. We choose to run all further experiments with a batch size of 128 samples as this seems to be the one size



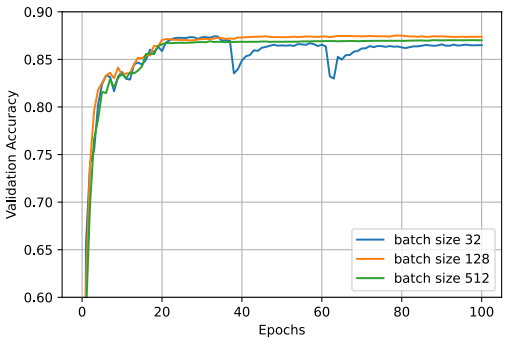
(a) SGD with a constant learning rate of 0.01.



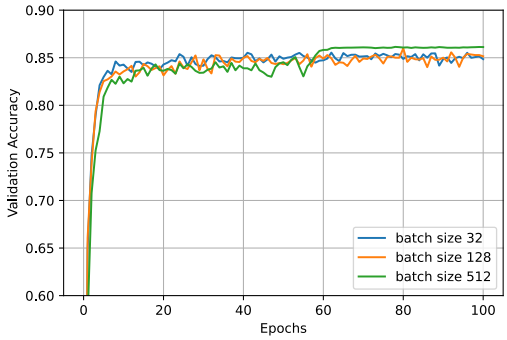
(b) SGD with a constant learning rate of 0.1.



(c) SGD with a learning rate cycle of 100 epochs with CLR of 0.001 to 1.0.



(d) SGD with a learning rate cycle of 25 epochs with CLR of 0.001 to 1.0.



(e) Adam with a constant learning rate of 0.001.

Figure 4.3.: Comparing the effect of different batch size settings on the convergence speed and quality of a ResNet18 on the CIFAR-10 data set for SGD and Adam with different learning rate schemes.

which does not favor any optimizer or learning rate scheme over the others. The effects of batch sizes on AdaSecant were not considered in this optimization because the algorithm was still not fully implemented at this point.

In Section 2.5 we already discussed two possible explanations for the phenomenon of super-convergence. One was the optimized traversal of saddle-like planes and sharp minima and the other one was the property of passing by the optimal learning rate within a cycle. The results observed in Fig. 4.3 and especially in Fig. 4.3(d) indicate another possibility. In Section 2.4 we thoroughly discussed the theory of sharp minima and their effect on generalization. Extending this idea, we propose another hypothesis to why super-convergence works. We call this intuition *Stochastic Escape*. Fig. 4.4 visualizes this idea. Consider parameter settings which would usually converge to a sharp minimum when using small learning rates. The region around a sharp minimum provides steep gradients to the optimizer. Furthermore, the spacial region around a sharp minimum is smaller than that area around a wide minimum. Larger learning rates multiplied with a large gradient are likely to leave that small region, as shown in Fig. 4.4(a). On the other hand, around a wide minimum the gradients are flatter and the distances required to leave that region are larger. It is therefore less likely to exit such a wide minimum region, even with large learning rates; as shown in Fig. 4.4(b). After many iterations it is stochastically less likely to converge to a sharp minimum than to a wide minimum. This would explain the higher accuracy observed by super-convergence and the lower sensitivity of CLR to batch size choices. Essentially, CLR destabilize the loss function just as small batch sizes do [22]. After finding an initial descent direction (i.e. smaller learning rates) and stochastically escaping sharp minima (i.e. larger learning rates), the CLR scheme transfers back to the local optimizer (i.e. smaller learning rates). This final stage of the CLR scheme finds the local minimum within the region surrounding it. The smaller learning rates towards the later stages of optimization are needed as to not overshoot the local minimum.

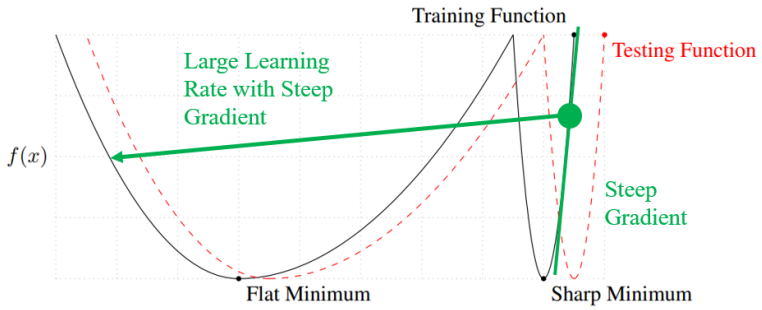
4.1.7. Decay Rate Settings

In this work we do not focus on learning rate decay. Nonetheless, we want to show that only CLR produce super-convergence instead of just the second part of the cycle, which can be seen as a decaying learning rate. Two experiments are therefore run with learning rate decay in order to control the results. We choose to decay the upper bounds chosen for SGD with CLR, which are 1.0 and 3.0. The decay rate γ is applied after each iteration. For the last iteration n_{iter} we desire a learning rate α_{min} of 0.001, which is equal to the lower bound for the most successful CLR schemes we chose. For an initial learning rate α_{max} of 1.0 we choose

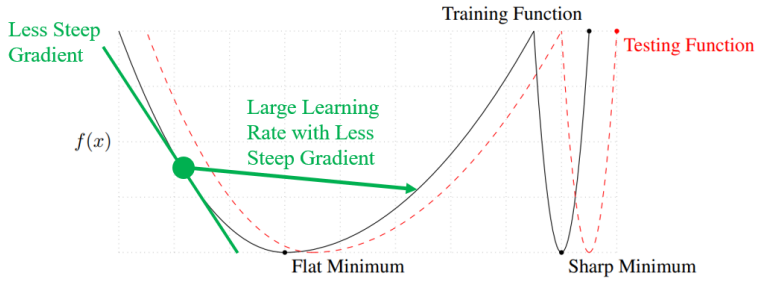
$$\gamma = \sqrt[n_{iter}]{\frac{\alpha_{min}}{\alpha_{max}}} = \sqrt[100\times\frac{50000}{128}]{0.001} \approx 0.9998231771 \tag{4.1}$$

and for an initial learning rate α_{max} of 3.0 we choose

$$\gamma = \sqrt[n_{iter}]{\frac{\alpha_{min}}{\alpha_{max}}} = \sqrt[100\times\frac{50000}{128}]{\frac{0.001}{3}} \approx 0.99979505799 \tag{4.2}$$



(a) Escaping a sharp minimum due to a steep gradient paired with a large learning rate and a resulting large update step.



(b) Staying within a wide minimum due to a flatter gradient paired with a large learning rate resulting in a intermediate update step.

Figure 4.4.: Visualizing the idea of *Stochastic Escape* of sharp minima.

as the decay rate. Thus, we evaluate the same ranges as we do for CLR schemes. As a similiar control trial, we test random learning rates within the range of 0.001 and 1.0 for SGD.

The final structure of experiments and a summary of the results can be found in Section 4.2 in Tables 4.1 4.2, 4.3, 4.4, and 4.5.

4.2. Overview of the Results

In this section we simply provide a summarized overview of the results from the experiments. Tables 4.1 4.2, 4.3, 4.4, and 4.5 contain these results, some of them are discussed in detail in their respectively corresponding sections. The first thing one might notice is the relatively poor performance of ResNet50 on the CIFAR-10 data set compared to that of ResNet18. We suppose the reason for this is overfitting. Data normalization and image padding should prevent that. Nonetheless, the results are still comparable for one network and one data set across the classifiers and learning rate schemes and settings. The best setting in terms of maximum validation accuracy within 100 epochs for each network and data set is marked in bold.

Data Set	Network	Optimizer	LR Scheme	LR	Batch Size	Max Acc. (%)	@epoch
CIFAR-10	ResNet18	SGD	constant	0.001	128	76.58	27
CIFAR-10	ResNet18	SGD	constant	0.01	32	86.29	70
CIFAR-10	ResNet18	SGD	constant	0.01	128	85.02	44
CIFAR-10	ResNet18	SGD	constant	0.01	512	80.32	30
CIFAR-10	ResNet18	SGD	constant	0.1	32	83.28	70
CIFAR-10	ResNet18	SGD	constant	0.1	128	82.28	97
CIFAR-10	ResNet18	SGD	constant	0.1	512	74.06	92
CIFAR-10	ResNet18	SGD	constant	1	128	74.1	76
CIFAR-10	ResNet18	SGD	constant	3	128	60.99	83
CIFAR-10	ResNet18	SGD	decay	1	128	81.95	53
CIFAR-10	ResNet18	SGD	decay	3	128	75.77	28
CIFAR-10	ResNet18	SGD	random	[0.001, 1]	128	70.98	96
CIFAR-10	ResNet18	SGD	100-cycle	[0.001, 1]	32	87.3	88
CIFAR-10	ResNet18	SGD	100-cycle	[0.001, 1]	128	87.18	86
CIFAR-10	ResNet18	SGD	100-cycle	[0.001, 1]	512	86.55	74
CIFAR-10	ResNet18	SGD	100-cycle	[0.001, 3]	128	86.72	76
CIFAR-10	ResNet18	SGD	100-cycle	[0.01, 1]	128	86.7	69
CIFAR-10	ResNet18	SGD	100-cycle	[0.01, 3]	128	86.31	85
CIFAR-10	ResNet18	SGD	25-cycle	[0.001, 1]	32	87.42	33
CIFAR-10	**ResNet18**	**SGD**	**25-cycle**	**[0.001, 1]**	**128**	**87.51**	**79**
CIFAR-10	ResNet18	SGD	25-cycle	[0.001, 1]	512	87.02	88
CIFAR-10	ResNet18	SGD	25-cycle	[0.001, 3]	128	87.28	42
CIFAR-10	ResNet18	SGD	25-cycle	[0.01, 1]	128	86.77	88
CIFAR-10	ResNet18	SGD	25-cycle	[0.01, 3]	128	87.13	38
CIFAR-10	ResNet18	SGD	10-cycle	[0.001, 1]	128	86.67	11
CIFAR-10	ResNet18	SGD	10-cycle	[0.001, 3]	128	86.57	23
CIFAR-10	ResNet18	SGD	10-cycle	[0.01, 1]	128	87.0	33
CIFAR-10	ResNet18	SGD	10-cycle	[0.01, 3]	128	87.02	83

Table 4.1.: Results for ResNet18 on CIFAR-10 using SGD as an optimizer. All trainings were run for 100 epochs. LR = learning rate, Max Acc. = maximal accuracy on the test set, @epoch refers to the epoch at which maximum accuracy is reached. The best setting for ResNet18 on CIFAR-10 is marked in bold.

Data Set	Network	Optimizer	LR Scheme	LR	Batch Size	Max Acc. (%)	@epoch
CIFAR-10	ResNet18	CLR-Adam	constant	0.0001	128	81.79	98
CIFAR-10	ResNet18	CLR-Adam	constant	0.001	32	85.54	62
CIFAR-10	ResNet18	CLR-Adam	constant	0.001	128	85.98	80
CIFAR-10	ResNet18	CLR-Adam	constant	0.001	512	86.14	78
CIFAR-10	ResNet18	CLR-Adam	constant	0.01	128	83.56	82
CIFAR-10	ResNet18	CLR-Adam	constant	0.1	128	81.07	89
CIFAR-10	ResNet18	CLR-Adam	constant	1	128	10.8	0
CIFAR-10	ResNet18	CLR-Adam	100-cycle	[0.0001, 0.01]	128	86.61	84
CIFAR-10	ResNet18	CLR-Adam	100-cycle	[0.0001, 0.03]	128	87.07	100
CIFAR-10	ResNet18	CLR-Adam	100-cycle	[0.0001, 0.1]	128	85.91	94
CIFAR-10	ResNet18	CLR-Adam	100-cycle	[0.0001, 0.3]	128	86.04	100
CIFAR-10	ResNet18	CLR-Adam	100-cycle	[0.001, 0.1]	128	86.22	87
CIFAR-10	ResNet18	CLR-Adam	100-cycle	[0.001, 0.3]	128	86.0	99
CIFAR-10	ResNet18	CLR-Adam	25-cycle	[0.001, 0.3]	128	87.28	27
CIFAR-10	ResNet18	CLR-Adam	10-cycle	[0.001, 0.3]	128	86.63	20
CIFAR-10	ResNet18	CUS-Adam	constant	0.001	128	85.56	65

Table 4.2.: Results for ResNet18 on CIFAR-10 using Adam as an optimizer. All trainings were run for 100 epochs. LR = learning rate, Max Acc. = maximal accuracy on the test set, @epoch refers to the epoch at which maximum accuracy is reached, CLR-Adam with constant learning rate is normal Adam.

Data Set	Network	Optimizer	LR Scheme	LR	Batch Size	Max Acc. (%)	@epoch
CIFAR-10	ResNet18	AdaSecant	calculated	None	128	60.09	2
CIFAR-10	ResNet18	Simple AdaSecant	calculated	3	128	81.33	62
CIFAR-10	ResNet18	Simple AdaSecant	calculated	10	128	82.86	36
CIFAR-10	ResNet18	Simple AdaSecant	calculated	$< \infty$	128	10.03	0

Table 4.3.: Results for ResNet18 on CIFAR-10 using (Simple) AdaSecant as an optimizer. All trainings were run for 100 epochs. LR = learning rate, Max Acc. = maximal accuracy on the test set, @epoch refers to the epoch at which maximum accuracy is reached, for Simple AdaSecant LR refers to the learning rate threshold.

Data Set	Network	Optimizer	LR Scheme	LR	Batch Size	Max Acc. (%)	@epoch
CIFAR-10	ResNet50	SGD	constant	0.01	128	83.7	97
CIFAR-10	ResNet50	SGD	25-cycle	[0.001, 1]	128	85.83	69
CIFAR-10	ResNet50	CLR-Adam	constant	0.001	128	86.28	68
CIFAR-10	**ResNet50**	**CLR-Adam**	**25-cycle**	**[0.001, 0.3]**	**128**	**88.21**	**26**
CIFAR-10	ResNet50	CUS-Adam	constant	0.001	128	86.89	72
CIFAR-10	ResNet50	Simple AdaSecant	calculated	10	128	78.51	71

Table 4.4.: Results for ResNet50 on CIFAR-10. All trainings were run for 100 epochs. LR = learning rate, Max Acc. = maximal accuracy on the test set, @epoch refers to the epoch at which maximum accuracy is reached, CLR-Adam with constant learning rate is normal Adam, for Simple AdaSecant LR refers to the learning rate threshold. The best setting for ResNet50 on CIFAR-10 is marked in bold.

Data Set	Network	Optimizer	LR Scheme	LR	Batch Size	Max Acc. (%)	@epoch
CIFAR-100	ResNet50	SGD	constant	0.01	128	54.43	18
CIFAR-100	ResNet50	SGD	25-cycle	[0.001, 1]	128	63.27	39
CIFAR-100	ResNet50	CLR-Adam	constant	0.001	128	58.56	93
CIFAR-100	ResNet50	CLR-Adam	100-cycle[a]	[0.001, 0.3]	128	61.21	98
CIFAR-100	**ResNet50**	**CLR-Adam**	**25-cycle**	**[0.001, 0.3]**	**128**	**63.72**	**27**
CIFAR-100	ResNet50	CUS-Adam	constant	0.001	128	59.32	97
CIFAR-100	ResNet50	Simple AdaSecant	calculated	10	128	50.32	94

Table 4.5.: Results for ResNet50 on CIFAR-100. All trainings were run for 100 epochs. LR = learning rate, Max Acc. = maximal accuracy on the test set, @epoch refers to the epoch at which maximum accuracy is reached, CLR-Adam with constant learning rate is normal Adam, for Simple AdaSecant LR refers to the learning rate threshold. The best setting for ResNet50 on CIFAR-100 is marked in bold.

[a] This does not fit the pattern used for ResNet50 on CIFAR-10 because it was an accidental run caused by wrongfully setting the cycle length parameter.

4.3. SGD and General Insights into Super-Convergence

In this section the effects of CLR on SGD are evaluated. Firstly, a comparison between the constant scheme and different CLR schemes is made. Secondly, different learning rates and learning rate ranges are compared for each scheme in order to determine the best settings and show the schemes' sensitivity to these settings. We then compare CLR schemes to other learning rate schedules which cover the same learning rate range in a different manner, namely exponential decay and randomly alternating learning rates for each iteration. Lastly, the effects of super-convergence are analyzed for deeper architectures and data sets with less samples per class. For this we use the ResNet50 architecture and the CIFAR-100 data set respectively. This comprehensive analysis provides some fundamental insights into the workings of super-convergence in general.

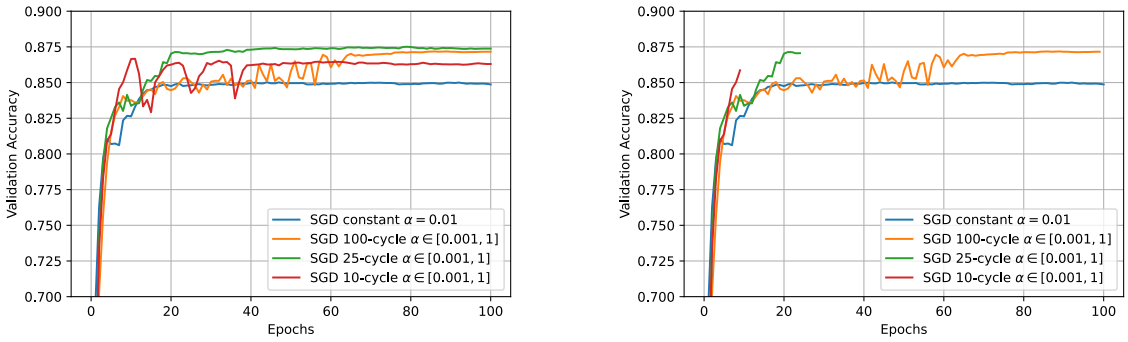
4.3.1. CLR Schemes Converge Faster

In Fig. 4.5 we present the results of different learning rate schemes for SGD. Fig. 4.5(a) shows the full training with potentially multiple cycle completions and Fig. 4.5(b) shows what would happen if we reduced the training length to the *1-cycle* scheme as introduced by Smith and Topin [1]. For the constant scheme a learning rate of 0.01 was used. For the CLR schemes a learning rate range of [0.001, 1] was used. These were the best learning rate settings we found for each scheme thus, giving each scheme its maximum potential for this comparison.

We observe the *25-cycle* scheme to be the best setting in terms of overall accuracy of the trained network. With a maximum validation accuracy of 87.51% in epoch 79 it is even the best setting we found for CIFAR-10 with ResNet18 (see Table 4.1). Nonetheless, the *10-cycle* scheme converges even faster in the earlier stages of training. By the end of epoch 11 the network achieves 86.67% accuracy (see Table 4.1). After that the increasing learning rates lead to a temporal escape from the optimum. The *100-cycle* scheme has no noteworthy advantage over the shorter cycles. However, it also outperforms the constant scheme.

These results are in accordance with the results of Smith and Topin [1] and the prior results of Smith [4]. We further observe that reducing the cycle to a *1-cycle* might not always be a good choice. For the *10-cycle* scheme the eleventh epoch is particularly important and should not be cut off in this trial. The *25-cycle* scheme also shows some, if minor, improvements over further iterations. Nonetheless, we observe the CLR schemes to not yield any major improvements after the completion of a full cycle and a few epochs. The completion of many complete cycles is not helpful to achieve higher validation accuracy. For efficient training, we recommend a full cycle plus a few extra epochs, to ensure convergence.

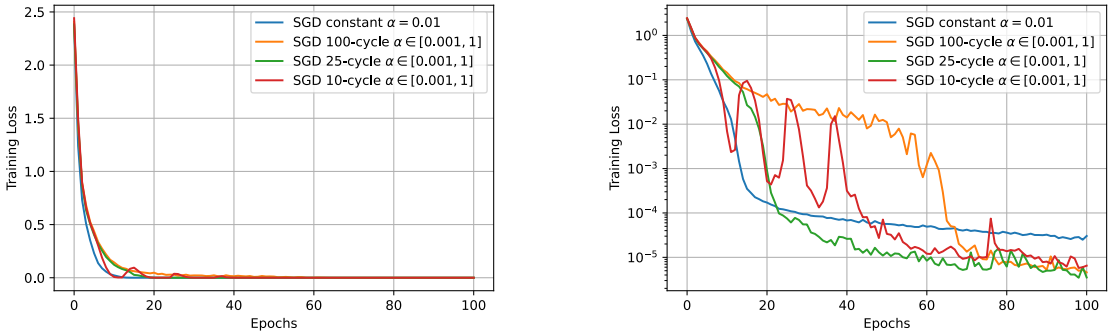
Analyzing the cross-entropy error of these schemes provides more insights. Fig. 4.6 shows the loss over time, full scale in Fig. 4.6(a) and logarithmically scaled in Fig. 4.6(b). SGD with constant learning rates descents the loss function continuously. The cyclical schemes



(a) Full plot over 100 epochs.

(b) Cut off after one full cycle (*1-cycle*).

Figure 4.5.: Validation accuracy of ResNet18 on CIFAR-10 trained with SGD with best constant learning rate (0.01) and different CLR routines with best range of [0.001, 1].



(a) Linear scale.

(b) Log-scale.

Figure 4.6.: Training cross-entropy loss of ResNet18 on CIFAR-10 trained with SGD with best constant learning rate (0.01) and different CLR routines with best range of [0.001, 1].

experience more sections where the loss increases after an epoch. Nonetheless, the overall convergence of the cyclical schemes is better than for the constant scheme.

In Section 4.1 we discussed the concept of *Stochastic Escape*. In short, CLR escape sharp minima and are therefore successful in achieving high accuracy. Fig. 4.6(b) shows that CLR find smaller minima and not just wider minima. Nonetheless, this is not a contradiction. We attribute the encounter of smaller minima to the third phase of the cycle, which is the local optimizer with small learning rates. This is because the results show that CLR minimize the loss function most effectively during the phases of training corresponding to small learning rates. In Section 4.1 we stated that small learning rates are necessary in order to find the minimum after *Stochastic*

Escape. Furthermore, if we take into account that the CLR schemes surpass the validation accuracy of the constant scheme during the phases of larger learning rates, the argument for *Stochastic Escape* is even strengthened. Further consider that during that phase of training the loss function is minimized less severely than during the phases corresponding to smaller learning rates. Fig. 4.5(b) shows that for the *25-cycle* and *100-cycle* schemes the validation accuracy surpasses the validation accuracy of the constant scheme roughly around the middle part of the circle (12.5 epochs and 50 epochs respectively). This is of course the part of the cycle which uses large learning rates and would therefore be responsible for *Stochastic Escape* and thus, better generalization (i.e. validation accuracy).

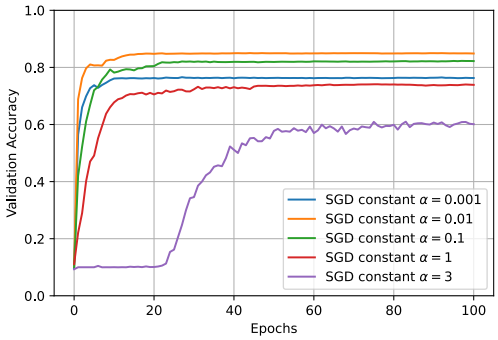
4.3.2. Low Sensitivity to Learning Rate Range

Fig. 4.7 shows different learning rate settings for SGD with constant learning rates and CLR with 10, 25, and 100 epochs per cycle. From Fig. 4.7(a) we learn that SGD with constant learning rates is very sensitive to the choice of learning rate. The learning rates 0.01 and 0.1 recommended by Bengio [30] perform best. Therefore, we confirm that smaller or larger learning rates are not recommendable. We use a learning rate of 0.01 for further experiments with SGD and constant learning rates as it is the best out of the five settings tested. Note that this plot uses a different scale than the ones following it in order to accommodate for the vast differences in convergence behavior.

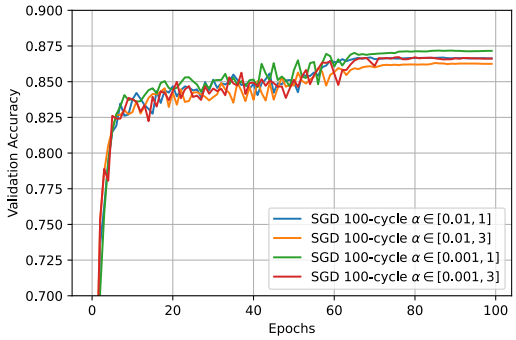
Figures 4.7(b), 4.7(c), and 4.7(d) show that CLR schemes are less sensitive to the exact choice of the learning rate range, at least within the lower bounds of 0.001 and 0.01 and the upper bounds of 1.0 and 3.0. More ranges should be tested in the future to investigate where this insensitivity stops. Earlier tests on the PyTorch models indicated that an upper bound of 10.0 is too large. Nonetheless, we did not find time to verify this on the current models. A learning rate range of [0.001, 1] performs slightly better than others in all schemes except for the *10-cycle* routine. We thus proceed to use this range for further experiments with SGD and CLR, instead of the range of [0.01, 3] mainly used by Smith and Topin [1].

4.3.3. Super-Convergence Emerges from Cyclical Nature of CLR

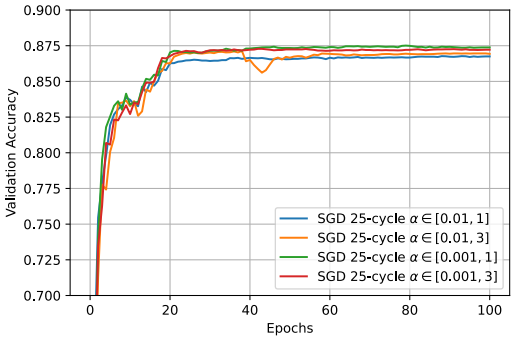
In order to get a better understanding of what causes super-convergence, we examine the training procedure for random learning rates, meaning that for each iteration a new learning rate is randomly generated within the range of 0.001 and 1.0. This should reveal if simply varying the learning rate over time can cause super-convergence-like behavior. Fig. 4.8(a) shows the learning rates used for this random learning rate scheme and Fig. 4.8(b) shows the resulting validation accuracy of a ResNet18 on the CIFAR-10 validation set. The performance of this scheme is visibly worse than those of any of the cyclical schemes. Furthermore, we observe similarities to SGD with a very large constant learning rate of 3.0 in Fig. 4.7(a). The



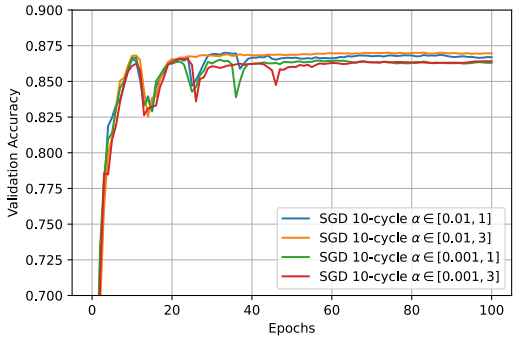
(a) SGD with constant learning rates between 0.001 and 3.0 (full scale).



(b) SGD with *100-cycle* and learning rate ranges with lower bounds of 0.001 and 0.01 and upper bounds of 1.0 and 3.0 (partial scale).

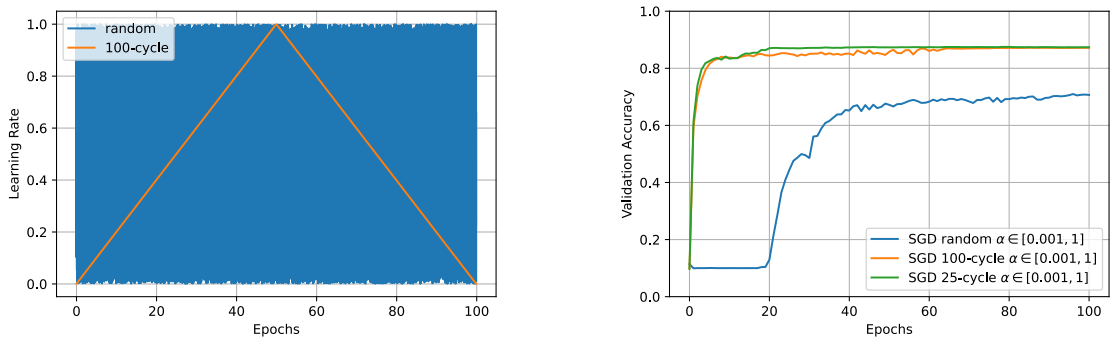


(c) SGD with *25-cycle* and learning rate ranges with lower bounds of 0.001 and 0.01 and upper bounds of 1.0 and 3.0 (partial scale).



(d) SGD with *10-cycle* and learning rate ranges with lower bounds of 0.001 and 0.01 and upper bounds of 1.0 and 3.0 (partial scale).

Figure 4.7.: Effect of different learning rates and learning rate ranges on different learning rate schemes with SGD. CLR schemes are less sensitive to the choice of learning rate range than the constant scheme is to the choice of learning rate. Note the different scaling for the constant scheme to accommodate for the full view of all settings.



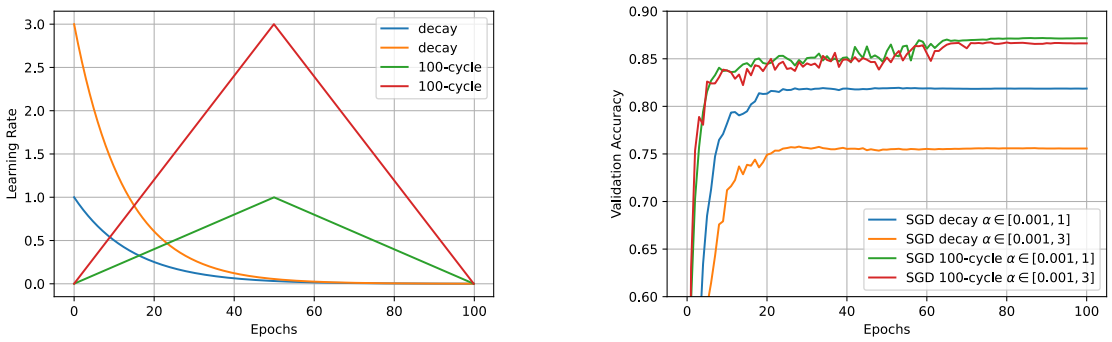
(a) Learning rate history.

(b) Validation accuracy.

Figure 4.8.: Comparing cyclical schemes to randomly setting the learning rate for each iteration with ResNet18 on CIFAR-10.

most prevalent feature in both is the early phase of training. Both schemes do not improve the network's performance for around 20 epochs. Only after that initial phase the validation accuracy increases. This indicates that the early phase of training requires small learning rates in order to find the correct direction for descent. This is in accordance with the explanation of Smith and Topin [1] for why super-convergence emerges, which we already discussed in Section 2.5. This explanation states that initially small learning rates find the right direction of descent, large learning rates speed up the training during the middle part, and finally small learning rates optimize without overshooting an ideal minimum. Another explanation we discussed was that an optimal constant learning rate might exist and with CLR we oscillate around it. This explanation seems less likely, or only partially responsible for super-convergence, as the results for random learning rates contradict that assumption. If it was the only reason, random schemes would yield similarly good results as cyclical schemes.

Fig. 4.9 shows the results for using exponential decay with large initial learning rates of 1.0 and 3.0 and decay rates of 0.999823 and 0.999795 respectively, the details of which were already discussed in Section 4.1. We choose a *100-cycle* routine with the same maximum learning rates used by the decaying schemes as a means of comparison because we set the decaying schemes to complete after 100 epochs. Therefore, this comparison seems least biased to us. During the first couple of epochs, when learning rates are still large (see Fig. 4.9(a)), exponential decay with large learning rates shows no signs of super-convergence when compared to a CLR scheme (see 4.9(b)). Over the entire 100 epochs the tested decay settings do not provide results which are comparable to those of the cyclical schemes, in terms of quality and speed. This indicates that the small learning rates during the early phases of training are needed for super-convergence. This is probably due to the fact that the optimizer has to find a correct direction of descent, which is done best when the learning rate is small and thus, the confidence in a single update step is also small. Only after finding that initial direction should learning



(a) Learning rate history.

(b) Validation accuracy.

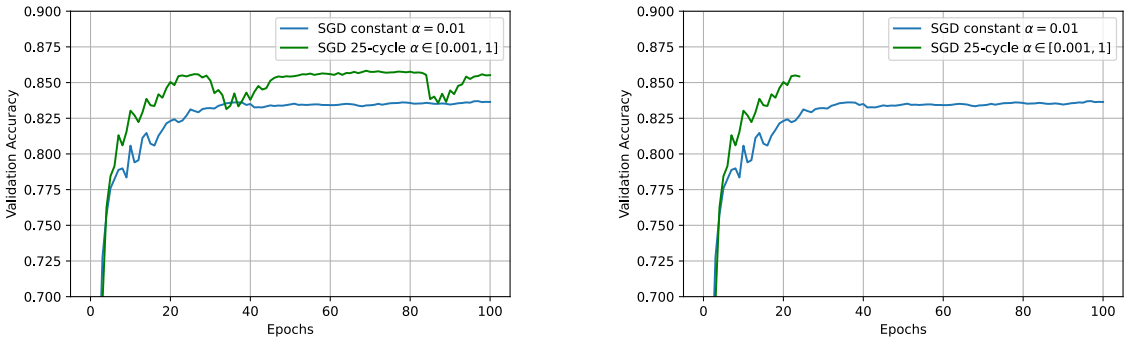
Figure 4.9.: Comparing *100-cycles* to exponential learning rate decay with very large initial learning rates for ResNet18 on CIFAR-10 using SGD.

rates be large in order to rapidly minimize the loss function. To visualize this, consider the topology of the loss function of a ResNet on the CIFAR-10 data set. In Section 2.1, Fig. 2.3(b) we presented a visualization of this loss function. Consider an optimization that, instead of moving towards the global minimum found in the center, moves towards a bad local minimum on the outer edges. This descent cannot converge optimally anymore.

Thus, it can be concluded that the full cycle of a CLR scheme is responsible for super-convergence. The emergence of the phenomenon requires the cyclical nature of the scheduler and not just random learning rates or decaying learning rates within that range. Of course, the number of experiments conducted with decaying learning rates is limited and more settings (e.g. cosine annealing used by He et al. [2]) should be tested for a final conclusion on this topic. Nonetheless, we have presented evidence that the cyclical nature of CLR schemes is responsible for super-convergence.

4.3.4. Super-Convergence is Independent of ResNet Depth

Since SGD works best with a *25-cycle* scheme with a learning rate range of [0.001, 1], we use this scheme to analyze the effects of super-convergence on deeper architectures, namely ResNet50. Therefore, we compare it to the performance of a ResNet50 on the CIFAR-10 data set using SGD with a constant learning rate of 0.01, which yields the best results for any constant scheme. Fig. 4.10 shows the results. For ResNet18 we observed an increase of 2.49 percentage points of validation accuracy when using a *25-cycle* routine instead of a constant scheme (see Table 4.1). For ResNet50 we observe an increase of 2.13 percentage points of validation accuracy when using a *25-cycle* routine instead of a constant scheme, which is a similar result (see Table 4.4). The speed up of convergence is also very similar and lastly,



(a) Full plot over 100 epochs.

(b) Cut off after one full cycle (*1-cycle*).

Figure 4.10.: Validation accuracy of ResNet50 on CIFAR-10 trained with SGD with best constant learning rate (0.01) and best cyclical routine (*25-cycle* with learning rate range of [0.001, 1]).

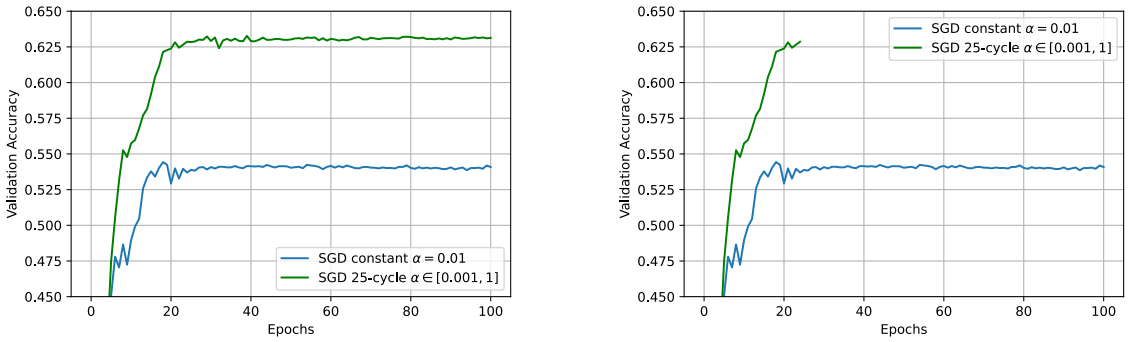
the performance peaks of the cyclical scheme at 69 epochs for ResNet50 and 79 epochs for ResNet18 are also not too far apart. Smith and Topin [1] observe that super-convergence works better with shallower network architectures. Our results do not confirm this observation. They rather indicate similar behavior independent of the network's depth and thus, a better transferability of the concept of super-convergence to other architectures.

4.3.5. Effects of Super-Convergence Increase when Training Data is Limited

Fig. 4.14 shows the effects of CLR on the behavior of SGD when training a ResNet50 on the CIFAR-100 data set. The validation accuracy increase of 8.43 percentage points between the constant scheme and the cyclical scheme is notably large (see Table 4.5). The effects of super-convergence are much more prevalent when training data is limited. The fewer samples per class are contained in the training data set, the higher the advantage on training speed and performance gained by CLR. These results are in accordance with the findings of Smith and Topin [1].

4.3.6. Summary

To conclude this section, we have verified the concept of super-convergence for SGD, as observed in prior works [1, 4]. In this thesis more learning rate ranges were tested than in these works and it was shown that super-convergence shows little sensitivity to the exact learning rate range used. Some intuitions to the reason for this phenomenon were examined, namely explanations regarding the optimality of the learning rates, explanations regarding



(a) Full plot over 100 epochs.

(b) Cut off after one full cycle (*1-cycle*).

Figure 4.11.: Validation accuracy of ResNet50 on CIFAR-100 trained with SGD with best constant learning rate (0.01) and best cyclical routine (*25-cycle* with learning rate range of [0.001, 1]). Note that for CIFAR-100 we use the same scale but display a different range for the validation accuracy than for the CIFAR-10 plots.

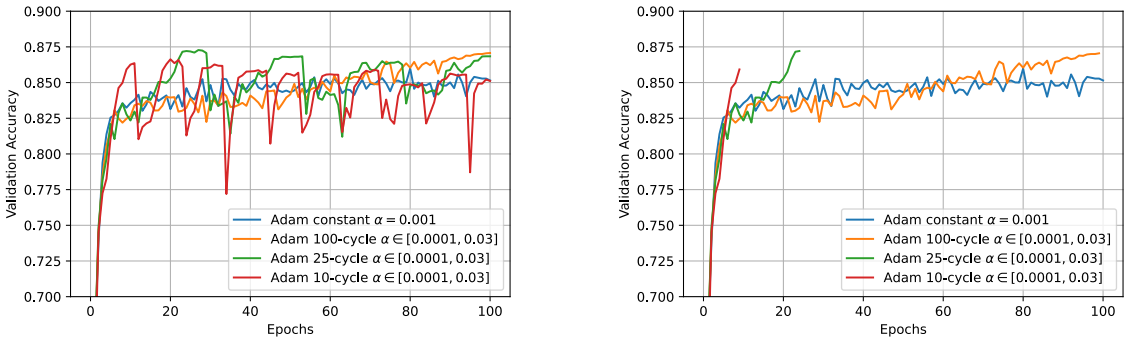
the topology, and the concept of *Stochastic Escape*. We have provided evidence that super-convergence emerges from the cyclical nature of CLR, rather than from the learning rate range covered. We have shown that super-convergence is independent of ResNet depth but decreases in strength with the number of training samples per class available. The results we obtained on ResNet depth were contradictory to those of Smith and Topin [1] and the results obtained on training data availability were in accordance with theirs.

4.4. Super-Convergence for Adam

In this section the effects of CLR on Adam are evaluated. Firstly, a comparison between the constant scheme and different CLR schemes is made. Secondly, different learning rates and learning rate ranges are compared for classical Adam and CLR-Adam using the *100-cycle* scheme. This is done in order to determine the best settings and show the schemes' sensitivity to these settings. Lastly, the effects of super-convergence are analyzed for deeper architectures and data sets with less samples per class. For this we use the ResNet50 architecture and the CIFAR-100 data set respectively.

4.4.1. CLR Schemes Converge Faster

Fig. 4.12 shows the results generated traing a ResNet18 on the CIFAR-10 data set with Adam and CLR-Adam with different cycle length. With a validation accuracy increase of 1.3



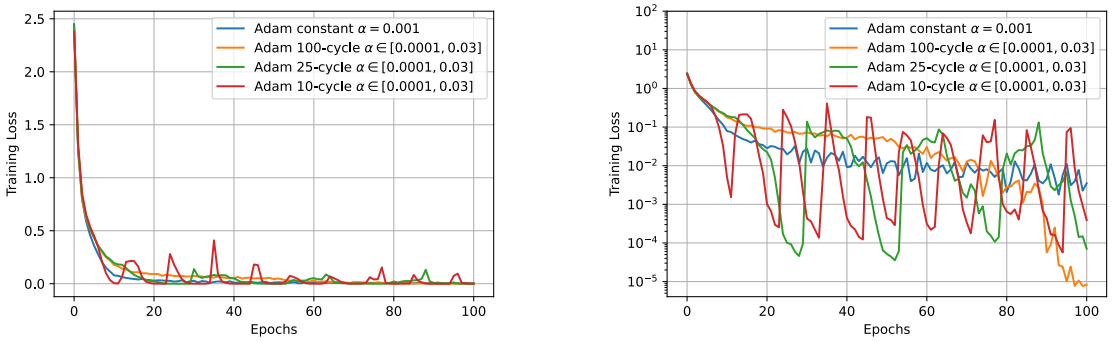
(a) Full plot over 100 epochs.

(b) Cut off after one full cycle (*1-cycle*).

Figure 4.12.: Validation accuracy of ResNet18 on CIFAR-10 trained with Adam with best constant learning rate (0.001) and different CLR routines with best range of [0.0001, 0.03].

percentage points compared to the 2.49 percentage points of validation accuracy increase when introducing CLR to SGD, the effects are significantly less prevalent (see Tables 4.1 and 4.2). Nonetheless, considering that the peak performance of the *25-cycle* scheme is reached after only 27 epochs, compared to 80 epochs needed for the constant scheme, we still consider the introduction of CLR-Adam a success. Comparing the behavior of CLR-Adam presented in Fig. 4.12 to that of SGD with CLR in Fig. 4.5 the same patterns emerge. This proves that CLR provide a speed up of training by order of magnitude for Adam, just as they do for SGD. Thus, it has been shown that super-convergence can be recreated for Adam, something that was not shown by Smith and Topin [1].

Fig. 4.13 shows the cross-entropy loss for training a ResNet18 on the CIFAR-10 data set using Adam with a constant learning rate of 0.001 and CLR-Adam with a learning rate range of [0.0001, 0.03]. For CLR-Adam the *100-cycle*, *25-cycle*, and *10-cycle* routines are shown. Fig. 4.13(a) shows the full data generated and Fig. 4.13(b) shows the same data re-scaled as a logarithmic plot to better show the optima reached. It can be observed that while Adam with a constant learning rate scheme converges relatively smoothly, CLR-Adam enters and escapes minima depending on the learning rate of the respective iteration. CLR-Adam escapes minima when the learning rate is large and enters minima when it is small. This fits to the concept of *Stochastic Escape* presented in Section 4.1. This is another indicator for the validity of this concept where large learning rates are used to escape sharp minima and smaller learning rates are used to optimize within the region surrounding the wider minimum.



(a) Linear scale.

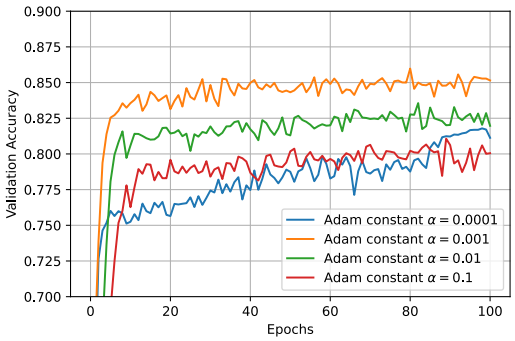
(b) Log-scale.

Figure 4.13.: Training cross-entropy loss of ResNet18 on CIFAR-10 trained with Adam with best constant learning rate (0.001) and different CLR routines with best range of [0.0001, 0.03].

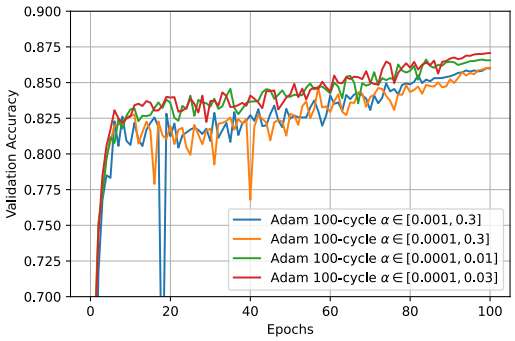
4.4.2. Low Sensitivity to Learning Rate Range

Fig. 4.14 shows different learning rate settings for Adam with constant learning rates and with a *100-cycle* scheme. In Fig. 4.14(a) it can be observed that Adam is already less sensitive to the exact choice of learning rate than SGD (see Fig. 4.7(a)). This is true as long as the learning rate is smaller than 1.0. The test run with a constant learning rate of 1.0 resulted in the divergence of the network. In Fig. 4.14(b) a selection of learning rate ranges for CLR-Adam using a *100-cycle* routine is presented. The learning rate ranges which were tested but not displayed showed very similar behavior to that observed in the plot. CLR-Adam, just as SGD with CLR, shows almost no sensitivity to the exact choice of learning rate range within the ranges tested in this thesis.

Another interesting observation regarding the smallest tested learning rate of 0.0001 can be made. Note that, even though convergence is slow initially, the scheme continues to improve even in the later stages of training when other schemes do not show any improvements anymore. This is not surprising since we would expect smaller learning rates, i.e. smaller regions of confidence as described in Section 2.3, to find good optima but also to approach them slowly. However, this observation is interesting to us as it supports the idea of using 0.0001 as the lower bound for the learning rates used with CLR-Adam. Larger learning rates during the middle of training can lead to faster convergence and maybe also *Stochastic Escape*. Smaller learning rates towards the end optimize for improved results. This finding is in accordance with the explanations for super-convergence used in Section 2.5.



(a) Adam with constant learning rates between 0.0001 and 0.1 (1.0 was also tested but resulted in divergence and is thus, left out).



(b) Adam with *100-cycle* and a selection of different learning rate ranges. Other learning rate ranges tested behaved within the bounds of the ones displayed and are left out for a better overview.

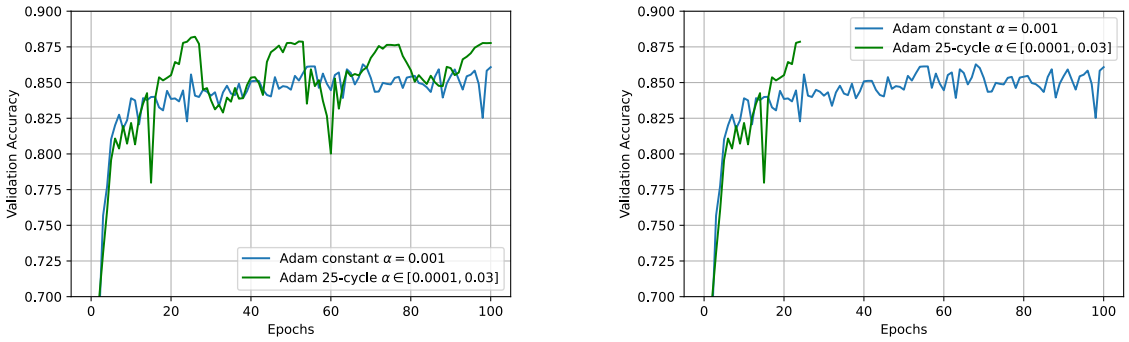
Figure 4.14.: Effect of different learning rates and learning rate ranges on different learning rate schemes with Adam. CLR scheme is less sensitive to the choice of learning rate range than the constant scheme is to the choice of learning rate.

4.4.3. Super-Convergence is Independent of ResNet Depth

Like we did for SGD, we choose the best learning rate range for Adam, which is [0.0001, 0.03], and the best cycle (*25-cycle*) to examine the effects of CLR-Adam using a ResNet50 trained on the CIFAR-10 data set. For comparison of the validation accuracy achieved during training, Adam with the recommended [5] (and empirically best) learning rate of 0.001 is used. Fig. 4.15 shows these results. We observe a slight increase of the effects of super-convergence when using deeper architectures. Actually, this setting was the only one out of the six settings tested that overcame the overfitting gap for the ResNet50 on the non-normalized CIFAR-10 data set. With ResNet18 an increase of 1.3 percentage points of validation accuracy was observed whereas with ResNet50 that increase was at 1.93 percentage points (see Tables 4.2 and 4.4). However, we do not find this increase to be significant, especially when comparing the validation accuracy for ResNet50 over time in Fig. 4.15 to that to ResNet18 in Fig. 4.12. We encourage future researchers to test more cycles and deeper architectures to better understand the dependency of super-convergence for Adam on the depth of the network.

4.4.4. Effects of Super-Convergence Increase when Training Data is Limited

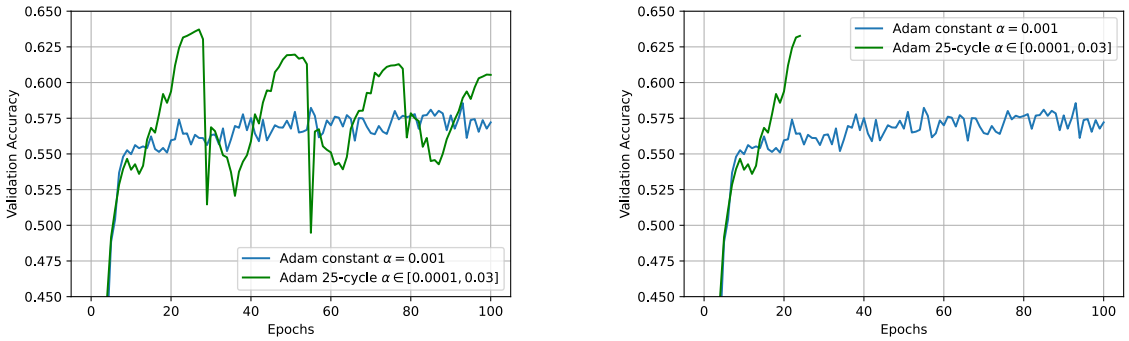
Fig. 4.16 shows the effects of CLR on the behavior of Adam when training a ResNet50 on the CIFAR-100 data set. A validation accuracy increase of 4.71 percentage points can be observed within the first 100 epochs (see Table 4.5). For ResNet50 and CIFAR-10 we only observed



(a) Full plot over 100 epochs.

(b) Cut off after one full cycle (*1-cycle*).

Figure 4.15.: Validation accuracy of ResNet50 on CIFAR-10 trained with Adam with best constant learning rate (0.001) and best cyclical routine (*25-cycle* with learning rate range of [0.0001, 0.03]).

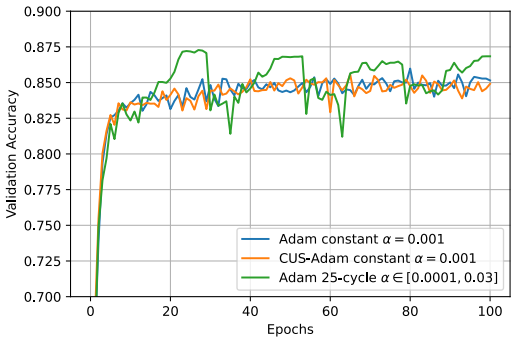


(a) Full plot over 100 epochs.

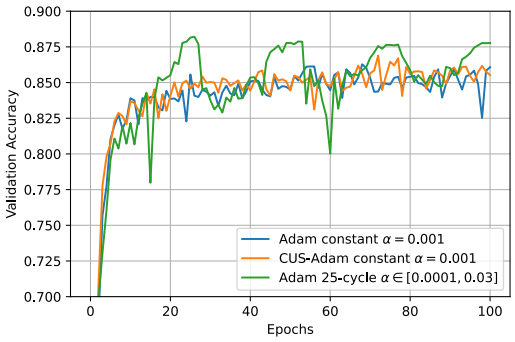
(b) Cut off after one full cycle (*1-cycle*).

Figure 4.16.: Validation accuracy of ResNet50 on CIFAR-100 trained with Adam with best constant learning rate (0.001) and best cyclical routine (*25-cycle* with learning rate range of [0.0001, 0.03]). Note that for CIFAR-100 we use the same scale but display a different range for the validation accuracy than for the CIFAR-10 plots.

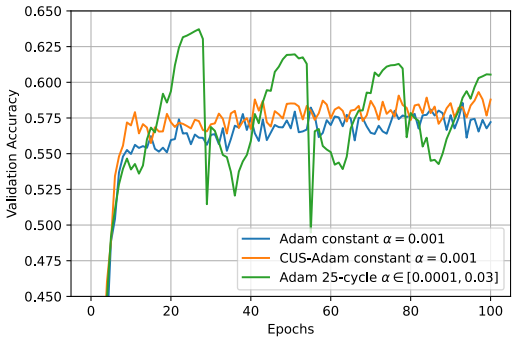
an increase of 1.93 percentage points of validation accuracy (see Table 4.4). Furthermore, CLR-Adam considerably speeds up the training procedure of ResNet50 on CIFAR-100, with the validation accuracy already peaking at 27 epochs. This is in accordance with the results for SGD and the observations of Smith and Topin [1] and thus, shows that Adam behaves similarly to SGD when CLR are introduced.



(a) ResNet18 on CIFAR-10.



(b) ResNet50 on CIFAR-10.



(c) ResNet50 on CIFAR-100.

Figure 4.17.: Validation accuracy of CUS-Adam compared to Adam with a constant learning rate scheme and Adam with a CLR scheme for different ResNet architectures and data sets. Note that for CIFAR-100 we use the same scale but display a different range for the validation accuracy.

4.4.5. CUS-Adam

Lastly, we evaluate the idea of CUS-Adam, introduced in Section 3.2. This implementation does not produce any results which significantly differ from those of regular Adam with the same constant learning rate. Fig. 4.17 shows the results for CUS-Adam compared to Adam with constant learning rates and CLR-Adam with the *25-cycle* routine. Independent of network depth or data set used, the cyclical update step routine does not produce results which significantly differ from that of Adam as introduced by Kingma and Ba [5]. This result indicates that the bias-correction term of the first momentum vector is not very important to Adam's performance. This is plausible since the bias-correction term is a denominator which quickly converges towards one.

4.4.6. Summary

To conclude this section, the effects of CLR on Adam are comparable to those on SGD. We therefore successfully extended the findings of Smith and Topin [1] who have shown that super-convergence emerges with SGD, Nesterov, AdaGrad, and AdaDelta. Showing that super-convergence also emerges when Adam is used as an optimizer indicates that this phenomenon is not tied to the optimizer at hand but rather to the topology of the loss function. CLR seem to exploit these topological properties. This is in accordance with the findings for SGD in Section 4.3. Lastly, we found that CUS-Adam is not a viable alternative to CLR-Adam.

4.5. AdaSecant

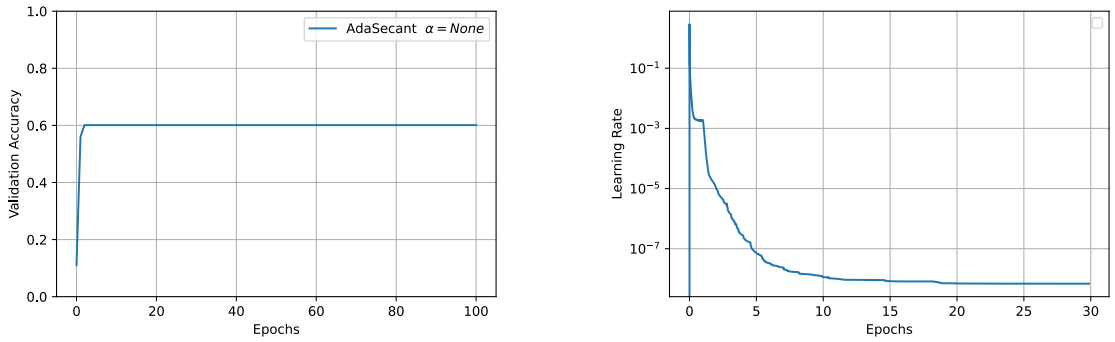
In this section the implementations of AdaSecant and Simple AdaSecant, both as discussed in Section 3.3, are evaluated. Furthermore, the interesting behaviors of the presented algorithms are investigated and connections to the previously discussed learning rate schemes are drawn.

4.5.1. Re-Implementation of AdaSecant in PyTorch

Fig. 4.18 shows the results of training a ResNet18 on the CIFAR-10 data set with AdaSecant for the first 30 epochs of training. This AdaSecant implementation corresponds to the one explained in Section 3.3 and the pseudocode of Algorithm 3.1. The validation accuracy peaks at epoch 2 at 60.09% accuracy (see Fig. 4.18(a) and Table 4.3). The reason for this early, and rather sub-optimal, peak can be found in Fig. 4.18(b), which displays the euclidean norm of the learning rate vector calculated by AdaSecant. Note that we use a log-scale to plot the learning rate. After relatively few iterations the learning rate drops below the value of 10^{-4}. From there on it decreases further. At such low levels the update step of AdaSecant becomes too small to have any effect on the validation accuracy of the network. We were not able to recreate the algorithm proposed by Gulcehre et al. [6] in PyTorch.

4.5.2. Simple AdaSecant Provides Convergence

In Section 3.3 we proposed an alternative to AdaSecant which we named Simple AdaSecant. This algortihm requires a per-parameter learning rate threshold. In experiments the thresholds 3.0, 10.0, and infinity were tested. Any component of the learning rate vector equal to or greater than that threshold would be set to zero. Fig. 4.19 shows the results of that Simple AdaSecant with thresholds 3.0 and 10.0. Using infinity as a threshold produced a divergent training routine and is therefore not shown. Fig. 4.19(a) shows the results for ResNet18 on the CIFAR-10 data set, Fig. 4.19(b) for ResNet50 on CIFAR-10, and Fig. 4.19(c) for ResNet50



(a) Validation accuracy.

(b) Euclidean norm of calculated learning rates (log-scale).

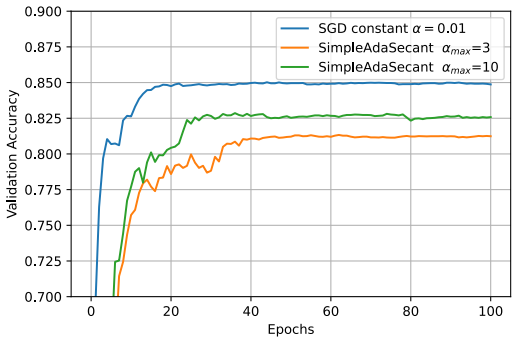
Figure 4.18.: First 30 epochs of training AdaSecant. Validation accuracy cedes to increase as the learning rate drops too low.

on the CIFAR-100 data set. Note that for CIFAR-100 we use the same scale but display a different range for the validation accuracy. While this alternative to AdaSecant produces convergent training routines, the results are not as good as those achieved by simple SGD with an optimized constant learning rate. Hence, they are also worse than the results of super-convergent learning rate schemes using CLR. For deeper architectures the performance gap increases and for data sets with less samples per class it decreases. This behavior is similar to that of CLR and super-convergence as observed by Smith and Topin [1] but different to the behavior of CLR which we observed in Section 4.3, regarding the deepness of ResNet architecture.

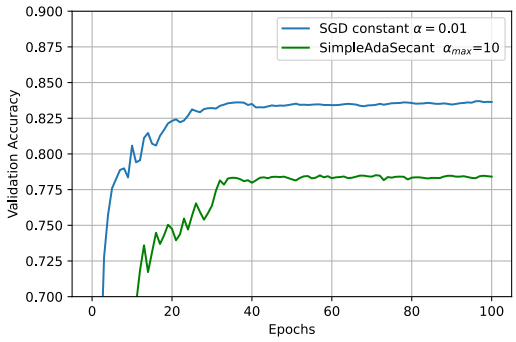
To explain why Simple AdaSecant does not perform as well as other methods we review the concept of sharp minima and *Stochastic Escape*. By construction Simple AdaSecant is approaching the closest minimum, which might be a sharp minimum. This might explain why the trained networks do not perform as well on new data as they do when trained with other optimizers. Fig. 4.20 explains the reason for this belief. In Fig. 4.20(a) we observe Simple AdaSecant to find smaller local minima than SGD does. Nonetheless, Fig. 4.20(b) shows that these minima do not generalize well for new data. This is the same behavior we explained in 2.4 when discussing sharp and wide minima; as discussed by Keskar et al. [18].

To validate this hypothesis, we propose to use SGD with CLR to escape sharp minima before applying the update steps defined by Simple AdaSecant. If the first 13 epochs[3] of a *25-cycle* routine are used, enough large learning rates will have been used to have escaped sharp minima. Simple AdaSecant should then function as the local optimizer that finds the true

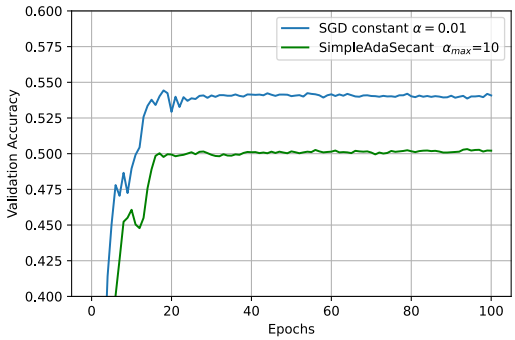
[3] 13 is approximately the middle of the cycle. Changing exactly at 12.5 made the implementation harder and provides no theoretical advantage.



(a) ResNet18 on CIFAR-10.



(b) ResNet50 on CIFAR-10.



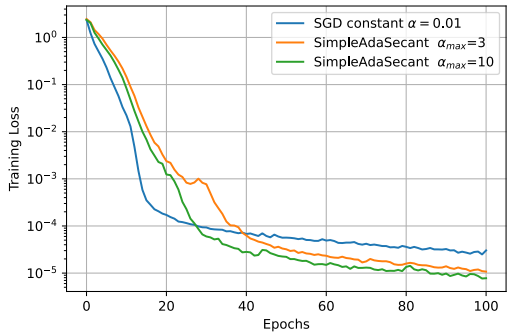
(c) ResNet50 on CIFAR-100.

Figure 4.19.: Validation accuracy of Simple AdaSecant for different networks and data set. Note that for CIFAR-100 we use the same scale but display a different range for the validation accuracy.

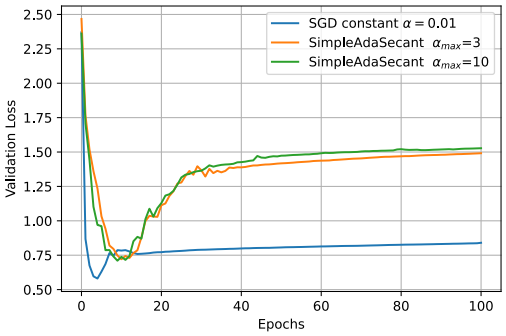
optimum within the region surrounding the wide minimum. Fig. 4.21(a) shows that this idea works. SGD with CLR helps Simple AdaSecant to escape disadvantageous minima. Afterwards Simple AdaSecant can locally improve the results of the network. Nonetheless, the results are still worse than when the second part of the cyclical routine is used as a local optimizer. However, the results are still superior to those of the constant scheme and Simple AdaSecant does improve the network to some degree. Thus, *Stochastic Escape* can be used in order to improve Simple AdaSecant.

4.5.3. Investigating the Learning Rates of Simple AdaSecant

Lastly, we analyze the learning rate calculated by Simple AdaSecant. Simple AdaSecant calculates a learning rate vector with different learning rates for each parameter of the network

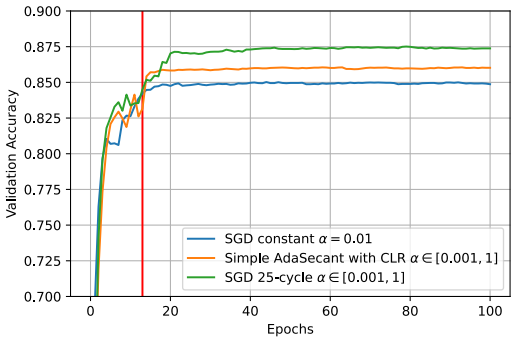


(a) Training loss on logarithmic scale.

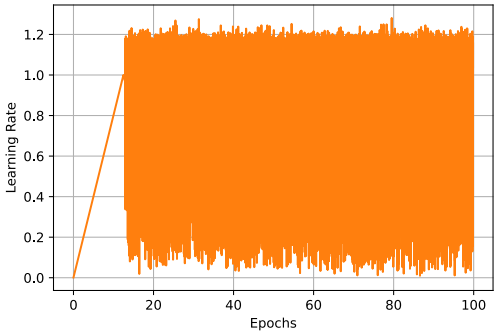


(b) Validation loss on linear scale.

Figure 4.20.: Training and validation loss of ResNet18 on CIFAR-10 with Simple Adasecant with per-parameter learning rate thresholds of 3.0 and 10.0 compared to SGD with optimal constant learning rate 0.01.

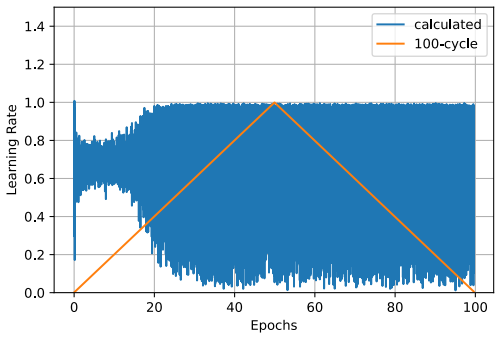


(a) Validation Accuracy, vertical line indicating switch to Simple AdaSecant.

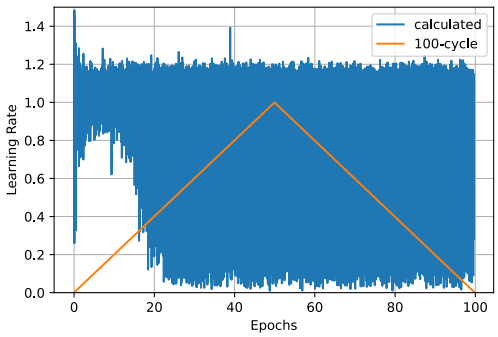


(b) Euclidean norm of learning rate used in each iteration.

Figure 4.21.: Validation accuracy and learning rates per iteration for a scheme using SGD with CLR within the range of 0.001 and 1.0 for 13 epochs and then Simple AdaSecant for the remaining 87 epochs. The network used is ResNet18 and the data set is CIFAR-10.



(a) Setting the per-parameter learning rate threshold to 3.0.



(b) Setting the per-parameter learning rate threshold to 10.0.

Figure 4.22.: Euclidean norm of learning rates calculated by Simple AdaSecant compared to *100-cycle* with best learning rate range of [0.001, 1].

and thus, for each component of the gradient vector. To compare this learning rate vector to a scalar learning rate we take the euclidean norm of that vector. The euclidean norm of a vector is comparable to a scalar because it represents the distance spanned by that vector. Fig. 4.22 shows the learning rate for each iteration of Simple AdaSecant with per-parameter learning rate thresholds of 3.0 and 10.0 compared to the learning rate of a *100-cycle* with empirically optimal learning rate range of [0.001, 1]. We observe that calculated learning rates lie mostly within or close to the bounds of the optimal learning rate range for SGD with CLR. Hence, we validated the optimality of the chosen learning rate range for SGD, even though the sensitivity of SGD with CLR to the exact range chosen is low. We also validate the functionality of Simple AdaSecant as the calculated learning rates lie within a reasonable range. Simple AdaSecant is comparable to SGD but not Adam because no momentum is used for the update step of Simple AdaSecant. Therefore, comparing the calculated learning rate of Simple AdaSecant with the optimal learning rate range for SGD with CLR is feasible.

4.5.4. Summary

To conclude this section, we were not able to re-implement AdaSecant as proposed by Gulcehre et al. [6]. However, we were able to implement an alternative algorithm that converges. Even though our implementation did not achieve state of the art performance, we did validate the core idea of AdaSecant which is the use of a second-order directional Newton Method with a secant approximation of the Hessian. Furthermore, we were able to identify a reason for the relatively poor performance of this naive implementation, namely the convergence towards sharp minima. This can help future researches to improve Simple AdaSecant. Lastly we cross-validated the optimally of the chosen learning rate ranges for SGD with CLR with the

reasonability of the learning rates calculated by Simple AdaSecant. Especially this last insight is an important contribution to understanding the influence of learning rates on the learning speed of neural networks.

5. Conclusion and Outlook

We have shown that the careful setting of learning rates can achieve tremendous improvements to the learning speed and peak validation accuracy of ResNets. CLR have poven to lead to super-convergent behavior, not only for SGD but for Adam as well. The use of CLR can speed up the training procedure by as much as orders of magnitude. Furthermore, it was shown that the exact range of CLR is less important to the speed and quality of convergence than the chosen cycle length. A *10-cycle* provides very fast convergence, with peak accuracy between the 10th and 12th epoch. A *25-cycle*, on the other hand, provides very high peak accuracy, still relatively early in training. This holds true for both optimizers investigated, i.e. SGD and Adam. Our results showed no dependency on ResNet depth of these effects. However, the speed-up of training and accuracy improvement on new data through super-convergence are greater with fewer training samples available.

Furthermore, we provided evidence that super-convergence can only arise from cyclical schemes instead of just any scheme covering a certain learning rate range. We tested randomly generated learning rates and decaying schemes with very large initial learning rates that cover equivalent ranges as the most promising cyclical schemes. Non of these schemes could supply a similar performance boost as CLR provide.

We tested two versions of Adam with cyclical update step sizes. While removing the bias-correction term of the first momentum vector used in Adam (i.e. CUS-Adam) did not provide advantages in training, the introduction of CLR did. We showed that learning rate ranges of 0.0001 and 0.03 provide strongly improved convergence speeds for Adam, just as ranges of 0.001 and 1.0 do for SGD. Therefore, it was shown that aside from SGD, Nesterov, AdaDelta, and AdaGrad (previously shown by Smith and Topin [1]), also Adam can be improved with CLR.

AdaSecant, the descent method proposed by Gulcehre et al. [6] that does not require the learning rate to be set, could not be successfully implemented in PyTorch. Nonetheless, a working alternative was proposed. This Simple AdaSecant algorithm validated the approach of Gulcehere et al. [6], even though we could not validate their original proposal. Simple AdaSecant yields satisfactory, but far from optimal, performance in training ResNets. Interestingly, the learning rates calculated by the algorithm lie within similar ranges of those proven to be successful for CLR schemes. This indicates some validity of the calculation procedure.

Lastly, with the concept of *Stochastic Escape* we provided another intuition for the success of CLR and lack of success for calculated schemes. We provided evidence for this intuitive explanation

by investigating the effects of different batch sizes on different schemes, investigating training loss versus validation loss, and conducting experiments using SGD with CLR before switching to a calculated scheme. Thereby, we were able to indicate that a core component responsible for super-convergence is the destabilization of the loss function caused by very large learning rates during the middle part of a cycle. We hope that this insight will help future researchers to better exploit the advantages of super-convergence, not limited to but with special emphasis on CLR.

Outlook

We encourage future researchers to normalize and padd the images of the CIFAR data sets in order to achieve state of the art results and thereby confirm the results of this thesis. By replicating the results with prepared data, the true superiority of CLR can be shown as the results can then be compared to the schemes used by other researchers. We do belief that the results are replicable, as the convergence speed seems to behave independently from the peak accuracy achieved. Previous experiments with the torchvision models, which perform worse than those used within this thesis, have shown this (see Appendix A.2).

To gain more insights into the effects during late training we encourage to run these experiments for 200 instead of just 100 epochs. We chose the latter as it allowed for more variations. Nonetheless, especially for constant schemes and *100-cycles* the networks might still improve after 100 epochs.

Furthermore, we recommend more variations of the experiments. These include but are not limited to:

1. Smaller lower bounds and larger upper bounds for the learning rate ranges of CLR for both Adam and SGD to test the points at which these schemes show sensitivity to the choice of learning rate size.
2. Test more cycle lengths, such as a *5-cycle* or a *50-cycle* routine.
3. Evaluate against more decay settings with different initial learning rates and different decay speeds. Furthermore, varying the decay type from exponential decay to cosine annealing used by He et al. [2] may also produce interesting insights. These decay settings should primarily be used as a means of control for proving that the cyclical nature of CLR is responsible for super-convergence.
4. Choosing different data sets and neural network architecture types.

As CLR have proven to be very successful in efficiently and effectively training neural networks, we propose to further investigate the cycle itself. Different cycles could be combined in creative ways. A cycle could take the upward slope of a *10-cycle* and the downward slope of a *90-cycle* to produce an unevenly balanced *100-cycle*. Alternatively, schemes that use a

full cycle completion of a smaller cycle and then another completion of a larger circle or vice versa are also options. One can consider to combine exponential decay and CLR, either by introducing a decaying downward slope or by decaying the peaks of a cycle with multiple completions. These variations of the cycles themselves may provide new insights into the reasons for super-convergence as well as provide even faster training routines than those that have been found so far. The cycle variations used by Alyafi et al. [24] can be used as a starting point for defining novel cyclical routines.

Regarding AdaSecant, we want to encourage future researchers to try a different implementation of the original proposal [6]. If these researchers are still unsuccessful with their implementation, we propose to configure and extend Simple AdaSecant with elements inspired from the original proposal. However, we also want to encourage new extensions such as the introduction of momentum to Simple AdaSecant.

To conclude this outlook, we want to propose our view on and vision of the future development of learning rate schemes. The optimal setting, scheduling, or calculating of learning rates remains an open topic for researchers. As new algorithms develop, new insights into the optimization strategies, best suited for neural networks, are being gathered. Currently, any scheme that can successfully exploit CLR has an advantage over others. The escape of disadvantageous minima is something that might not ever be possible to achieve by using curvature information alone. Therefore, we recommend to use CLR with either SGD or Adam for global optimization and *Stochastic Escape*. Nonetheless, we do belief in local optimization through calculated learning rates. Hence, our vision is a combined algorithm that used CLR for *Stochastic Escape* and calculated learning rates in a later phase for local optimization. Such an algorithm must successfully exploit the complexity of the loss function by stochastically finding advantageous minima, i.e. wide minima. Within that region we belief curvature information to be reliable enough to provide further optimization to the network. In Section 4.5 a first attempt at creating such an algorithm was made by combining SGD with CLR with Simple AdaSecant. If improvements to Simple AdaSecant are made, we belief this approach to be the future of learning rate settings.

A. Appendix

A.1. Relationship between Adam's second momentum vector and the distance covered by the gradients

We want to give an intuition for what $\sqrt{E[g^2]}$, and thus $\sqrt{\hat{v}^{(j)}}$, represents. We can interpret $\sqrt{E[g^2]}$, as the (expected) average distance covered by the gradients, separately in each dimension, scaled by the square root of the number of iterations $\sqrt{n_{iter}}$. For this, consider the element-wise evaluation of the gradients $g^{(j)} = \left(g_0^{(j)}, ..., g_n^{(j)}\right)^{\mathrm{T}}$:

$$\sqrt{E[g_i^2]} = \sqrt{\frac{1}{n_{iter}} \sum_{j=1}^{n_{iter}} (g_i^{(j)})^2} \tag{A.1}$$

$$= \frac{1}{\sqrt{n_{iter}}} \cdot \sqrt{\sum_{j=1}^{n_{iter}} (g_i^{(j)})^2} \tag{A.2}$$

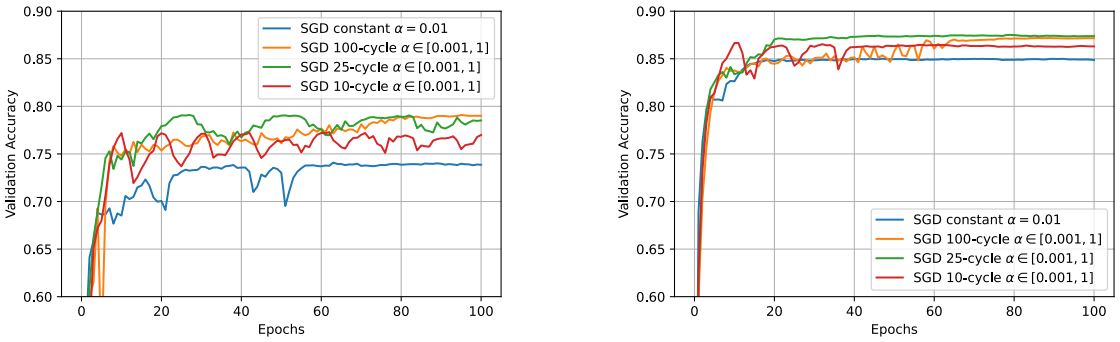
$$= \frac{1}{\sqrt{n_{iter}}} \cdot \left|\left|\left(g_i^{(0)}, ..., g_i^{(n_{iter})}\right)^{\mathrm{T}}\right|\right|_2 \tag{A.3}$$

$$= \sqrt{n_{iter}} \cdot \underbrace{\frac{1}{n_{iter}} \cdot \left|\left|\left(g_i^{(0)}, ..., g_i^{(n_{iter})}\right)^{\mathrm{T}}\right|\right|_2}_{\text{average distance covered in dimension } i}. \tag{A.4}$$

This intuition should help us to comprehend why the two properties (invariance to the scale of the gradient and automatic step size annealing) arise. Firstly, the distance of the gradients normalizes the gradients and thus, provides invariance to their scaling. Secondly, the quotient of the expected value over the gradients and the average distance covered stays at least approximately constant. Nonetheless, if gradients point in opposite directions it may also decrease. In either case we are left with the square root of the number of iterations j in the update step's denominator and a constant c in the nominator. Because the learning rate α is constant, the update step $\Delta^{(j)}$ gets smaller over the iterations j:

$$\Delta^{(j)} \approx -\alpha \cdot \frac{E[g]}{\sqrt{E[g^2]}} \lessapprox -\alpha \cdot \frac{c}{\sqrt{j}}. \tag{A.5}$$

This property of Adam is responsible for the automatic step size annealing.



(a) Results using the torchvision models.

(b) Results from the main part of this thesis.

Figure A.1.: Comparing the performance of SGD with constant learning rates and CLR using the models from this thesis and the torchvision models on CIFAR-10. We observe that super-convergence emerges independently from the maximum accuracy that the networks can achieve.

A.2. Results Using Torchvision Models

Fig. A.1 shows that super-convergence is independent of the maximum accuracy a network can achieve on a given data set. This especially indicates that data normalization and padding will result in similar results as those presented in the thesis. Interestingly, we observe the effects of super-convergence to be slightly stronger when the networks perform worse. This behavior is comparable to the behavior observed when less training data is available. Thus, for normalized and padded data, we expect less of a boost from CLR. However, the performance increase is still expected to be noticeable and significant.

Bibliography

[1] L. N. Smith and N. Topin, "Super-convergence: Very fast training of residual networks using large learning rates," 2018.

[2] 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 (CVPR)*, June 2016.

[3] S. Targ, D. Almeida, and K. Lyman, "Resnet in resnet: Generalizing residual architectures," *arXiv preprint arXiv:1603.08029*, 2016.

[4] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE winter conference on applications of computer vision (WACV)*, pp. 464–472, IEEE, 2017.

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

[6] C. Gulcehre, J. Sotelo, M. Moczulski, and Y. Bengio, "A robust adaptive stochastic gradient method for deep learning," in *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 125–132, IEEE, 2017.

[7] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[8] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," *arXiv preprint arXiv:1712.09913*, 2017.

[9] W. E. L. Ilboudo, T. Kobayashi, and K. Sugimoto, "Robust stochastic gradient descent with student-t distribution based first-order momentum," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[10] N. Ketkar, *Stochastic Gradient Descent*, pp. 113–132. Berkeley, CA: Apress, 2017.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Stochastic Gradient Descent*, pp. 147–148. MIT press, 2016.

[12] Y. Liu, Y. Gao, and W. Yin, "An improved analysis of stochastic gradient descent with momentum," *arXiv preprint arXiv:2007.07989*, 2020.

[13] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning (adaptive computation and machine learning series)," *Cambridge Massachusetts*, p. 298, 2017.

[14] Y. Nesterov, "A method of solving a convex programming problem with convergence rate $\mathcal{O}(1/k^2)$," in *Sov. Math. Dokl*, vol. 27, pp. 372–376, 1983.

[15] 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.

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

[17] Z. Xinchao, "Simulated annealing algorithm with adaptive neighborhood," *Applied Soft Computing*, vol. 11, no. 2, pp. 1827–1836, 2011.

[18] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[19] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, "Sharp minima can generalize for deep nets," in *International Conference on Machine Learning*, pp. 1019–1028, PMLR, 2017.

[20] S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry, "How does batch normalization help optimization?," in *Proceedings of the 32nd international conference on neural information processing systems*, pp. 2488–2498, 2018.

[21] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[22] T. Takase, S. Oyama, and M. Kurihara, "Why does large batch training result in poor generalization? a comprehensive explanation and a better strategy from the viewpoint of stochastic optimization," *Neural computation*, vol. 30, no. 7, pp. 2005–2023, 2018.

[23] A. Lewkowycz, "How to decay your learning rate," *arXiv preprint arXiv:2103.12682*, 2021.

[24] B. Alyafi, F. I. Tushar, and Z. Toshpulatov, "Cyclical learning rates for training neural networks with unbalanced data sets," *Jmd in medical image analysis and applications-pattern recognition module*, 2018.

[25] 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*, pp. 4700–4708, 2017.

[26] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, no. 5, p. 14, 2015.

[27] J. Ehiwario and S. Aghamie, "Comparative study of bisection, newton-raphson and secant methods of root-finding problems," *IOSR Journal of Engineering*, vol. 4, no. 4, pp. 01–07, 2014.

[28] Y. Levin and A. Ben-Israel, "Directional newton methods in n variables," *Mathematics of Computation*, vol. 71, no. 237, pp. 251–262, 2002.

[29] N. Ketkar, *Introduction to PyTorch*, pp. 195–208. Berkeley, CA: Apress, 2017.

[30] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*, pp. 437–478, Springer, 2012.

All links were last followed on.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature