Department of Cognitive Science

University of Tübingen

Bachelor Thesis

in Machine Learning

submitted by

Georg M. Tirpitz

in 2024

Towards Faster Training of Neural Networks with Cyclical Learning Rate Schedules and Soft Reset in the AlgoPerf Benchmark

This Bachelor thesis has been carried out by Georg M. Tirpitz

at the

Fachbereich Informatik

under the supervision of

Prof. Dr. Phillip Hennig

Abstract:

The scheduling of hyperparameters is an important part of deep learning methods. One very crucial hyperparameter is the learning rate. There are different approaches on how to construct these learning rate schedules. It is common practice to use schedules with a warm-up phase followed by a decay phase forming a schedule that we will call one cycle. In this thesis, we will differentiate between sequential schedules with full reinitialization of all parameters after each cycle where the cycles are independent of each other as opposed to cyclical learning schedules where the model and optimizer parameters are kept after every cycle.
We will propose the soft reset method making the reset after each cycle dependent on the performance of the preceding one with the intention to combine the sequential and the cyclical approach and maintain the advantages of both whilst mitigating the disadvantages. In addition we propose early switching criteria for cycles to further improve the training time efficiency of this approach.
All the experiments and research were carried out using the network architectures, codebase and means of evaluation provided by the AlgoPerf benchmark [Dahl et al., 2023] in order to construct and evaluate our proposed method.

Contents

1 Introduction

In the current state of machine learning, artificial neural networks, and deep learning algorithms are the most promising and performant for most complex tasks like image recognition, speech generation, etc. Although the outcome of these methods is in some cases surprisingly accurate, it is also well-known that they are very computationally heavy. It takes a great amount of training data and many iterations on costly hardware to make these methods work efficiently. That leads to high costs in time and energy and thus high CO2 emissions [Dhar, 2020] as well as other expenses. Naturally, it is a question of high interest which algorithms are most efficient for each problem and model, especially when training models on a scale such as GPT [OpenAI et al., 2024] or Gemini [Team et al., 2024]. Even an improvement of a few percent can make a great difference in total time and energy consumption as well as having more accurate models that generalize well.

So, finding out the state-of-the-art algorithms in deep learning and how new algorithms may or may not improve learning is an important question to be answered. Although a huge majority of training learning algorithms for deep learning are based around the idea of some form of Gradient Descent, the way most algorithms tackle this problem can differ very much. It goes from a family of very well-known optimizers like SGD [Robbins and Monro, 1951], Adam [Kingma and Ba, 2014], and Nadam [Dozat] to a variety of newer optimizers [Gupta et al., 2018, Chen et al., 2023, Liu et al., 2024], each claiming to boost performance. The choice of optimizer is then just one factor and has to be tuned to find the optimal set of hyperparameters. There are still other factors, like choosing the right regularization approach like

L2-regularization, weight decay, or others. Furthermore, the choice of well-suiting learning rate schedules or data selection is mostly considered to be a part of the term "deep learning algorithms". Therefore even an optimizer is not one algorithm but rather a family of algorithms regarding its tuning. So there are many choices for combining all the factors into an efficient deep-learning algorithm that might then just be problem or model-specific.

Unlike in other branches of research, there are currently no reliable and standardized scientific methods for reliably comparing the performance of deep learning algorithms with one another. This is due to a set of problems. First, there is no written methodological set of guidelines or a certain scientific standard regarding machine learning that everyone agreed on using and which is easy to look up, like for example the APA (American Psychological Association) guidelines for psychology. Second, to measure the performance of a nondeterministic algorithm, you need an evaluation program or method. This implies the obvious questions of "how to measure" an algorithm and what metric to use. This again creates a subset of problems. When it comes to the question of "how to measure", there is no standardized evaluation code or method. Thus, it is common practice for the authors that propose an algorithm to measure its performance with their own choice of many evaluation metrics and methods. This can of course lead to strong biases in the evaluation of new algorithms and therefore does not sufficiently fulfill scientific methodological requirements. Furthermore, this makes replicating evaluations of deep learning algorithms problematic because the initial authors can always claim that the algorithm was used in an unfavorable model or used for an unfavorable task and tuned with a non-optimal set of hyperparameters, as well as the author of the replication could claim the opposite. The absence of a standardized objective basis of tools and methods of evaluation of deep learning can end in confusion and complicate the scientific discourse regarding deep learning algorithms. The same applies to the question of "what to measure" because there is not just one metric. You can either measure the maximum performance of inference, measured with accuracy, for example, that can be achieved by ignoring training time. You could also measure how many iterations an algorithm needs to hit a certain accuracy that is deemed to be a "satisfying accuracy". The claimed performance of an algorithm can highly depend on the proposed metric, as an algorithm can have great maximal accuracy but suboptimal learning speed or vice versa. Further, this relationship between measures of the performance of an algorithm is also highly dependent on the set of chosen hyperparameters and it can be highly task and model-specific.

The named problems have the consequence that although there is a long list of deep learning algorithms, there is no basis for ranking these algorithms regarding their general performance or the performance on specific problems. As there is no reliable way to compare deep learning algorithms, on the one side developers prefer to stick to well-known algorithms that are considered reliable, such as Adam [Kingma and Ba, 2014] or SGD [Robbins and Monro, 1951], and new algorithms find it difficult to establish themselves. On the other hand, it can seem like there is news about a groundbreaking new algorithm every other day. The AlgoPerf Benchmark Dahl et al. [2023] attempts to make ground for an objective comparison by creating evaluation code and methods that everyone can use to benchmark their proposed algorithms and create a more scientific basis for the discourse in the deep learning community.

This thesis will focus on the factor of learning rate schedules and model reset strategies. We will develop a method to improve the Adam baseline provided in the AlgoPerf Benchmark that utilizes the idea of sequentially using different learning schedules with each training cycle in one training run. The focus of this thesis is to research how different learning rate schedules perform and behave over all the proposed models, called workloads, of the Benchmark. After this, I will propose a refactoring method for the reset after every schedule cycle that aims to keep the advantages of both sequential and cyclic learning rate schedules.

2 Backgrounds

As mentioned in the introduction our proposed methods will be implemented and tested in the software environment code base provided by the Algoperf benchmark. The following description is a quick summary of the methods of the AlgoPerf benchmark that are discussed in the official paper of the project [Dahl et al., 2023].

2.1 The AlgoPerf Competition

The general structure of the AlgoPerf [Dahl et al., 2023] benchmark is in the form of a benchmark competition. This approach has proven to be a good design for comparing machine learning methods in the past, as can be seen with the ImageNet [Deng et al., 2009] competition, for example. The benchmark only compares learning algorithms whilst the models stay fixed. The definition of learning algorithms extends beyond just the set of optimizers that are currently in use. Even an optimizer itself is not just an algorithm, but rather a family of algorithms because their performance is highly dependent on the set of hyperparameters used. To our knowledge, there is yet no efficient algorithm that is not reliant on hyperparameters defined by user but instead is able to tune all parameters of the learning algorithm itself. Thus, things like handling the dropout and regularization are included in the definition of learning algorithms to be compared, as well as strategies that include things like gradient clipping, batch normalization, or even certain strategies to read in or otherwise alter the input data of the input data queue that gets handed over to the submission code by the source code. The following part is to be thought of as a summary of the description in the original Paper [Dahl et al., 2023]. All following information is drawn from their Paper. The motivation behind this benchmark is discussed in the Chapter 1.

2.1.1 Measuring the Algorithms

The measurement of the AlgoPerf benchmark is a so-called "time-to-result" measurement. Thus the measurement is a wall-clock timer that stops when the validation target score of a workload is reached. Therefore the score of the algorithm is the time it took to reach the workload-specific target score. An algorithm is evaluated over all workloads and the final score is an aggregate over all workloads. Not every computation in a training run is within the timed operations. Evaluation steps in which the model gets evaluated on an evaluation set during a training run are not part of the timed operations and thus do not affect the time score, which also means that changing the frequency of evaluation steps in a submission is not permitted. The target of every workload was evaluated by trying out different hyperparameter settings of four popular training algorithms each for 200 training runs per workload. The hyperparameter settings of the training run with the best validation score at the end of the predetermined runtime budget of each workload were then taken. The algorithm was trained again 20 times with these hyperparameter settings but different random seeds and the median of the validation score was set as the target score of each workload. The target-setting runtime of each workload is set as $0.75\times$ the maximum allowed runtime for a submission. The evaluation metric differs from workload to workload and can be seen in Table 2.1.

2.1.2 Workloads

A workload is defined as a set of each a model, data set, loss function, and a target defined in terms of its evaluation metric Table 2.1. An algorithm is run over 6 different kinds of tasks with a total of 8 fixed workloads because image and speech recognition tasks have two different models each. The other tasks are translation, molecular property prediction, clickthrough prediction, and MRI reconstruction. In addition to the 8 fixed workloads, there are six randomized workloads for which the submission must be evaluated. The randomized workloads are drawn from a pool of fixed workloads to which minor modifications have been made. They were introduced to keep submitters from overfitting to the fixed workloads and thus favor algorithms that generalize well when it comes to modifications and changes that one may introduce to a workload.

2.1.3 Competition Hardware and Environment

To be able to compare wall-clock training time the hardware has to of course be standardized, as well as the execution environment and the software environment. The source code by the MLCommons is written in PyTorch [Paszke et al., 2019] and Jax [Bradbury et al., 2018]. For the software versions and execution environment see the MLCommons/algorithmic-efficiency repository. The tuning of hyperparameters can take on any consistent hardware the submitter has access to, but the final evaluation of the submission takes place on 8 $\times$ NVIDIA V100 GPUs with 16GB of VRAM, because it is believed to be a widely used Hardware in most cloud computing systems.

Task	**Dataset**	**Model**	**Loss**	**Metric**	**Validation Target**	**Test Target**	**Maximum Runtime**
clickthrough rate prediction	Criteo1TB	DLRMsmall	CE	CE	0.123649	0.126060	7703
MRI reconstruction	fastMRI	U-Net	L1	SSIM	0.7344	0.741652	8859
Image classification	ImageNet	ResNet-50 VIT	CE CE	ER ER	0.22569 0.22691	0.3440 0.3481	63,008 77,520
Speech recognition	LibriSpeech	Conformer DeepSpeech	CTC CTC	WER WER	0.078477 0.1162	0.046973 0.068093	101,780 92,509
Molecular property prediction	OGBG	GNN	CE	mAP	0.28098	0.268729	18,477
Translation	WMT	Transformer	CE	BLEU	30.8491	30.7219	48,151

Table 2.1: Table of fixed workload sets. Data from Dahl et al. [2023] used with permission from the authors. The abbreviations in the table stand for the following terms: CE (cross-entropy loss), L1 (mean absolute error), SSIM (structural similarity index measure), ER (error rate), CTC (connectionist temporal classification loss), WER (word error rate), mAP (mean average precision) and BLEU (bilingual evaluation understudy score).

2.1.4 Tuning

There are two rulesets for tuning in the benchmark, external tuning and self-tuning. The external tuning allows a limited search over a search space of hyperparameters by having multiple training runs in parallel with different hyperparameter settings and stopping when one of the training runs reaches the target. The self-tuning on the other side has no tuning outside the timed operations and it is just one training run to be scored.

In the external tuning condition, the hyperparameters get tuned throughout 5 training runs called *trials* per workload. The parameters are either drawn from a quasirandom search [Bousquet et al., 2017] over the workload-agnostic search space given by the submission code or the submitter can supply a list of 20 fixed sets of hyperparameter settings. For the submission to reduce the variance of the tuning, there are 5 independent training sessions called *studies* for each workload. Every study consists of a set of 20 trials as mentioned above. In each study the best, meaning the one with the lowest time score, of the 20 trials will be selected and the median over all five studies will then be the benchmark score of the corresponding workload. As stated above the self-tuning ruleset allows no such tuning over a predefined search space of hyperparameters outside of the timed operations of the submission. That means that every kind of hyperparameter tuning takes place within the submitted learning algorithm. That means that the learning algorithm has a fixed set of hyperparameters within the submission/learning algorithm or is automating part of or all of the hyperparameter tuning within the timed operations. Thus a study consists of just one scoring training run. Again the score of each workload will be the median score of 5 studies. To compensate for having to tune within the training loop and within the timed operations the self-tuning training runs are granted 3 $\times$ the maximum runtime.

2.2 Optimizers

As the AlgoPerf benchmark revolves around the learning algorithm it is important to talk about some crucial main parts of the learning algorithm. The largest part of the learning algorithm, as defined by the AlgoPerf, is arguably the choice of the optimizer. There is currently a big selection of common optimizers, some of the popular ones were already mentioned in the Introduction [Robbins and Monro, 1951, Kingma and Ba, 2014, Dozat, Chen et al., 2023, Gupta et al., 2018, Liu et al., 2024]. An optimizer in the context of deep learning can be thought of as an algorithm that aims to minimize a task-specific loss function by adjusting the parameters of an artificial neural network in cases where no analytic solution seems tractable. For most cases that treat deep learning as a non-convex optimization problem, the stochastical optimizers are an extension of the idea of gradient descent [Okewu et al., 2019]. The best way to compare these algorithms is to look at the defining part of the algorithm which is the formulation of the update rule of network parameters. In the following, we will discuss the core ideas of our choice of the most popular optimizers [Robbins and Monro, 1951, Kingma and Ba, 2014, Dozat] and how they relate to each other by looking at their update rules respectively.

2.2.1 Stochastic Gradient Descent (SGD)

One of the simplest optimizers to be discussed here is Stochastic Gradient Descent [Robbins and Monro, 1951]. The underlying idea is the concept of finding a local minimum by going the opposite direction of the gradient written as ∇ of the loss function $\mathcal{L}$ parameterized by the model parameters θ. The parameter η is called the learning rate and is there to stop gradients from enforcing too large updates on the parameters, which could lead to divergence and oscillations instead of convergence to a local minimum. Usually η lies between 0.1 to 0.0001. The loss function $\mathcal{L}$ is task-specific and the parameters θ are defined by the defined model. This leads to the following formulation of SGD following adaptions by Choi et al. [2020].

SGD (η)

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}(\theta_t)$$

2.2.2 Common Adaptations

There are common adaptations to the update of parameters trying to achieve faster convergence or convergence closer to a true local minimum. One approach to achieve this is to introduce heavy ball or Polyiak momentum [Polyak, 1964] to the parameter update by accounting for the previous update v_t by adding this with factor γ to the current update. This leads to a smoother trajectory of updates of the optimizer to counteract oscillations and overshooting by smoothing out rapid changes of gradient direction and jumps in the size gradients between the last and the current parameter update. The factor γ is usually between 0 and 1.

```
**HEAVY BALL MOMENTUM**$(\eta, \gamma)$
$v_0 = 0$
$v_{t+1} = \gamma v_t + \nabla \mathcal{L}(\theta_t)$
$\theta_{t+1} = \theta_t - \eta v_{t+1}$
```

The Nesterov Momentum [Nesterov, 1983] is an improvement of the heavy ball momentum. It evaluates the momentum terms' contribution ahead of the current position by adding up the current gradient to the momentum. It can intuitively be thought about as also including the predicted next step of the optimizer in the momentum by this quality of "looking ahead". This leads to improved convergence in most cases than just using the momentum, especially in high curvature settings. The Nesterov momentum is formulated as:

NESTEROV(η, γ)

$v_0 = 0$

$v_{t+1} = \gamma v_t + \nabla\mathcal{L}(\theta_t)$

$\theta_{t+1} = \theta_t - \eta(v_{t+1} + \nabla\mathcal{L}(\theta_t))$

Another popular adaptation of SGD is RMSprop (Root Mean Square Propagation) [Tieleman, 2012], which is designed to address some of the limitations of traditional SGD by adapting the learning rate for each parameter based on the average of recent squared gradients.

The RMSprop algorithm maintains an exponentially weighted moving average of the squared gradients. This moving average helps to normalize the updates by dividing the gradient by the root mean square (RMS) of these moving averages. This adaptation helps to scale down the learning rate for parameters with large gradients to avoid exploding updates and scale up the learning rate for parameters with small gradients to speed up convergence. The ε is introduced to the formula for numerical stability.

RMSPROP $(\eta, \gamma, p, \varepsilon)$

$v_0 = 1$, $m_0 = 0$

$v_{t+1} = \rho v_t + (1-\rho)\nabla\mathcal{L}(\theta)^2$

$m_{t+1} = \gamma m_t + \frac{\eta_t}{\sqrt{v_{t+1}+\varepsilon}}\nabla\mathcal{L}(\theta)$

$\theta_{t+1} = \theta_1 - m_{t+1}$

2.2.3 The ADAM Optimizer

One of the most commonly used optimizers is the Adam optimizer [Kingma and Ba, 2014]. It is an optimizer combining all the above ideas in one update rule. The Nadam optimizer [Dozat] is an extension of the Adam optimizer with the only difference of applying the Nesterov momentum twice. For our submission, we will limit our experiments to the use of the Adam optimizer due to its popularity, common use, and rather simple construction. The optimizer used for the experiments will follow the definition found in Choi et al. [2020].

ADAM $(\alpha_t, \beta_1, \beta_2, \varepsilon)$

$m_0 = 0, v_0 = 0$

$m_{t+1} = \beta_1 m_t + (1 - \beta_1)\nabla\mathcal{L}(\theta_t)$

$v_{t+1} = \beta_2 v_t + (1 - \beta_2)\left(\nabla\mathcal{L}(\theta_t)\right)^2$

$b_{t+1} = \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$

$\theta_{t+1} = \theta_t - \alpha_t \frac{m_{t+1}}{\sqrt{v_{t+1}}+\varepsilon} b_{t+1}$

2.3 Learning Rate Schedules in Deep Learning

One of the arguably most crucial hyperparameters is the learning rate η_t. It is used in almost every deep learning optimizer to enforce smaller parameter updates, thus keeping the gradients from exploding or diverging and ensuring that the optimizer converges to a local minimum. A frequently taught simplified heuristic is that the larger the learning rate, the faster the training, but also the higher the risk of divergence, causing training instability. Therefore, the challenge of choosing the right learning rate is to balance training speed and training stability. The simplest implementation of the learning rate is a constant learning rate η_t not dependent on the training time, current epoch, or update time t. Although this is still used in some cases, it is not state-of-the-art. Many studies have shown that having a learning rate schedule instead of a static learning rate leads to better convergence [Coleman et al., 2019, Smith, 2017, Loshchilov and Hutter, 2017]. The current knowledge of learning rate schedules often divides it into two phases: the warm-up phase and the decay or annealing phase. A depiction of this is shown below in Figure 2.1. We will call a schedule like that shown in Figure 2.1 one schedule cycle. The warm-up phase, colored in orange in Figure 2.1, is the phase at the beginning of each schedule where the learning rate typically starts at zero and goes up to the maximum learning rate set as a hyperparameter. The warm-up phase is usually around the first $1 - 5\%$ of the length of the whole training cycle. The increase of η_t in the beginning can be modeled by linear, cosine, exponential, or other monotonic functions. The claim is that warm-up phases allow for a greater maximum learning rate by avoiding unstable large changes at the beginning of the training [Gotmare et al., 2018]. The work of Gotmare et al. [2018] even suggests that this is especially true for the last fully connected layers of a neural network. Since the claim of the advantages of warm-up, that it improves training stability, seems to hold throughout the literature, it is common practice. It is also used in the baselines provided in the code from [Dahl et al., 2023] and adopted for our implementation.

The second phase of a schedule cycle colored in blue in Figure 2.1 is the decay or annealing phase. After reaching the maximum learning rate after the warm-up the learning rate is set to the maximum value for a short period before it starts to decay up to the initial or preset minimal value of the learning rate. The decay can be, similar to the warm-up, modeled by linear, cosine, exponential, or step functions as long as they decrease monotonically. The reason here is again to achieve better

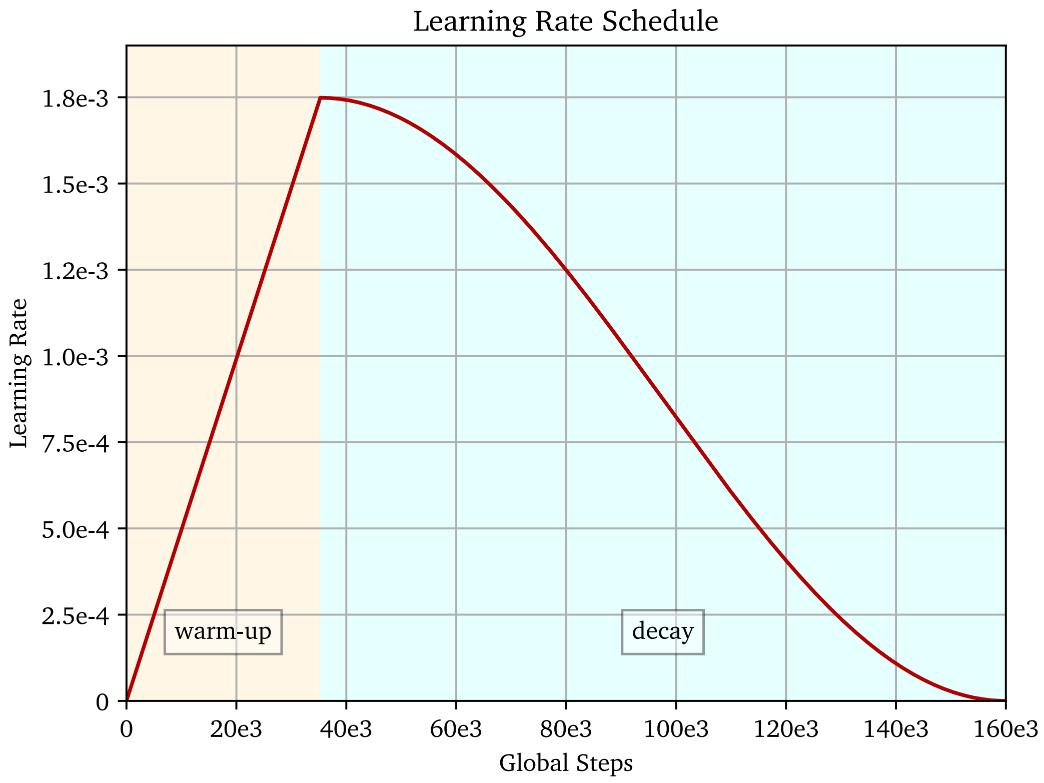


Figure 2.1: Depiction of an example for a learning rate schedule consisting of a warm-up phase, indicated by the orange area, in combination with a decay phase, indicated by the blue area. The dimensions of the depiction are oriented to the dimensions of the experiments below. The warm-up phase is greatly exaggerated for visualization purposes to 0.22. The warm-up factor usually lies between $0.01 - 0.05$.

convergence of the optimizer. The general intuition here is that the decay prevents overshooting a local minimum by making too-large updates while already being in the near region of a local minimum in the solution space. Analogous to the warm-up, having a decaying learning rate allows for a larger maximum learning rate compared to a constant learning rate without causing too much training instability. Thus resulting in faster convergence in the midst section of the cycle without having a severe tradeoff in training stability in the end phase of the cycle by overshooting or diverging out of possible solutions.

As mentioned before, we distinguish between sequential and cyclical learning rates. In the former, there is a full reinitialization of parameters and a full reset of the optimizer momentum at the end of each cycle, making the cycles independent of each other. In the latter, we keep the model parameters and momentum unchanged at each new cycle. The reason to distinguish between a cycle and a schedule is that some authors claim that a training schedule should consist of multiple sequential learning rate cycles, each consisting of a warm-up and a decay phase. This cyclic schedule shall lead to better results in some cases [Smith, 2017, Loshchilov and Hutter, 2017]. There is little known about the reason these cyclical schedules perform so well in some cases. One possible explanation is that the cyclical learning rates deal efficiently with multi-modal functions. Having a cyclical schedule can then help to escape local minima that do not provide as optimal solutions as others [Huang et al., 2017]. Thus, the cyclical schedule provides a better ratio of exploration and exploitation. Further research from Gotmare et al. [2018] suggests, however, that this might be an oversimplification. They base their objection on measuring connectivity between different modes found by different runs of the optimizer and canonical correlation analysis, which does not fully support the hypothesis of Huang et al. [2017]. In addition, they found in their research that the cyclical schedule, while having higher training loss, has a better validation loss, leading to improved generalization. Other work suggests that cyclical learning rates can help in cases where the training duration is unknown and hard to predict. It is therefore hard to determine an efficient schedule beforehand [Zhai et al., 2022]. However, this is not as important in our case where we have limited global steps to reach a preset validation goal.

In the following, we will work with sequential learning rate schedules also consisting of a warm-up and decay phase. The distinction between sequential and cyclical learning rates comes from the fact that we will look at cases where we will not just restart the learning rate but also reset model parameters and/or the momentum of the optimizer. Sequential learning rate then means that there is a complete reset after every cycle, making the cycles independent of each other. Cyclic schedules keep parameters of the preceding run to at least some degree, making the cycles dependent on the preceding one. The distinction between cyclical and sequential learning rates and the topic of resetting will be the main focus of this thesis and our proposed method later on in Chapter 3.

2.4 Training Environment

Our approach to the benchmark is to construct an implementation of our approach following the self-tuning ruleset, which means there are no outer loop modifications of hyperparameters like in the external ruleset but we are granted $3\times$ the full runtime budget. The implemented optimizer is the AdamW algorithm. We adopted the hyperparameters given by the Nadam baseline because it was the only self-tuning baseline and the hyperparameters that are optimal for Nadam are supposedly close to the Adam ones. Most importantly it keeps us from an expensive search for optimal hyperparameters.

2.4.1 Hardware

The computations and training of the benchmark were performed on the hardware of the computing cluster of the University of Tübingen. The Training runs discussed in part 3 were performed on Nvidia A100 GPUs. The number of GPUs varies but the resources for each plot shown in section 3 are mentioned below every graphic. The first runs of the baselines and experiment with the code were conducted on Nvidia 2080ti GPUs. The hardware of the preceding baseline runs that resulted in helpful insights is mentioned in the Appendix with the corresponding plots. The hardware dictated by the benchmark ruleset consists of 8 Nvidia V100 GPUs. We used Nvidia A100 GPUs but were mostly not able to get more than two at a time.

2.4.2 Software

For the software environment, we use the singularity container on the server system instead of the Docker setup. Using a virtual environment instead of Docker or a singularity container would lead to reduced performance. The Python libraries and software environments used for the benchmark are given by the singularity container setup suggested. All the software and libraries with corresponding versions can be looked up at the MLCommons/algorithmic-efficiency repository. Errors appeared whilst trying to run the PyTorch [Paszke et al., 2019] baselines within our environment. It seems like the torch.compile module could not communicate with our Cuda files and therefore our GPU resources. Since later versions of PyTorch did not have the same issue it led us to assume that the problem was due to the version of PyTorch used in the benchmark in combination with the singularity container setup. Due to complications with PyTorch and indications from previous experiments that JAX [Bradbury et al., 2018] has a small speed advantage over PyTorch, we used JAX for all our experiments.

3 Our Approach

We explore in this chapter how sequential and cyclical schedules can be applied to the benchmark. We will then vary the strategy of resetting to see the differences between distinct learning rate schedules with either a hard reset of model and optimizer parameters or just reinitializing model parameters but keeping the momentum of the optimizer. From there on we will propose a strategy to make the hardness of the reset for the model parameters as well as the optimizer after each cycle dependent on how successful the preceding cycle was. We will also propose early switching criteria to stop a cycle and reset immediately when the training metrics do not seem promising for the current cycle. By combining these ideas to efficiently utilize the granted runtime budget we hope to propose a learning algorithm to improve by adjusting the learning rate schedule and the reset strategy with an Adam optimizer.

In the following results of the training runs will be discussed. Please note that the experiments presented here are individual training runs and mostly apply to the OGBG dataset. This limits the generalization of the findings. The results discussed here should be a proof of concept and can not be seen as strong evidence for the efficiency of the proposed soft reset method for general usage in deep learning.

The x-axis of all the following graphs is the domain of global steps rather than the time score of the benchmark to have more means of comparison between different runs. The advantage here is that the reset will always take place at the same global step for the same workload which is not true for the time domain. In addition, the global step dimension is better at plotting different training runs that use different hardware because this heavily influences the training time. It is more complicated in our case to be as consistent with the hardware like the AlgoPerf benchmark suggests, because we depend on a shared cloud computing system for our computation. Even when keeping the same resources for a training run this also leads to asynchrony between training runs with the same datasets and even the same hyperparameter setting due to fluctuation of computing performance. This is easy to see in Figure 3.1, where there is a strong shift of the location of the last reset in the soft reset graph. The fact that the shift in the x-axis direction after the first cycle is almost not noticeable but the shift of the reset after the second cycle is much greater suggests some inconsistencies in the hardware performance or software performance of the computing cluster. This is just a presumptive and the true cause of these differences in the time domain is still to be identified, but will not be the main focus of this thesis. The meaning of the legend of Figure 3.1 will be discussed later on in Chapter 3.

To further justify plotting on the global step dimension we used the Python timeit library to show that our approach does not significantly increase runtime compared to the baseline. The operations added for our proposed algorithm are just storing variables and computing a gradient on a scale that is not significant compared to the rest of the training algorithm. The timeit results are as follows on OGBG on an A100 GPU. For a training step the $model\_reinit\_schdedule()$ function takes on average 5 µs and storing the metrics for early switching takes on average 3 µs. This results in a total 1.28 s on OGBG. After evaluation steps, there are more metrics to store and the early stopping criteria are checked. After an evaluation step the $model\_reinit\_schedule()$ function takes on average 1000 µs and the storing of metrics takes on average 30 µs there are on average 130 evaluation steps, resulting in

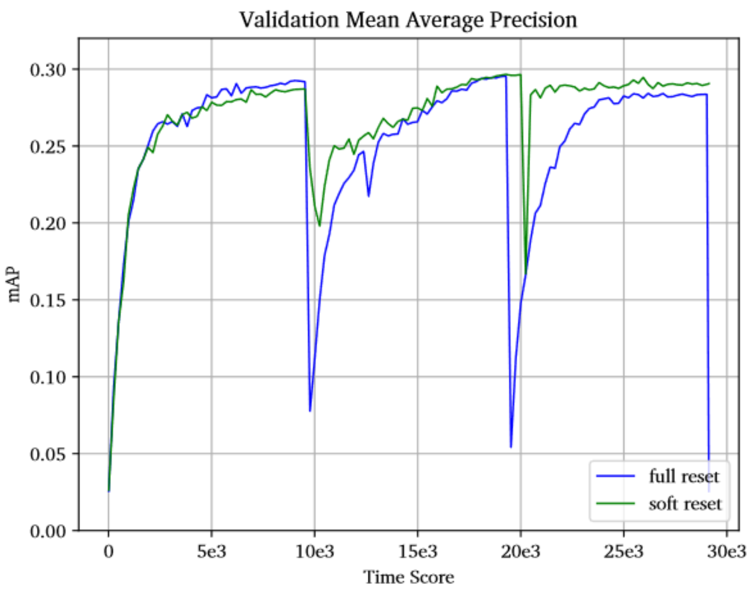


Figure 3.1: A run with full and soft reset (explained in Figure 3.8a) with time score measured as described in [Dahl et al., 2023] on the x-axis. Training on OGBG dataset with one Nvidia A100 GPU.

a total of approximately 0.14 s. At last, our modified reinitialization of the optimizer that gets called at the time of reset takes about 1 s. There are currently 3 resets scheduled on every training run. This results in a total additional runtime of <5 s compared to a total submission time of an average of 30 000 s training time on OGBG.

This shows that our proposed soft reset and early switching algorithm does not contribute significantly to the total training time.

It is also worth noting that we used 2× of suggested global steps instead of 3. In the experiments of Chapter 3 it has always been 3 schedule cycles because, in the preceding experiments, it was observed that it takes less than one target setting budget to reach the target in some cases. The max global steps were set to 2× instead of 3× the global steps as intended by the self-tuning. This is due to limited resources and also to have some kind of a reset between cycles to see if this can have positive effects on the next cycle. The assumption of positive effects stems from the observation that cyclical schedules seem to boost the performance of multimodal models in some cases [Gotmare et al., 2018]. Therefore every schedule cycle could in theory reach the target when performing very well, but in most cases, it is a very soft reset.

3.1 Implementation

Our implementation is based on the NadamW Baseline of the AlgoPerf Github. We changed NadamW into AdamW. NadamW seemed to outperform Adam in our preceding experiments MNIST and OGBG. However, we decided to use Adam for its very common use. The difference between Adam and AdamW is that in the case of AdamW, the weight penalty is introduced after the parameter update rather than before the update like in Adam with regularization where the regularization term is part of the update rule. We used AdamW because it has been the baseline configuration and shown to be more performant than Adam with the classic L2-regularization in our preceding experiments where AdamW had a submission score of 7428 and Adam with L2-regularization had a submission score of 7752. A lower score means it took less time to complete the run.

The other adaptations of the code concern the scheduling of learning rate cycles. We implemented a dictionary containing all the schedule parameters needed to encode a cyclic learning rate schedule with a validation score-dependent reset of model parameters and optimizer momentum as well. Each learning rate cycle consists of a warm-up and a decay phase. Changes were made to the *init_optimizer_state()* given by the AlgoPerf repository to allow for changes in decay strategies and possibilities to alter the learning rate for each cycle as well as rescheduling the learning rate in the case of early switching a cycle. The function *model_reinit_schedule()* mainly implements the scheduling, early switching, and validation score-dependent reset of model parameters. For a detailed description of the Code see the UML diagram in Figure 3.2. The code for the soft and full reset as well as the code for time measurements are on our Github. For the formulation of early switching criteria and validation score dependent reset see the following Section 3.3.

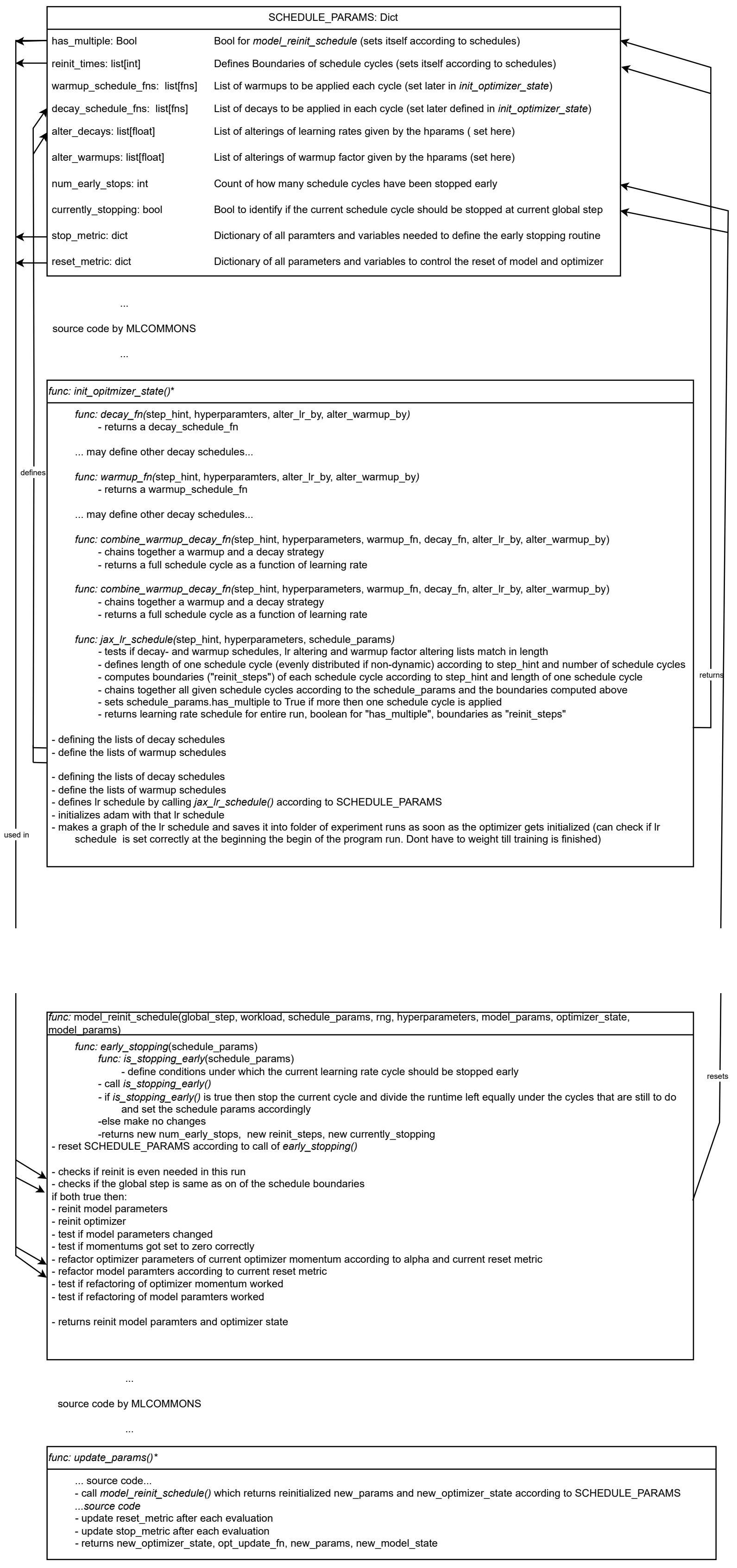
3.2 Exploratory Experiments and Results

Before introducing our proposed algorithm we will show some exploratory experiments that are important to see the behavior of the training runs on OGBG with the AlgoPerf setup. We also show the results for sequential schedules with a full reset and a reset that keeps the momentum of the optimizer which motivated the soft reset strategy.

3.2.1 Behaviour of Different Schedules

The first thing to look at before focusing more on the reset of momentum and reinitialization of model parameters after each cycle is the behavior of different decays and warm-up schedules. Even though it is not the main focus of the thesis it is important to see how different schedules generally behave. These training runs should give an overview of the OGBG dataset's behavior regarding smaller schedule differences. These first experiments were conducted on the OGBG dataset due to its manageable size and computing time.

We see in Figure 3.3 that choosing different warm-up factors in Figure 3.3f as well as choosing different decays (cosine, exponential and linear decay) in 3.3d do not lead to much of a difference between the cycles when it comes to the validation target metric. Choosing much larger differences for the schedules learning rate would heuristically lead to differences in performance [Gotmare et al., 2018] but this will not be the main objective of this thesis. In the case of selecting different maximum



SCHEDULE_PARAMS: Dict	
has_multiple: Bool	Bool for *model_reinit_schedule* (sets itself according to schedules)
reinit_times: list[int]	Defines Boundaries of schedule cycles (sets itself according to schedules)
warmup_schedule_fns: list[fns]	List of warmups to be applied each cycle (set later in *init_optimizer_state*)
decay_schedule_fns: list[fns]	List of decays to be applied in each cycle (set later defined in *init_optimizer_state*)
alter_decays: list[float]	List of alterings of learning rates given by the hparams (set here)
alter_warmups: list[float]	List of alterings of warmup factor given by the hparams (set here)
num_early_stops: int	Count of how many schedule cycles have been stopped early
currently_stopping: bool	Bool to identify if the current schedule cycle should be stopped at current global step
stop_metric: dict	Dictionary of all paramters and variables needed to define the early stopping routine
reset_metric: dict	Dictionary of all parameters and variables to control the reset of model and optimizer

...

source code by MLCOMMONS

...

*func: init_opitmizer_state()**

```
    func: decay_fn(step_hint, hyperparamters, alter_lr_by, alter_warmup_by)
        - returns a decay_schedule_fn

    ... may define other decay schedules...

    func: warmup_fn(step_hint, hyperparamters, alter_lr_by, alter_warmup_by)
        - returns a warmup_schedule_fn

    ... may define other decay schedules...

    func: combine_warmup_decay_fn(step_hint, hyperparameters, warmup_fn, decay_fn, alter_lr_by, alter_warmup_by)
        - chains together a warmup and a decay strategy
        - returns a full schedule cycle as a function of learning rate

    func: combine_warmup_decay_fn(step_hint, hyperparameters, warmup_fn, decay_fn, alter_lr_by, alter_warmup_by)
        - chains together a warmup and a decay strategy
        - returns a full schedule cycle as a function of learning rate

    func: jax_lr_schedule(step_hint, hyperparameters, schedule_params)
        - tests if decay- and warmup schedules, lr altering and warmup factor altering lists match in length
        - defines length of one schedule cycle (evenly distributed if non-dynamic) according to step_hint and number of schedule cycles
        - computes boundaries ("reinit_steps") of each schedule cycle according to step_hint and length of one schedule cycle
        - chains together all given schedule cycles according to the schedule_params and the boundaries computed above
        - sets schedule_params.has_multiple to True if more then one schedule cycle is applied
        - returns learning rate schedule for entire run, boolean for "has_multiple", boundaries as "reinit_steps"

- defining the lists of decay schedules
- define the lists of warmup schedules

- defining the lists of decay schedules
- define the lists of warmup schedules
- defines lr schedule by calling jax_lr_schedule() according to SCHEDULE_PARAMS
- initializes adam with that lr schedule
- makes a graph of the lr schedule and saves it into folder of experiment runs as soon as the optimizer gets initialized (can check if lr
    schedule  is set correctly at the beginning the begin of the program run. Dont have to weight till training is finished)
```

func: model_reinit_schedule(global_step, workload, schedule_params, rng, hyperparameters, model_params, optimizer_state, model_params)

```
    func: early_stopping(schedule_params)
        func: is_stopping_early(schedule_params)
            - define conditions under which the current learning rate cycle should be stopped early
        - call is_stopping_early()
        - if is_stopping_early() is true then stop the current cycle and divide the runtime left equally under the cycles that are still to do
            and set the schedule params accordingly
        -else make no changes
        -returns new num_early_stops,  new reinit_steps, new currently_stopping
- reset SCHEDULE_PARAMS according to call of early_stopping()

- checks if reinit is even needed in this run
- checks if the global step is same as on of the schedule boundaries
if both true then:
- reinit model parameters
- reinit optimizer
- test if model parameters changed
- test if momentums got set to zero correctly
- refactor optimizer parameters of current optimizer momentum according to alpha and current reset metric
- refactor model paramters according to current reset metric
- test if refactoring of optimizer momentum worked
- test if refactoring of model paramters worked

- returns reinit model paramters and optimizer state
```

...

source code by MLCOMMONS

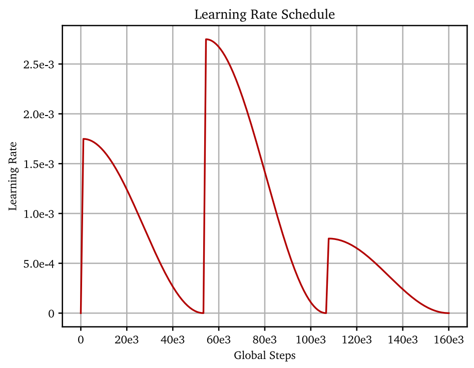
...

*func: update_params()**

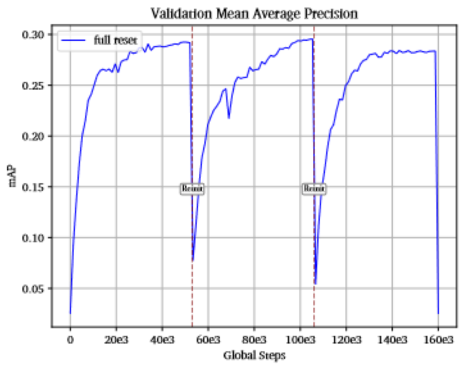
```
    ... source code...
    - call model_reinit_schedule() which returns reinitialized new_params and new_optimizer_state according to SCHEDULE_PARAMS
    ...source code
    - update reset_metric after each evaluation
    - update stop_metric after each evaluation
    - returns new_optimizer_state, opt_update_fn, new_params, new_model_state
```

Figure 3.2: UML diagram of the implementation of our approach using the submission template by the AlgoPerf self-tuning baseline.

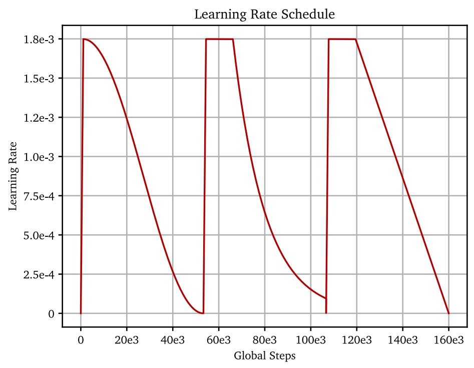
learning rates for the cosine decay in Figure 3.3d there are minor differences to observe. It seems that a greater maximum learning rate seems to promote a steeper learning curve but also causes some training instabilities. However, this cannot be generalized, as we observe individual runs with a potentially noisy data set.



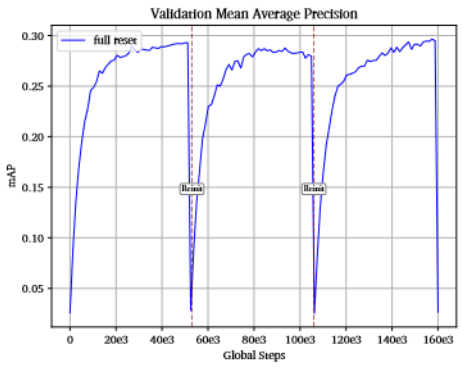
(a) Learning rate schedule with sequential maximum learning rates as $\eta_1 \approx 0.00175$, $\eta_2 \approx 0.00275$, $\eta_3 \approx 0.00075$ for each cycle, cosine decay and warm-up-factor of 0.02.



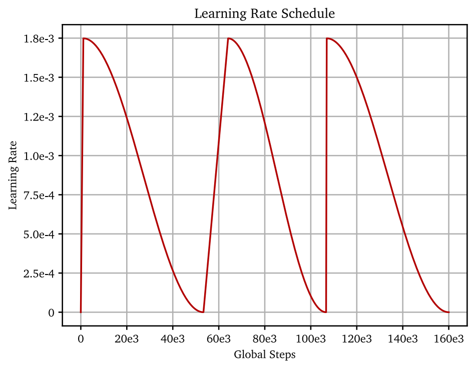
(b) Training run with sequential learning rate schedule of **a)** and full reset. Evaluated on target metric of validation Mean Average Precision (mAP) of OGBG.



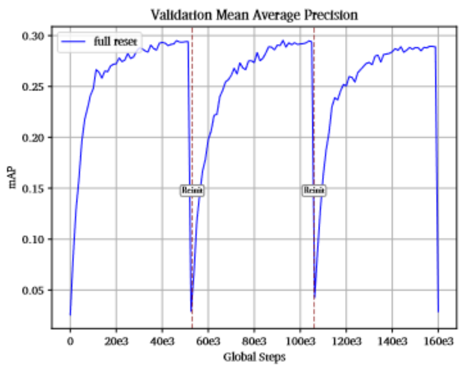
(c) Learning rate schedule with sequential maximum learning rates as $\eta_1, \eta_2, \eta_3 \approx 0.00175$ for each cycle with cosine, exponential, and linear decay and warm-up-factor of 0.02.



(d) Training run with sequential learning rate schedule of **c)** and full reset. Evaluated on target metric of validation Mean Average Precision (mAP) of OGBG.



(e) Learning rate schedule with sequential maximum learning rates as $\eta_1, \eta_2, \eta_3 \approx 0.00175$ for each cycle with cosine decay and warm-up-factor of 0.02, 0.2 and 0.005.



(f) Training run with sequential learning rate schedule of **e)** and full reset. Evaluated on target metric of validation Mean Average Precision (mAP) of OGBG

Figure 3.3: Training runs with different learning rate schedules. Training on OGBG dataset with one Nvidia A100 GPU. η given approximately because we adopted it from the original baseline as $\eta = 0.0017486387539278373$.

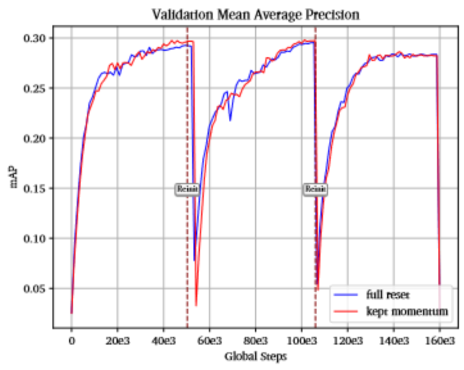
3.2.2 No Momentum Reset vs. Momentum Reset on OGBG

In the experiments seen in Figure 3.4 training runs with the same learning rate schedules and hyperparameters as in Figure 3.3 were conducted but this time keeping the momentum of the optimizer and not setting it to zero. The graphs of Figure 3.4 show these training runs (red curves) side by side with the runs from Figure 3.3 where also the momentum is set to zero (blue curves). The model parameters were reinitialized in every of these cases. The motivation is to see how much the optimizer momentum influences the reset and the preceding schedule cycle.

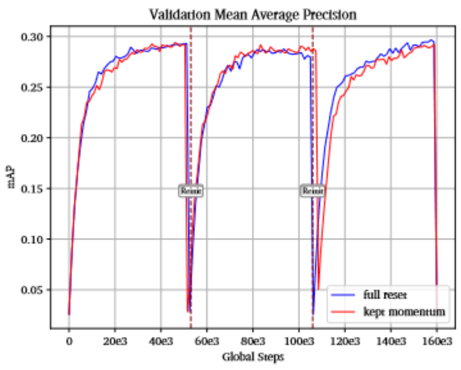
Before discussing the results of Figure 3.8 we would like to remark that some of the graphs fall back to initial validation loss after the third cycle. This is due to the schedule containing a reset at the last global step. For the training runs in the full reset and kept momentum condition, this last reset can be found in the data. Later on there will be training runs where the last reset is not evaluated. The exact reason for this is not known by now but will not affect the discussion of the results.

The graph of Figure 3.4c shows that the runs without resetting the optimizer momentum do not fall back near the values of the first validation loss in step 0. This is because evaluation steps depend on the wall-clock runtime and not the global steps. So it is not ensured that an evaluation step is made immediately after the reinitialization of model parameters, which leaves some time for the loss to improve between the reset and the next evaluation step. This is also why the validation loss does not necessarily return to the initial value after each reset, even with a complete reset (blue curves).

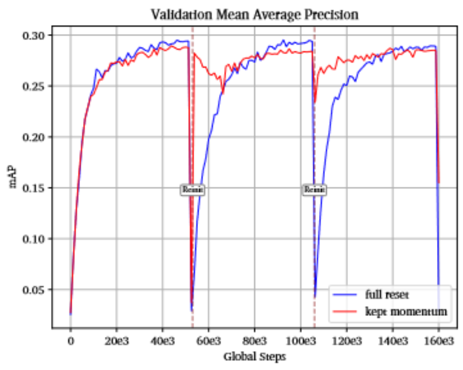
There are two interesting things to observe here. First, there is a big effect of the optimizer momentum in Figure 3.4c but there seems to be no effect in Figure 3.4a and Figure 3.4b. There was a preceding experiment where we ran the baseline without a reset of the momentum and it also showed a strong effect on the recovery of the target metric when keeping the optimizer momentum Figure A.1. This hints that Figure 3.4c is probably not just a coincidence. This effect of faster recovery in itself is not surprising but the strength of this effect in some of the experiments is. We can not explain the inconsistency of this effect. It is to be researched if it is due to unknown behavior in the training or a mistake of any nature on our side which can never be ruled out. To conclude, keeping the momentum of the optimizer when resetting seems to significantly influence the recovery of model performance after a reset in at least some cases. Therefore besides the reinitialization of model parameters, it is important to consider optimizer momentum when choosing a reset strategy for sequential or cyclic training runs.



(a) Training run Figure 3.3b (blue) and training run with the same learning rate schedule and hyperparameters as Figure 3.3b but no reset of optimizer momentum (red).



(b) Training run 3.3d (blue) and training run with the same learning rate schedule and hyperparameters as 3.3d but no reset of optimizer momentum (red).



(c) Training run Figure 3.3f (blue) and training run with the same learning rate schedule and hyperparameters as Figure 3.3f but no reset of optimizer momentum (red).

Figure 3.4: Training runs with different learning rate schedules and reset strategies. One with a reset of model parameters and optimizer momentum (blue) and one with just a reset of the model parameters (red). Training on OGBG dataset with one Nvidia A100 GPU. η is given approximately because we adopted it from the original baseline as $\eta = 0.0017486387539278373$.

3.3 Our Algorithm

The first results of the training runs on OGBG showed that not resetting the optimizer momentum after each cycle can have surprisingly positive effects on the subsequent cycle compared to a full reset. The validation target metric rises surprisingly fast after reinitializing the model parameters but not the optimizer momentum compared to reset everything. This leads to the assumption that cyclic learning rates with no reset or a partial reset improve sequential learning rate cycles with a complete reset of model parameters and optimizer momentum. The assumption of positive effects of cyclical learning rates is also backed by Loshchilov and Hutter [2017], Gotmare et al. [2018]. On the other hand, just keeping the model parameters and optimizer momentum unchanged at each cycle may harm the robustness of the approach of sequential learning rates with a complete reset at the beginning of every cycle. The original idea was to use sequential learning rates to make use of the 3 $\times$ runtime budget granted in the self-tuning rule set by sequentially trying out different learning rates and or decay variations within one training run. The advantage of this approach is that we will have more than one chance to reach the validation target score. The robustness of the sequential approach stems from the fact that the runs are independent of each other and a failed cycle does not affect the other cycles. Preceding experiments showed that there can be runs where almost no learning occurs in some or all of the scheduled cycles. This can be even more significant in noisy datasets. These training cycles most often will not escape their plateau and waste valuable training time. With a cyclical approach, a failed cycle can lead to no learning in all following cycles by keeping bad parameters. An example of this can be seen in Figure 3.5. Having a cyclical learning rate approach that does not reset parameters and momentum each time yields the risk of keeping parameters and momentum of failing cycles resulting in a continuation of a failed cycle or a very bad start of the current cycle resulting in slower learning by having to first escape the plateau of the preceding cycle. In summary, cyclical learning rates have an advantage in keeping parameters and momentum at the start of every cycle instead of resetting potentially promising model parameters and optimizer momentum. This can lead to a faster convergence as can be seen in preceding experiments and Loshchilov and Hutter [2017], Gotmare et al. [2018]. However cyclical schedules lack the robustness of sequential schedules that ensure that a failed training run will not be continued in the subsequent cycle.

We will introduce two approaches to firstly, tackle the problem of wasting time on failed training cycles which probably have no chance of reaching the validation target score, and secondly to combine the faster convergence of cyclical learning rate schedules that keep promising model parameters and optimizer momentum configurations at a subsequent cycle with the robustness and training stability of sequential learning rate schedules. The first problem is tackled by introducing early switching criteria for training cycles in Subsection 3.3.1. The second problem is tackled by introducing a reset strategy of model parameters and optimizer momentum where the degree of resetting or keeping the parameter configuration of the preceding cycle is dependent on the validation target metric in Subsection 3.3.2.

3.3.1 Early Switching of Training Cycles

In the following, we will introduce our early switching. It is the idea of stopping failed training runs early to switch to the next cycle not wasting time completing the failed training cycle. Our early switching conditions are dependent on the validation loss of the model at the validation step t written as $\mathcal{L}(\theta_t)^{val}$. We will introduce a heavy momentum to the validation loss to smooth out the function because we want to make decisions based on the current approximated trend of the derivative of the

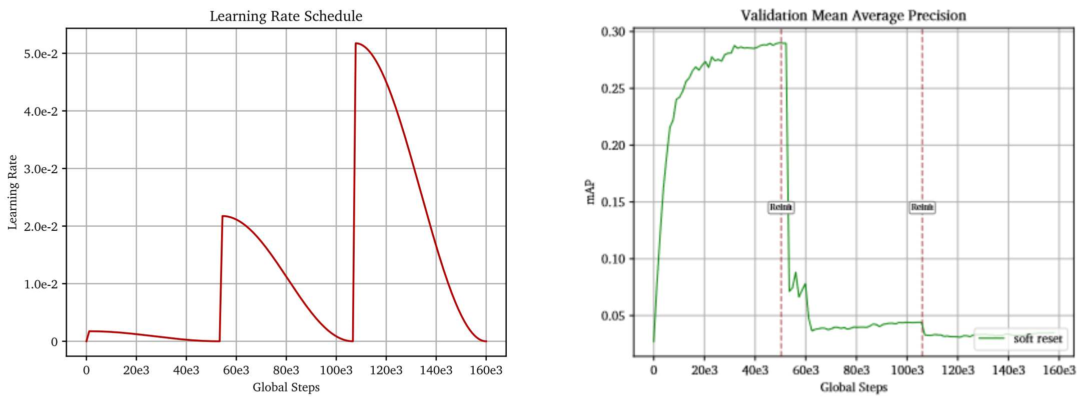


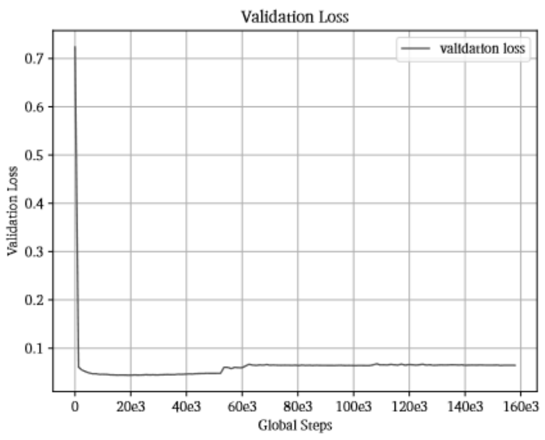
Figure 3.5: A failed run intentionally brought to fail by choosing a learning rate schedule with maximal learning rates of $\eta_1 \approx 0.00175$, $\eta_2 \approx 0.02175$, and $\eta_3 \approx 0.05175$. Training on OGBG dataset with one Nvidia A100 GPU.

loss function rather than every slight change in the loss. This is to prevent making decisions based on a potentially strongly oscillating metric. The function used for defining the conditions for early switching is thus defined as:

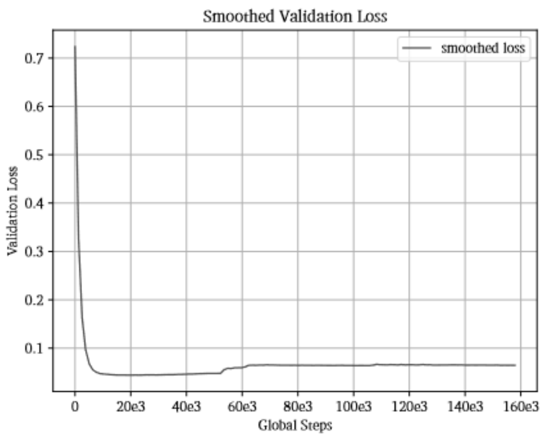
$$l(t) = \lambda \mathcal{L}(\theta_t)^{val} + (1 - \lambda) l(t-1)$$

λ is the factor of the momentum. Aiming for a heavy momentum to prevent early switching due to oscillating validation loss triggering the conditions. Therefore the λ was set between 0.5 to 0.7 in my experiments. It should generally be adapted to match the noisiness of the training data. The Figure 3.6 depicts an example of the different measures of the validation loss mentioned in this chapter. The example corresponds to the training run of Figure 3.5.

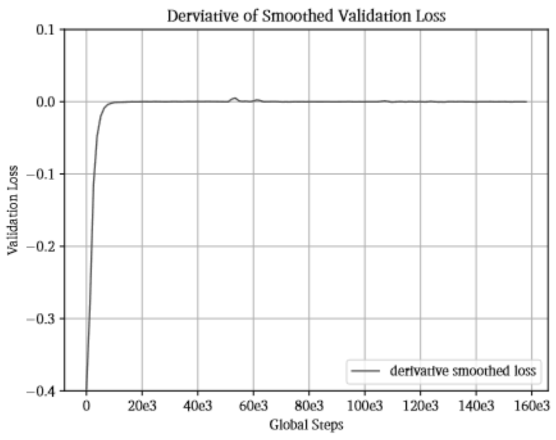
The function $\mathcal{S}(t) : \mathbb{N} \longrightarrow \{0, 1\}$, with $t = 1, ..., T$ where T is the number of validation steps, is the function that takes in the function $l(t)$ and returns 0 or 1 that should be interpreted as boolean values. This function is implemented by *is_ stopping_ early()* in the code. Returning 1 will lead to switching to the next cycle early and 0 will continue the current schedule. $\mathcal{S}(t)$ is defined as:



(a) Valdiation loss $\mathcal{L}(\theta_t)^{val}$



(b) Smoothed validation loss $l(t)$ with $\lambda = 0.6$



(c) Derivative of smoothed validation loss $l'(t)$ with $\lambda = 0.6$

Figure 3.6: Depiction of the different measures of the validation loss. It shows the measurements of the training run shown in Figure 3.5.

$$\mathcal{S}(t) = \begin{cases} 1 & \text{if } l'(t) \geq \varepsilon_1 \\ 1 & \text{if } t - \operatorname*{argmin}_{x \in \mathcal{T}} l(x) \geq k \ \wedge \ ml'(t) + l(t) \geq \min_{x \in \mathcal{T}} l(x) + \varepsilon_2 \\ 0 & \text{else} \end{cases}$$

Where $\mathcal{T} = 1, ..., t$. The variables $\varepsilon_{1,2}$, k, and m are parameters set by the user. The ε is for the first case. The derivative $l'(t)$ of the validation loss with momentum $l(t)$ is tracked and after every evaluation step it is checked if $l(t) > \varepsilon_1$. A positive $l(t)$ means that the validation loss is increasing instead of decreasing which is a characteristic of a failing training run. The ε_1 is the tolerance for allowing a positive derivative. The ε_2 is for the second case. In this case, we first check the distance of the current index and the index of the current global minimum of $l(t)$ with $t - \operatorname*{argmin}_{x \in \mathcal{T}} l(x) \geq k$. If we are k steps away from the current global minimum we check if we can reach this global minimum after m steps estimated by our current state of $l(t)$ and $l'(t)$. This is the term $ml'(t) + l(t) \geq \min_{x \in \mathcal{T}} l(x) + \varepsilon_2$. ε_2 is the tolerance for being above the global minimum.

In most of the experiments, the $\varepsilon_{1,2}$ are set to $0 < \varepsilon \ll 1$. The second stopping condition is redundant in the case where $\varepsilon \leq 0$. The case is intended for greater ε_1, where you allow the loss to recover after a phase of a positive gradient of the loss of k steps. If the current global minimum is not reachable within again m steps in the direction of the current gradient the cycle also stops. The second case is the check whether the model has a chance of recovering and improving even after a short phase of having a positive slope. This is a condition that is interesting when working with very noisy datasets where having a monotonic decreasing $l(t)$ might not be achieved with a reasonable degree of momentum.

Figure 3.7 shows a training run where early switching occurred. For the training

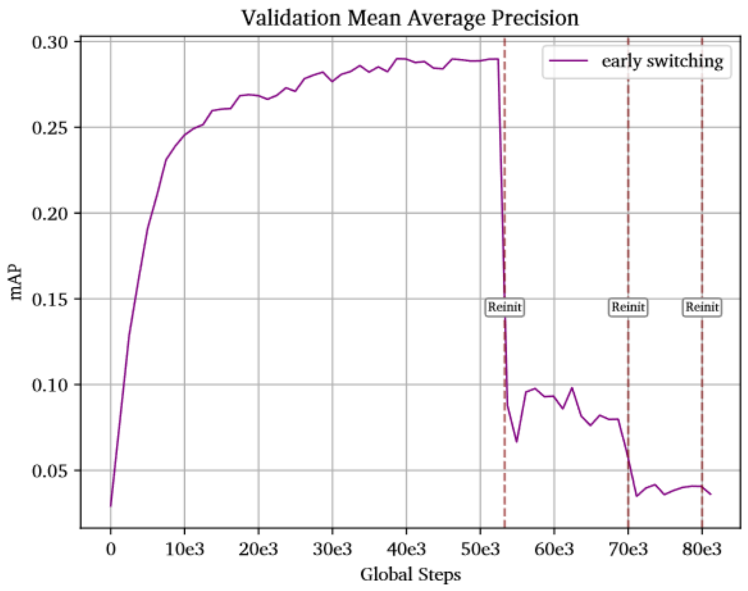


Figure 3.7: A failed run intentionally brought to fail by choosing a learning rate schedule with maximal learning rates of $\eta_1 \approx 0.00175$, $\eta_2 \approx 0.02175$, and $\eta_3 \approx 0.05175$. The parameters for $\mathcal{S}(t)$ are $\varepsilon_1 = 0.01$, $\varepsilon_2 = 0.01$, $m, k = 9$ and $\lambda = 0.7$.The first reset followed the schedule and the following were due to early switching. Training on OGBG dataset with one Nvidia A100 GPU.

run, we choose bigger maximum learning rates for the second and third cycles as displayed in Figure 3.5. The purpose of this was to intentionally introduce training instability to enforce bad cycles showcasing the early switching. In Figure 3.7 the model did reset at step 70001 and 80003. The first reset was a scheduled one. Looking at the graph of the mAP score it seems that the early switching occurred at useful locations. The training run was killed at the moment of the third reset because at this point the number of resets was the same as the number of scheduled cycle. The way we implemented the early switching the run will terminate as this occurs. One alternative way of implementation would be a queue of schedule cycles, allowing for more early switching if needed. Choosing this learning rates is good for displaying early switching but will always end up in early determination of the training run due to a high probability of having bad cycles. It was not possible to find good examples of early switching with good hyperparameters in the given time. The reason for this is that training on OGBG with A100 GPUs in combination with good hyperparameters is very stable making it hard to show examples of early switching. We choose to use greater learning rates in Figure 3.7 for this reason.

Another important thing when using early switching is the choice of parameters. The parameters for Figure 3.7 where $\varepsilon_1 = 0.01$, $\varepsilon_2 = 0.01$, $m, k = 9$ and $\lambda = 0.7$. We believe that these parameters are model-specific, but we will try to give you some pointers for tuning below. It seems that it is a good heuristic to choose $m = k$ to reduce complexity. We also recommend choosing $m, k > 5$ providing enough time to recover after an increase in the validation loss. The chosen momentum λ was between 0.5 and 0.7. The momentum depends on the noisiness of the dataset and the choice of ε_1 and ε_2. The choice of ε_1 and ε_2 is more complex but crucial. ε_1 depends on how noisy the validation loss of the dataset and model of interest is and is therefore very case-specific. ε_2 depends on the interest to have a monotonically decreasing validation loss. If this is not of high interest the ε_2 should be greater than ε_1 since the second case of $\mathcal{S}(t)$ can be very restrictive leading to too many early switches.

The limited time given to construct this thesis kept us from tuning the early switching perfectly for every workload. Too liberal criteria of switching to the next cycle led to not being able to switch out bad cycles we deemed to be a waste of training time, killing the advantage it was supposed to bring. Too strict criteria switched out noisy but promising cycles, which led to spearing remaining cycles over too much training time. This could at least be compensated by having a queue of hyperparameter settings that exceeds the number of originally planned cycles but would still not make efficient use of training time by stopping promising cycles. Due to the lack of sufficient tuning of the early switching, we decided to have very liberal switching criteria to still be able to display the soft reset for all workloads. We still are confident that an appropriate tuning of early switching is a valuable addition to the approach.

3.3.2 The Soft Reset Method

As said before the goal is to combine the advantages of cyclical and sequential schedules. We therefore want to have the robustness of sequential schedules regarding failed learning cycles but do not want to throw away promising model parameters or optimizer momentum after a more or less successful training cycle. This would cost valuable training time and resources. The approach to this problem is not to completely reset the model parameters and optimizer momentum but to make it dependent on the value of the validation target metric of the workload respectively. The validation target metric is the metric that seems to be the best fit because it is the metric that says the most about the actual performance of the model on yet unseen data. For this, we propose the factor of the amount of reset of the model parameters τ as a formula consisting of the validation target score y_{target} and the current value of the validation target metric y_t at the validation step t. The scalar for rescaling τ is a normalized difference between the current and the target value of the respective target metric. We used the $max()$ to adjust for cases where $y_t > y_{target}$. Those are cases where the target metric at the time of the reset y_t exceeds the target y_{target}. The $max()$ keeps us from using negative numbers which would not be the behavior of the cyclical approach we try to reproduce in this case.

$$\tau = \frac{max(y_{target} - y_t, 0)}{y_{target}}$$

This formula was intended for OGBG, FastMRI, WMT, and Criteo at best. For those the validation score is a score that should be as high as possible like for example the mAP for OGBG. For some of the other workloads (for example Imagenet), the validation target score should be as low as possible. The current formula is a proof of concept and only aimed at workloads with target metrics that increase with increasing training time. The formula should also be adapted for decreasing target metrics as future work on the approach.

The scalar τ will be used to define the ratio between using the current parameters at step t and the reinitialized parameters at the beginning of each new cycle. The formula for rescaling parameters at the beginning of each new training cycle of the schedule is as follows:

Soft reset of model parameters:

$$\theta_t = (1 - \tau)\theta_t + \tau\theta_{reinit}$$

The function in the code for θ_{reinit} is given by the AlgoPerf source code and initializes the parameters following the same distribution but with a different rng seed each time it is called.

Soft reset of optimizer momentum:

$$m_t = (1 - \tau\alpha)m_t$$
$$v_t = (1 - \tau\alpha)v_t$$

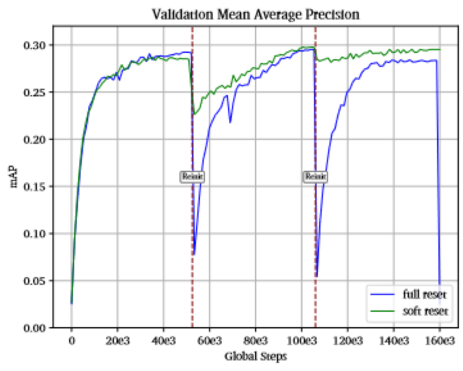
Where m_t and v_t are the first and second moment averages of the Adam optimizer respectively. It is only one summand in the formula because resetting the optimizer means setting the momentum to zero. The variable $0 < \alpha < 1$ should reduce the amount of rescaling. This is due to the observation of preceding experiments, that keeping the momentum of the optimizer can be very effective in some cases and not as harmful as bad model parameters in failed training cycles.

This approach makes two restrictive assumptions. The first is that we know the exact target score of each workload in advance, which is not common outside the AlgoPerf ruleset.

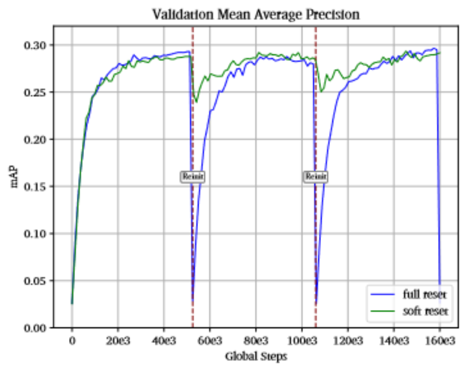
The second one is that we assume our proposed schedule has cycles of roughly the length of one target runtime. With durations of a cycle much less than one target setting runtime, it is of course very unlikely for a training cycle to get near the target score even if it is very promising because the target setting runtime is the estimated time to reach the target score. In this case, we would enforce a much harder reset on every training cycle much less dependent on the actual performance of the training cycle.

Figure 3.8 shows the soft reset approach to resetting parameters and optimizer momentum compared to their full reset counterparts in terms of hyperparameter settings. The first thing to notice is that the runs with the soft reset start with way higher mAP scores after the reset due to the faster recovery of the validation loss. Second, these runs reach the asymptotic learning phase, meaning the phases where almost no increase in the target metric occurs, very soon after the reset. The results show that the approach leads to a softer reset of the model and faster recovery in terms of the target metric. Thus, the soft reset approach shows the effects on the learning behavior we were trying to achieve.

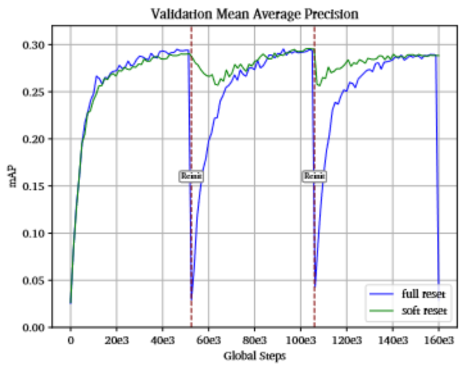
The results show that the soft reset is not as consistent as the full reset. With constant we mean how similar the learning behavior is compared to one another after a reset. We see in Figure 3.4 that the shape of the mAP score curve looks very similar after every reset for the full reset condition. This is not the case for the soft reset. This is seen in Figure 3.8a where at the start of the second cycle there



(a) Training run Figure 3.3b (blue) and training run with the same learning rate schedule and hyperparameters as Figure 3.3b but with soft reset method (green). The τ values for the resets are 0.014 and 0 respectively.



(b) Training run Figure 3.3d (blue) and training run with the same learning rate schedule and hyperparameters as Figure 3.3d but with soft reset method (green). The τ values for the resets are 0 and 0 respectively.



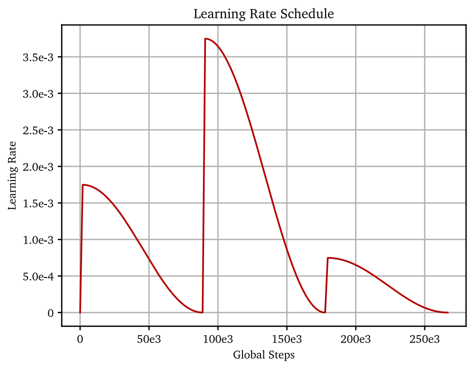
(c) Training run Figure 3.3f (blue) and training run with the same learning rate schedule and hyperparameters as Figure 3.3f but with soft reset method (green). The τ values for the resets are 0 and 0 respectively.

Figure 3.8: Training runs with different learning rate schedules and reset strategies. Once with a complete reset of model parameters and optimizer momentum (blue) and once with the soft reset method (green) proposed in section Subsection 3.3.2. Training on OGBG dataset with one Nvidia A100 GPU. η is given approximately because we adopted it from the original baseline as $\eta = 0.0017486387539278373$. The α for the reset of the optimizer momentum was set to 0.5 for all runs.

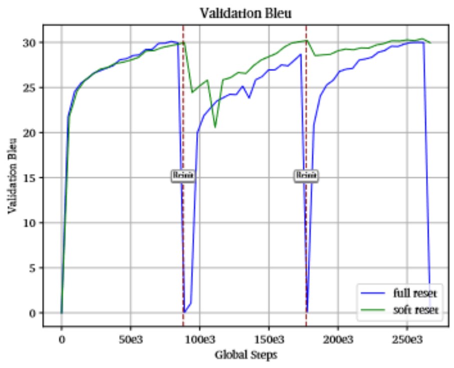
is a noticeable reset in the mAP curve whereas at the start of the third cycle, the mAP score is almost immediately back to the asymptotic phase of the preceding run. This is probably due to the values of τ that determine the proportion of reinitialized parameters. The τ for the first rest in Figure 3.8a is $\tau = 0.014$ and $\tau = 0$ for the second reset. Heuristically, you would expect that a cycle with a high last mAP score and therefore small τ value is followed by a softer reset and fast recovery in the next cycle. In Figure 3.8a we see an example of this. The other results do not show examples of this due to $\tau = 0$ for every rest in Figure 3.8b and Figure 3.8c. We did not expect to reach the target in this many training runs. We see that the approach is almost identical to a cyclical approach with no reset keeping the advantages of the cyclical approach in the case of a promising training cycle. Even for the resets with $\tau = 0$ we see at least some kind of a reset or disruption in the learning curve. We suspect that this could be due to the change in learning rate, but this assumption cannot be supported by the results.

Another interesting remark for Figure 3.8c is that the mAP curve in its shape is very similar to the corresponding curve of Figure 3.4c where the model parameters were reinitialized but the optimizer momentum was kept. Both show a quick recovery after the reset, followed by a short phase of decline. We can not offer a satisfactory explanation for this occurrence of similarity in different conditions.

The last important observation of the results depicted in Figure 3.8 is the maximum mAP score of each cycle. Although the soft reset does lead to a way faster recovery of the mAP score the results only show an improvement of the maximum mAP score in the last cycle of Figure 3.8a. The depicted training runs of the OGBG dataset so far do mostly not show a steady incline of the mAP score but rather enter a phase of asymptotic learning. In our results, a faster recovery due to a soft reset does lead to an earlier entry in this asymptotic phase but does not improve the maximum score in most cases. This might be a characteristic of the OGBG dataset.



(a) Learning rate schedule for WMT dataset with maximum learning rates of $\eta_1 \approx 0.00175$, $\eta_2 \approx 0.00275$, $\eta_3 \approx 0.00075$ for each cycle.



(b) Training run with full reset (blue) and training run with soft reset method (green). The τ values for the resets are 0.028 and 0.024 respectively.

Figure 3.9: Training runs with different learning rate schedules and reset strategies. One with a complete reset of model parameters and optimizer momentum and one with the soft reset method proposed in section Subsection 3.3.2. Training on WMT dataset with one Nvidia A100 GPU. η is given approximately because we adopted it from the original baseline as $\eta = 0.0017486387539278373$. The α for the reset of the optimizer momentum was set to 0.5 for all runs.

To see if the soft reset method also works on other workloads we trained on the WMT dataset. The results of Figure 3.9 shows the comparison between a sequential schedule with a full reset and a cyclic schedule with the soft reset method. You can see in Figure 3.9b that the training on WMT with the soft reset shows similar characteristics as the OGBG training runs. At the time of reset, there is a way smaller dip in the Bleu score, which is the target metric of the WMT dataset, than in the full reset condition. This effect is very pronounced at the last cycle where the reset is almost not noticeable for the soft reset with condition ($\tau = 0.024$) but goes down to a Bleu score of almost 0 for the full reset. We also see again that a greater of $\tau = 0.028$ in the first reset does lead to a harder reset than for the second with $\tau = 0.24$. Even subtle differences in τ seem to affect the softness of the reset.

Unlike most runs on OGBG, the soft reset does lead to a higher maximum target metric score as can be seen in the second cycle of Figure 3.9b.

The learning behavior after the first reset is also worth mentioning. It shows the same short phase of decreasing target metric score that can also be observed in Figure 3.8c and Figure 3.4c. The fact that Figure 3.4c has a different learning rate schedule with the same maximum learning rate for every cycle speaks against the explanation of this effect due to the same learning rate schedule. The reason for this short phase of decrease is therefore unknown.

Figure 3.9 shows that the soft reset also works as intended for the WMT dataset. This reinforces the assumption that the advantages of the soft approach might generally apply to deep learning methods.

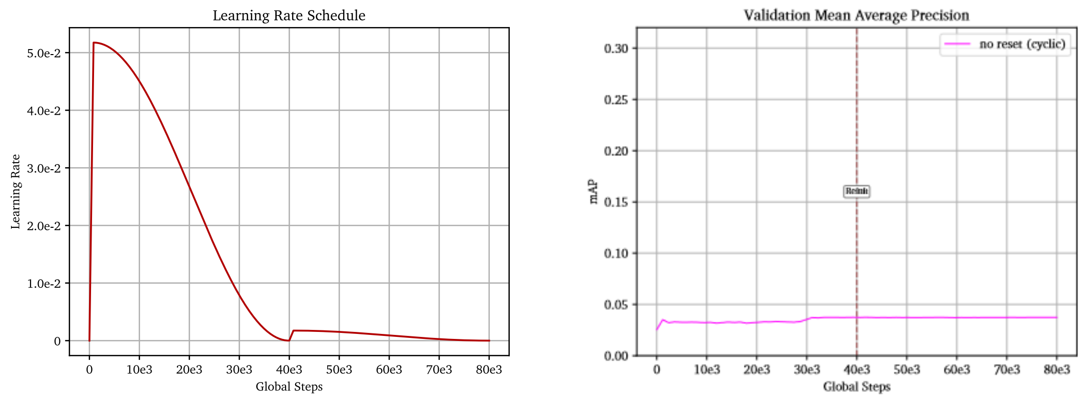


Figure 3.10: A cyclical training run with no reset and only 1× the target setting budget. The first cycle is intentionally brought to fail by choosing a learning rate schedule with maximal learning rates of $\eta_1 \approx 0.05175$ and the second cycle with the baseline learning rate of $\eta_2 \approx 0.00175$. Training on OGBG dataset with one Nvidia A100 GPU.

We showed that the soft reset approach does have a big advantage over the sequential approach for cycles with good training performance. This is because we do not perform a full reinitialization of parameters and potentially discard promising model parameters which would waste valuable training time. This is almost the same reset behavior as for cyclical learning rates with no reset if the cycle had a successful preceding training cycle. So in that case the soft reset is similar to the cyclical approach. The advantage as mentioned also in Subsection 3.3.1 is that the soft reset is much more robust to failed cycles. The Figure 3.10 shows the disadvantage of the cyclical approach. In Figure 3.10 we set the learning rate of the first cycle very large to force a failed cycle. We see that keeping the parameters at the reset leads to a flat-lining of the target metric even after the reset where we continue with a cycle with the baseline maximal learning rate.

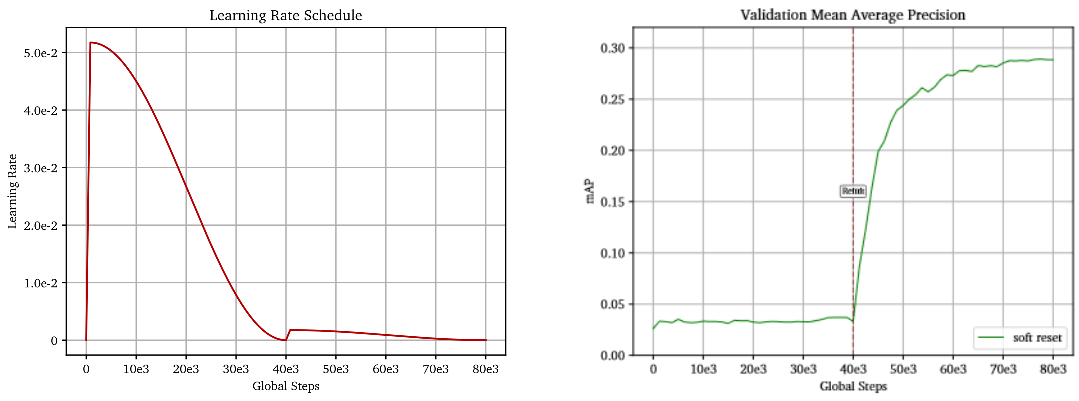


Figure 3.11: A training run with the soft reset approach and only 1× the target setting budget. The first cycle is intentionally brought to fail by choosing a learning rate schedule with maximal learning rates of $\eta_1 \approx 0.05175$ and the second cycle with the baseline learning rate of $\eta_2 \approx 0.00175$. The τ scalar for the reset was 0.868. Training on OGBG dataset with one Nvidia A100 GPU.

This flat-lining of the target metric does not happen for the soft reset condition. The results in Figure 3.11 show that the training run is able to recover after a failed cycle. The soft reset approach enforces a hard enough reset with $\tau = 0.868$ to recover the performance. This shows that the soft reset does not have the vulnerability regarding failed training cycles as the cyclical approach.

We were able to show that the soft reset approach shows the advantages of the cyclical approach by reusing promising parameters after a promising training cycle.

We also showed that, unlike the cyclical approach, it is robust to failed training cycles, keeping the advantage of the sequential approach. It seems that we were able to combine the cyclical and sequential approaches, discarding the disadvantages and mostly getting the best of both worlds.

4 Conclusion

In the following, we will discuss the chances and advantages of our approach as well as the limitations and future adaptations that are to be made to make this approach usable for a wide variety of deep learning problems.

4.1 Summary

In this thesis, we implemented our approach with the submission template and environment of the AlgoPerf Benchmark [Dahl et al., 2023]. We utilized the codebase and means of evaluation provided by the benchmark to test our hypothesis on some of the workloads of the benchmark. The approach is to use the AdamW optimizer with the thesis focusing on the learning rate schedule and the strategy of resetting parameters following the schedule. The initial idea was to efficiently use the extra granted runtime budget by trying out different learning rate schedules sequentially in one training run. Sequentially means that after every learning rate schedule, there is a full reinitialization of the model parameters and a reset of the optimizer momentum to zero. In contrast, we will call schedules that go through different learning rate settings without resetting or reinitializing any parameters cyclical schedules. Observations of the first baseline experiments and insights from literature about cyclic learning rate schedules with no reset of parameters Smith [2017], Gotmare et al. [2018] lead to a new approach to the reset. This is to further optimize the schedule and fully utilize the runtime. We proposed a method combining two ideas to be able to better utilize runtime, keeping the advantages of robustness against training instability and also being able to try out different learning rate schedules within the training loop. These two elements of the algorithm are the strategy of resetting the parameters and early switching criteria for training cycles that show training instability or a decline in learning performance. For the reset strategy, we propose a method we call the soft reset method. The idea is to utilize the runtime more efficiently by forcing a soft reset for promising parameters and momentum but performing a type of hard reset in the event of a previous failed learning cycle Subsection 3.3.2. This is utilized by making the hardness of the reset dependent on the workload's target metric score at the moment of the reset which is the beginning of a new schedule cycle. In addition, the early switching criteria are implemented to make more efficient use of time by stopping schedule cycles that do not seem promising due to declining performance or severe training instability. The cycles get switched early to gain more time for other learning rate settings in the new cycle (Subsection 3.3.1). These changes to the learning algorithm are expected to make more efficient use of the time budget of the Benchmark.

4.2 Advantages of our Approach

The results of the experiments with the soft reset approach are shown and discussed in the Chapter 3. It showed that our formulation of a soft reset has the effect of a much faster recovery of performance after a reset that follows a cycle that showed sufficient learning progress. The assumption that having a soft reset boosts the performance of the following cycle cannot be confirmed by the results. It is still to be researched with more data and appropriate metrics if our approach does lead to a better balance of exploration and exploitation for multimodal problems by introducing a soft reset even in good-performing cycles leading to more exploration.

The fast recovery of performance after a reset does provide a clear advantage for similar use cases with the same goal as this benchmark, especially when combining it with early switching mechanisms. The fast recovery allows for a higher frequency of schedule cycles. This is very desirable because one motivation for the sequential/cyclic approach to this benchmark was to be able to try out multiple learning rates in the course of one training run. The soft reset allows for a higher frequency of schedule cycles by omitting the first phase of learning that starts from the ground up ultimately enabling us to try out more learning rate settings, which is an advantage over the sequential approach. This is not restricted to trying out different learning rate settings but can be extended to any hyperparameter search.

Another advantage we hoped for was to keep the robustness against failed cycles with our formulation of the soft reset, which is an advantage over a cyclical schedule that does not change the parameters at the ends of cycles. Not changing parameters after a cycle when trying out different hyperparameter settings leaves the model vulnerable to failed cycles that could lead to dead ends of learning for the whole training in the worst cases. We can observe this being true for most of the conducted experiments.

The advantages above are assumed to be potentiated by also introducing early switching. The results of Subsection 3.3.1 show that the early switching works somehow satisfactorily for the runs displayed, but it took tuning to get this to work just for the experiments on the OGBG dataset. It was not possible in the course of this thesis to tune the early switching to work as intended on every dataset.

An additional argument in defense of our approach is that it is not computationally heavy in theory and in practice our code could still be improved by wrapping our methods in the compiled and parallelized functions provided by the benchmark. This was difficult considering that our implementation has been constructed in the context of the submission template.

4.3 Limitations

Although the approach has some advantages and the results mostly improve the training performance, the interpretation of our results as well as the approach itself is subject to a few limitations. First, as mentioned before, the results shown in Chapter 3 are individual training runs rather than an average of trained ensembles. This is due to the limited research time and hardware. The results are therefore not supported by enough empirical data to be generalized. The motivation behind this research was exploratory in nature and should be seen as a proof of concept rather than making strong claims that could be supported by large amounts of data. As a proof of concept, however, the results are sufficient to demonstrate the advantages of the soft reset approach in combination with early switching methods.

Secondly, our proposed approach introduces hyperparameters for the soft reset and early switching which have to be tuned in practice. Especially for the early switching, hyperparameters are not obvious and depend on the specific data and model. Every approach that adds more hyperparameters introduces more complexity. The additional complexity of a problem must be outweighed by the benefits it brings. The approach therefore has to yield an improvement of the current algorithm to be useful. We argue that it does yield benefits in some cases.

Other than methodological limitations, there are some limitations implied by the theoretical approach itself due to the assumptions that are made. The most noticeable assumption is that the formula for the soft reset needs a target value. This exact target value of the corresponding workload target metric is defined for each workload. Therefore, in the context of this benchmark, we can always assume knowledge of an exact target score. In a more general use case, having an exact predefined target is a rare case. Another assumption for adjusting a learning rate schedule in this experiment was that the estimated runtime to reach the target setting is provided for each workload. Even with knowledge of the target setting runtime budget and target score, there is a restriction to the frequency of cycles in the current formulation of the soft reset. For a cycle length that is much less than a target-setting budget, the current formula has the issue that it is highly unlikely in every case for the model to reach the target within the cycle. This means that even a good performance will be punished to some degree because the target score is almost impossible to reach within the cycle. Then the difference between the target metric score at the time of the reset and the actual target and therefore the rescaling coefficient for a soft reset will be large in almost every case. Therefore, knowing the expected or desired performance at the time of reset is crucial in formulating the soft reset coefficient of parameters and momentum. Although knowing the training time needed to reach the target seems like a restrictive assumption, it is not absolutely necessary to know the exact target setting and runtime of a model. Possible workarounds are discussed below.

4.4 Future Research, Improvements, and Addressing of Limitations

Knowing the limitations of the soft reset approach, I will suggest some changes that can be implemented to try to improve the outcome of the method and tackle some of the problems stated above. I will also put some seemingly restrictive limitations into perspective to show that they are not so restrictive in practice.

A limitation of the thesis stated above is that it was not possible to gather enough data to make general statements about the soft reset with an early switching approach. This is tackled by gathering more training runs on a bigger variety of workloads.

Further, one characteristic of the AlgoPerf Benchmark is that the exact target of a workload is defined which is utilized in the soft reset of the model at the time of the reset. As already stated it is not common to have an exact predefined target in practice. Although this is mostly true, it should be said that in most cases where a deep learning model is used, you have an estimated target in mind. This is especially true for benchmarks but can also be true for other more general cases. In most cases outside the AlgoPerf Benchmark, the soft reset approach can be used because you can probably define an estimated target that aligns with your training goals and desired performance.

Another assumption made for the cyclical schedule is that we know the estimated target setting runtime of each workload. This is needed in this case to construct cycles that in theory can reach the target within the cycle. One of the problems with this is constructing a cyclical schedule with unknown runtime. Looking at the results of Zhai et al. [2022] where they introduced an infinite cyclical learning rate schedule. Another problem was the case where the length of a schedule cycle was significantly shorter than the estimated target setting runtime. A workaround for this is to construct a monotonically rising function of runtime or training steps to estimate the training progress. Then the refactor coefficient will compare the output of this time-dependent function with the current target metric score rather than the overall target when resetting. This solution has a clear disadvantage in that you have to have some knowledge of the problem and training behavior of the model to estimate this time-dependent function.

This cannot be done analytically and has to be done with either experience or preceding experiments which has to be seen as a clear limitation of the proposed approach.

In addition to the future improvements to the approach, I would like to mention another interesting observation of our results. The surprising result was the unexpectedly strong impact of the optimizer momentum on the reset and the following learning behavior in some cases. I hope the results motivate further research to unravel the optimizer momentum's role when reinitializing the model parameters, especially in the context of multimodal loss functions.

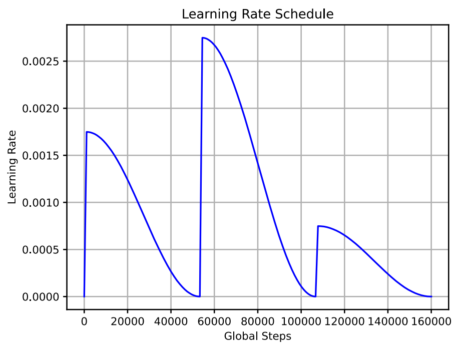
4.5 Takeaway of this Thesis

This work has shown that the approach of cyclical schedules with a soft reset in combination with early switching strategies has potential. There are still some important questions to be answered and there is room for improvement in this method, some of which have been already discussed above. We are confident to say that it is worth investing time in further research and optimization of this method.

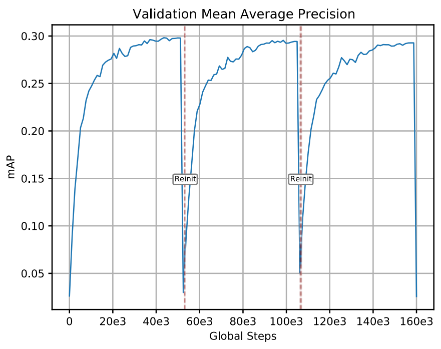
Appendix

A Figures

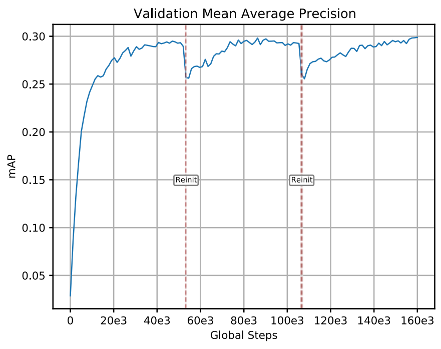
A.1 Figures



(a) Learning rate schedule with cosine decay learning rates as $\eta_1 \approx 0.00175$, $\eta_2 \approx 0.00275$, $\eta_3 \approx 0.00075$ for each cycle.



(b) Training run with schedule of Figure A.1a and full reset.



(c) Training run with schedule of Figure A.1a but no reset of optimizer momentum.

Figure A.1: Training runs with different learning rate schedules and reset strategies. One with a reset of model parameters and optimizer momentum and one with just the model parameters. Training on OGBG dataset with two Nvidia 2080Tis. η is given approximately because we adopted it from the original baseline as $\eta = 0.0017486387539278373$

B Lists

B.1 List of Figures

C Bibliography

George E. Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, Juhan Bae, Justin Gilmer, Abel L. Peirson, Bilal Khan, Rohan Anil, Mike Rabbat, Shankar Krishnan, Daniel Snider, Ehsan Amid, Kongtao Chen, Chris J. Maddison, Rakshith Vasudev, Michal Badura, Ankush Garg, and Peter Mattson. Benchmarking neural network training algorithms, 2023.

Payal Dhar. The carbon impact of artificial intelligence. *Nat. Mach. Intell.*, 2(8): 423–425, 2020.

OpenAI, Josh Achiam, and etal. Steven Adler. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Gemini Team, Rohan Anil, and et. al. Sebastian Borgeaud. Gemini: A family of highly capable multimodal models, 2024. URL https://arxiv.org/abs/2312.11805.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Timothy Dozat. Incorporating Nesterov Momentum into Adam. In *Proceedings of the 4th International Conference on Learning Representations*, pages 1–4.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms, 2023.

Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training, 2024.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Olivier Bousquet, Sylvain Gelly, Karol Kurach, Olivier Teytaud, and Damien Vincent. Critical hyper-parameters: No random, no cry, 2017.

Emmanuel Okewu, Philip Adewole, and Oladipupo Sennaike. Experimental comparison of stochastic optimizers in deep learning. In *Computational Science and Its Applications – ICCSA 2019*, pages 704–715, Cham, 2019. Springer International Publishing.

Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On empirical comparisons of optimizers for deep learning, 2020.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

Yurii Nesterov. A method for solving the convex programming problem with convergence rate o (1/k2). In *Dokl akad nauk Sssr*, volume 269, page 543, 1983.

Tijmen Tieleman. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4 (2):26, 2012.

Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, page 14–25, jul 2019. doi: 10.1145/3352020.3352024. URL https://doi.org/10.1145/3352020.3352024.

Leslie N. Smith. Cyclical learning rates for training neural networks, 2017.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation, 2018.

Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get m for free, 2017.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers, 2022.

Deposition:

I hereby declare that I am the sole author of this thesis and that I have not used any sources other than those listed in the bibliography and identified as references.

Tübingen, den 22.07.2024 ....

Georg Martin Tirpitz