Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

# Bachelor Thesis Cognitive Science

## Investigating Probabilistic Preconditioning on Artificial Neural Networks

Ludwig Bald

September 9, 2019

**Gutachter**

Prof. Dr. Philipp Hennig
(Methoden des Maschinellen Lernens)
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

**Betreuer**

Filip De Roos
(Methoden des Maschinellen Lernens)
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

**Bald, Ludwig:**
*Investigating Probabilistic Preconditioning on Artificial Neural Networks*
Bachelor Thesis Cognitive Science
Eberhard Karls Universität Tübingen
Thesis period: von-bis

# Abstract

Note to the reader: This thesis is unfinished and I put zero confidence in its correctness, quality of citations or completeness.

In machine learning, stochastic gradient descent is a widely used optimizsation algorithm, used to update the parameters of a model after a minibatch of data has been observed, in order to improve the model's predictions. It has been shown to converge much faster when the condition number (i.e. the ratio between the largest and the smallest eigenvalue) of ... is closer to 1. A preconditioner reduces the condition value. Tho goal of this thesis was to reimplement the algorithm as an easy-to-use-class in python and take part in the development of DeepOBS, by being able to give feedback as a naive user of the benchmarking suite's features. In this thesis I present my implementation of the probabilistic preconditioning algorithm proposed in [Roos and Hennig, 2019]. I use DeepOBS [Schneider et al., 2019] as a benchmarking toolbox, examining the effect of this kind of preconditioning on various optimizers and test problems. The results...

theabstract, citing!

# Zusammenfassung

Abstract auf Deutsch

# Acknowledgments

I would like to thank Aaron Bahde and Frank Schneider for developing the excellent benchmarking framework DeepOBS, which was essential to this thesis, and for always being quick to answer questions and fix bugs.

I would also like to thank my supervisor Filip De Roos, who was always available when I had questions about the algorithm published by him.

# Contents

# Chapter 1

# Introduction

What is this all about?

# Chapter 2

# Fundamentals and Related Work

## 2.1 Probability Basics

One important concept in probability theory is the Gaussian distribution. The real-valued *normal* or *Gaussian* distribution is defined as:

$$N(x|\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

It has the parameters *mean* $\mu$ and *variance* $\sigma^2$.

Extending this to a *multivariate* Gaussian distribution with $n$ dimensions, the definition needs to be adapted:

$$N(\vec{x}|\vec{\mu},\Sigma) = \frac{1}{(2\pi)^{n/2}} \frac{1}{det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})\right)$$

where $\vec{x}$ or simply $x$ is the $n$-dimensional vector of random variables. The parameters are similar to the one-dimensional case. The $n$-dimensional vector $\vec{\mu}$ or simply $\mu$ denotes the mean. The symmetric $n \times n$ matrix $\Sigma$ is the *covariance matrix*, where

$$\Sigma_{ij} = cov(x_i, x_j) = \Sigma_{ji}$$

For the diagonal entries it follows that:

$$\Sigma_{ii} = cov(x_i, x_i) = var(x_i)$$

If the variables $x_i$ are independent,$\Sigma$ is a diagonal matrix.

Matrix-valued Normal distributions can be written as vector-valued normal distributions by flattening the matrix to a vector.

cite the book

- Point estimates
- Bayes' Rule

## 2.2 Machine Learning

Machine Learning intro

- recent changes: big data, specialized hardware
- Machine Learning is used everywhere and really important
  - key factor in automatization.
- The process of machine learning
  - Gathering data
    - Splitting the data (Train, Test, validation)
  - Designing a model
    - Hyperparameter tuning
  - **Training the model**
  - Application

\subsection{Loss functions} How do we recognize the optimal solution, or even tell which of two solutions is better? Depending on the task that the model is meant to perform, we can define different so-called Loss Functions (also called "risk"). These are typically some sort of distance measure between the model's output and the desired, output. However, there are other values we care about, such as the model's ability to generalize, i.e. how well it performs on data it has never seen before. If a model performs well on the trainig data set, but fails to generalize, this is called overfitting and means the model is not very useful on new data. The model memorizes the data, but doesn't get the underlying structure. To combat overfitting, people add a regularization term to the Loss function, which penalizes large parameters, which are usually an indicator that the model is overfitting. \todo[inline]{formula} \todo[inline]{split data}

## 2.3 Optimization

In order to find the optimal values for the parameters of a model, different methods have been proposed. For low-dimensional optimization problems with a small number of parameters, it is often possible to find the optimal parametrization analytically. For high-dimensional optimization problems, the analytical solution often is computationally intractable. As a result, numerical optimization algorithms have emerged. These algorithms use the available limited information in order to iteratively approximate the optimal parametrization. The number of iterations needed until the algorithm converges varies based on the implicit prior assumptions about the model.

### 2.3.1 First-Order and Second-Order Methods

symbol

For many models, it is possible to obtain information about the derivatives of the loss function at a given point in parameter-space. Optimization algorithms can be grouped by the order of the highest-order derivative they use.

There are zeroth-order methods that only require the value of the loss function itself.

First-order methods use the first derivative, or the gradient. Adaptive Modifications of SGD like Adam and Momentum keep track of past gradients in order to take better steps. This implicitly assumes that past gradients contain information about future gradients, which is often the case. For example, if there is little change in the last gradients, we can assume that we are far away from the minimum, where gradients should converge to zero. Therefore, we are able to take larger steps.

Second-Order-Methods are methods that explicitly use the Hessian $B$ of the Loss function, or the second derivative. This means they can use the explicit representation of curvature to directly compute where the gradient will hit zero.

-RMSprop -Adam - Conjugate gradient? s.o.: -Newton: Needs access to the Hessian. -Quasi-Newton. We can keep track of an estimate of the Hessian along the way.

Some of these optimization processes have "hyperparameters" (different from the models "parameters" that we want to optimize).

### 2.3.2 Gradient Descent

If the gradient of the Loss function at a certain point in parameter space is known, we can use this information to update the parameters in a way that takes us closer to the solution. A popular family of algorithms is derived from SGD. SGD means: Just take a step in the direction of steepest gradient. Scale the step size by the steepness of the gradient. If the gradient is very steep, take a larger step. If it is small, take a smaller step, like a drunk student tumbling down a hill and ending up on a local minimum. SGD has only a single hyperparameter, the "learning rate" $\alpha$, which is multiplied with the step. Its optimal choice depends on the data, the model and is generally not obvious.

**Done:** formula: Update Rule

$$w_{i+1} = w_i - \alpha \cdot \nabla L(w_i)$$

### 2.3.3 Stochastic Gradient Descent

Traditional Gradient Descent uses the whole data set to compute the true gradient, which is computationally instractable for large datasets, the usual

case in machine learning. Instead, we use a variation called SGD. We compute only an estimate for the true gradient by "minibatching", using only a few, for example 64 data points in the forward pass. This greatly improves convergence speed, as the required computations are much easier to perform. However, especially for smaller batch sizes, this adds noise to the system, meaning that our parameter update step points only roughly in the direction of the steepest actual gradient. It has also been shown to improve generalization of the model. Many variants of SGD have been proposed, adding things like a momentum term or otherwise adapting the learning rate dynamically.

**Done:** formula: Noisy update rule

$$w_{i+1} = w_i - \alpha \cdot \nabla \hat{L}(w_i)$$

## 2.4 Preconditioning

Convergence speed depends on the *condition number* $\kappa = \frac{\lambda_n}{\lambda_1}$ of the Hessian.

The performance of SGD does not only depend on the choice of the learning rate, but also on the structure of the Loss landscape. The gradient of the loss function is a jacobian matrix of partial derivatives. Imagine an optimization proble with two parameters and a loss landscape that looks like an ellipse. If the random starting point is chosen to be towards the direction of the longer symmetry axis, the gradient will be rather flat and SGD will take a long time to converge. With the same logic, a circular, bowl-shaped loss landscape is optimal. If we start at the steep wall, SGD will converge quite quickly. If you know the explicit form of the Hessian matrix, you can see this problem by comparing its eigenvalues. A circular, bowl-shaped loss function will have only eigenvalues that are the same size. In the elliptical case, which has previously been reported to be the standard case in machine learning, at least one eigenvalue is much larger than the others. The mathematical measure for this phenomenon is called the "condition number" and is defined as the ratio between the largest and the smallest eigenvalues. SGD converges much faster when the condition number is closer to 1. The largest eigenvalue is sometimes referred to as the "spectral radius".

derivation of why the condition number is important, Limit on convergence

$$\frac{\kappa - 1}{\kappa + 1}$$

Preconditioning is a way to reduce the condition number. A preconditioner is a matrix that rescales the other matrix in such a way that the ratio of eigenvalues approaches 1. If the eigenvalues are known, this is quite easy. However, in deep learning, they are not known. In this thesis I present my implementation of an algorithm which aims to efficiently estimate the eigenvalues

and construct a Preconditioner, while taking into account the noise caused by minibatching.

The PROBLEM:

Up until now, there was no easy way to make use of preconditioning in a noisy setting, such as minibatched deep learning. I present an implementation of Filips Algorithm in an easy-to-use python class and demonstrate its strengths and weaknesses. - The convergence of optimizers depends on the condition number of the Hessian doneformula, which describes the whole loss landscape.

Image

Formula of how an optimal preconditioner works


This figure illustrates the condition number. It shows what SGD does on an illconditioned two-dimensional problem. And what preconditioning can do to change this problem.

## 2.5 Deep learning

### 2.5.1 Artificial Neural Networks

A popular machine learning paradigma is living through a resurgence: Artificial Neural Networks. The fundamental building block is the single neuron, whcih somewhat resembles a biological neuron. Its activation depends on the sum of the activation of its inputs. A neural network model is a sequence of neuron layers, connected with each other. There is an input layer, which is a direct mapping of the training data point. The input layer' activation are fed forward into the "hidden layers", which in turn feed their activations to the neurons of the output layer. Information about observed data is stored in the model's parameters, the weights and biases of each layer. As a mathematical object, a neuron is an activation function which depends on the sum of the weighted activations of the inputs. Often there is a bias, which is a static value added to the input activation.

nonlinear (affine) activation functions

architectures per task. ANN: CNN (visual/spatial), RNN (temporal)

### 2.5.2 Regularization vs. weight decay

We minimize a different parameter than what we actually care about. (Loss on the training set vs accuracy on the test set). This can lead to problems,

so there are some solutions. If the model's parameters are too large, that often means it is overfitting. in order to discourage this, a regularization term is sometimes added to the loss function. Usually this is the L2 norm of the parameter vector, but sometimes the L1 norm is also used. - But there are differences (see [Chaudhari et al., 2017])

Bayesian?

Formula

Maybe illustrative plot

### 2.5.3 Automatic Differentiation

For the most interesting class of optimizers, it is required to know the gradient at the current point in parameter space. In neural networks, this is achieved by Automatic Differentiation. Loss functions in neural networks have an interesting structure, in that they are a composition of all the layerwise activation functions. This structure leads to the algorithm of backpropagation: During training, the model (using its current value for its parameters) is fed with data from the data set. The loss function of this minibatch is computed. While this happens, the computations that take place are tracked and built into a graph structure. This is necessary, because the loss function in practice isn't defined in a closed-form way directly on the parameters, but as a composition of layerwise activation functions. This means that for every parameter tracked by the graph, we can infer its influence on the loss. The parameters need to be leaves of the tree, which means they don't themselves depend on something else. (provide example graph image here). The gradient in the direction of an individual leaf (partial derivative) is then computed by applying the chain rule, starting from the output of the loss function. This is called the backward pass.

Forward pass

Derivation

## 2.6 Benchmarking

using deepobs [Schneider et al., 2019] There are no standard established benchmarking protocols for new optimizers. It isn't even clear what measures to consider, or how they are to be measured. As a result, nobody knows which optimizers are actually good. And some bad optimizers will seem good. DeepOBS is a solution to this problem, standardizing a protocol, providing benchmarks and standard test problems.

- Description and examples of previous optimizer plots.
- Short overview of how deepobs handles stuff:
  - Test problems:
    - DeepOBS includes the most used standard datasets and a variety of neural network models to train. This ensures that everyone is evaluated on the same problem.
  - Tuning:

    - Many optimizers have hyperparameters that greatly affect the optimizer's performance
    - These need to be tuned by running many settings seperately, for example in a grid search. The actual deepobs protocol isn't ready yet.
  - DeepOBS generates commands for the grid search.
  - Running:
    - DeepOBS provides a standard way to run your optimizer, taking care of logging parameters and evaluating success measures.
  - Analyzing:
    - DeepOBS provides the analyzer class, which is able to automatically generate matplotlib plots showing the results of your runs.

## 2.7 Related Work

in which other ways has this problem been adressed? (What even is the problem?)

# Chapter 3

# Approach (Implementation)

## 3.1 Description of the algorithm

The exact inner workings of the algorithm are described in more detail in [Roos and Hennig, 2019]. Here I will give an overview of the algorithm's structure and steps: part 1: - Gather observations of the curvature in-place: part 2: - Estimating the Hessian and construct the preconditioner part 3: - Every step, re-scale the gradients by applying the preconditioner

### 3.1.1 Modifications of the algorithm

The implementation and theoretical work lead to the following proposed changes to the abstract algorithm.

#### Parameter groups

Mainly due to technical reasons (see ), support was added for parameter groups, but this also yields an interesting theoretical change. The algorithm already treated every parameter layer as an independent task for inversion (???), but estimated a global step size, which was the same for every parameter. With this modification, the algorithm is able to use larger step sizes for parameters that allow for it. In theory, this would allow for faster learning in the direction of those parameters. However, the time to reach total convergence relies on the slowest parameter, not on the fastest. The benefit of this approach in practice remanins to be tested.

reference chapter

#### Automatic Assessment of the Hessian's quality

In the original algorithm, the Hessian is arbitrarily re-estimated every epoch. Hessian re-estimation is needed, as demonstrated in the original paper, alpha

changes over time. The goal of this modification was to expose a measure of how useful/correct the estimated Hessian is, after observing a bit of data. A new estimation proces would be started if the current estimate was worse than a certain threshold. Multiple approaches were tried. A key point of trouble was the variance-less estimate of the Hessian. The original paper mentions that the Hessian is taken as the mean of a multivariate Gaussian distribution, but it fails to take this distribution's variance into account. Instead, it just assumes this Hessian is the optimal choice. - Comparing the predicted gradient to the actual observed gradient.

- Maybe: Automatic restart of the estimation process, once the old Hessian is determined to be out of date.

#### Focusing on the adaptive learning rate

Deep learning problems are very high-dimensional and usually feature many (10%) large eigenvalues. For performance reasons, the proposed algorithm however estimates only a low-rank preconditioner. When applied, the preconditioner therefore might not reduce the problem's condition number, or might reduce it only by a little bit. It appears likely that the added computational need of constructing and repeatedly applying the preconditioner does not come with a performance benefit. Therefore I propose abandoning the preconditioning part of the algorithm. In experiment x this assumption is tested.

ref

## 3.2 Documentation for the class Preconditioner

### 3.2.1 Overview

The class `Preconditioner` provides an easy way to use the probabilistic preconditioning algorithm proposed by \cite{roos2019active}. This is how to use it:

1. Get the source file from the repo and include it in your project
2. Initialize the preconditioner like any other optimizer. There are reasonable default values for the hyperparameters.
3. Depending on the version you're using, manually call `start_estimate()` at the beginning of each epoch.

In the next section there is more detailed documentation for the class, its attributes and functions.

### 3.2.2 Methods

- Public functions
  - Constructor
  - `start_estimate()`
  - `step()`
  - `get_log()`
  - (`maybe_start_estimate()`)
- Private functions
  - `_initialize_lists()`
  - `_init_the_optimizer()`
  - `_gather_curvature_information()`
  - `_estimate_prior()`
  - `_setup_estimated_hessian()`
  - `_apply_estimated_inverse()`
  - `_hessian_vector_product()`
  - `_update_estimated_hessian()`
  - `_create_low_rank()`
  - `_apply_preconditioner()`

### 3.2.3 Implementation Details

In the following, I will highlight and explain some key software design decisions I took while implementing and refactoring the algorithm provided by Filip. The main goals for the implementation were to make the algorithm as easy to use as possible for a standard usecase, while maintaining the flexibility I needed to research specific variations. The changes to the abstract algorithm are discussed in detail in \todo{ref} The simple, default usecase was a user trying to use the preconditioner as proposed in the original paper, to optimize a neural network model. In this case, the necessary changes to the user's existing code should be minimal, with hyperparameters set to reasonable default values. For development, it is important to understand the algorithm's structure and the code and be able to easily modify the algorithm without disturbing other parts.

The code in its final form is provided as a self-contained class. According to Python best practices \todo{cite}, all the internal functions that a user should not call are marked as hidden by having names beginning with an _. This is an implementation of the design pattern "Low Coupling". All functions have been given descriptive names. For example, in order to start the estimation process, the function `start_estimate()` needs to be called.

The class `Preconditioner` inherits from `torch.optim.Optimizer`. This means it follows all the conventions for how optimizers are expected to behave in pytorch. This makes it intuitive to use for the pytorch user. Features

that were added to support this include the `state` dict, which can be used to save and load the state of the optimizer in order to pause or continue training. Another supported feature are parameter groups. This allows to treat different parameters seperately, for example different layers of a neural network. Specifically, each Parameter group will be optimized with a seperate learning rate.

The class `Preconditioner` has a field containing an inner optimizer to be used for the actual parameter updates. In the original paper, only SGD was studied as an option, but this modification allows to use preconditioning and the adaptive learning rate estimation together with other optimizers. The Preconditioner is mostly active when estimating the Hessian. Once the estimate is complete, the class turns into a "decorator" for the provided inner optimizer class. Before the inner optimizer does its optimization step, the previously constructed preconditioner is applied to the parameters' gradient.

Logging data during training should not be a responsibility of the optimizer. In order to expose the data of interest, the class exhibits a method `get_log()`, which returns some values of interest. The user then takes care of logging and writing the data to a file outside the optimizer.

In order to change behavior of the class during development and research, the best way to make different versions is to make use of python's built-in subclassing. For example, I needed a version that skips applying the preconditioner every step, but behaves exactly the same in all other ways. This was solved by using a subclass of `Preconditioner`, which overwrites the function `_apply_preconditioner()` with an empty function.

Some bugs were also fixed. Previously, the algorithm skipped some mini-batches, which was a problem in the small `quadratic_deep` dataset.

## 3.3 Test Problems

For most experiments, the standard test problems included in DeepOBS were used, namely The test problems used were deep learning models. - mnist/fmnist - cifar10 - quadratic_deep

For some experiments, I used a seperate implementation of a cifar10 test problem.

## 3.4 DeepOBS baselines

Baselines are taken from XXX, well-tuned hyperparameters, but no schedule or such things. Some baselines come with variance, others (Nesterov) don't.

## 3.5 Technical details

The experiments were run on the TCML cluster at the University of Tübingen. A Singularity container was set up on Ubuntu 16.4 LTS with python 3.5, pytorch (version) and DeepOBS (see Appendix for Singulariy recipe). Computation was distributed over multiple GPU compute nodes using the workload manager Slurm. During development, I used git for distributed version control.

Other optimizers like Adam and SGD are taken from the current, but unpublished DeepOBS baselines.

# Chapter 4

# Experiment

## 4.1 Experiment 1: Preconditioning

In order to investigate the effect on training performance of the proposed preconditioning algorithm, two similar, but distinct versions were used. The full algorithm as described in section \todo{cite} as `PreconditionedSGD` and the same algorithm, which does everything in the same way except for applying the preconditioner at every SGD step, called `AdaptiveSGD`. This was achieved by subclassing and overwriting the `_apply_precenditioner()` method with an empty method. Those two optimizers were tested on the DeepOBS testproblems `cifar10_3c3d` and `fmnist_2c2d`, using the standard settings. The hyperparameters used for the experiment were the same for both optimizers: `num_observations = 10`, `prior_iterations = 5`, `est_rank = 2`, `optim_class = torch.SGD`, `lr = None`

The results are presented in figure \ref{fig:exp_preconditioning}, together with the DeepOBS baselines on the same problems. In almost every metric, both tested optimizers were outperformed by all widely used standards like simple SGD and Adam. Interesingly, the `fmnist_2c2d` problem seems prone to some kind of overfitting: All the standard optimizers perform start to increase test loss after reaching a local minimum. However, the test accuracy does not seem to suffer from this same problem. `PreconditionedSGD` and `AdaptiveSGD` don't show this behavior, but also don't reach convergence after 100 epochs, as they still continue to improve. Note however that the difference in accuracy is quite a bit smaller on `fmnist_2c2d` than on `cifar10_3c3d`. Another finding is that the variance of PreconditionedSGD and AdaptiveSGD is larger than the variance of the baseline optimizers.

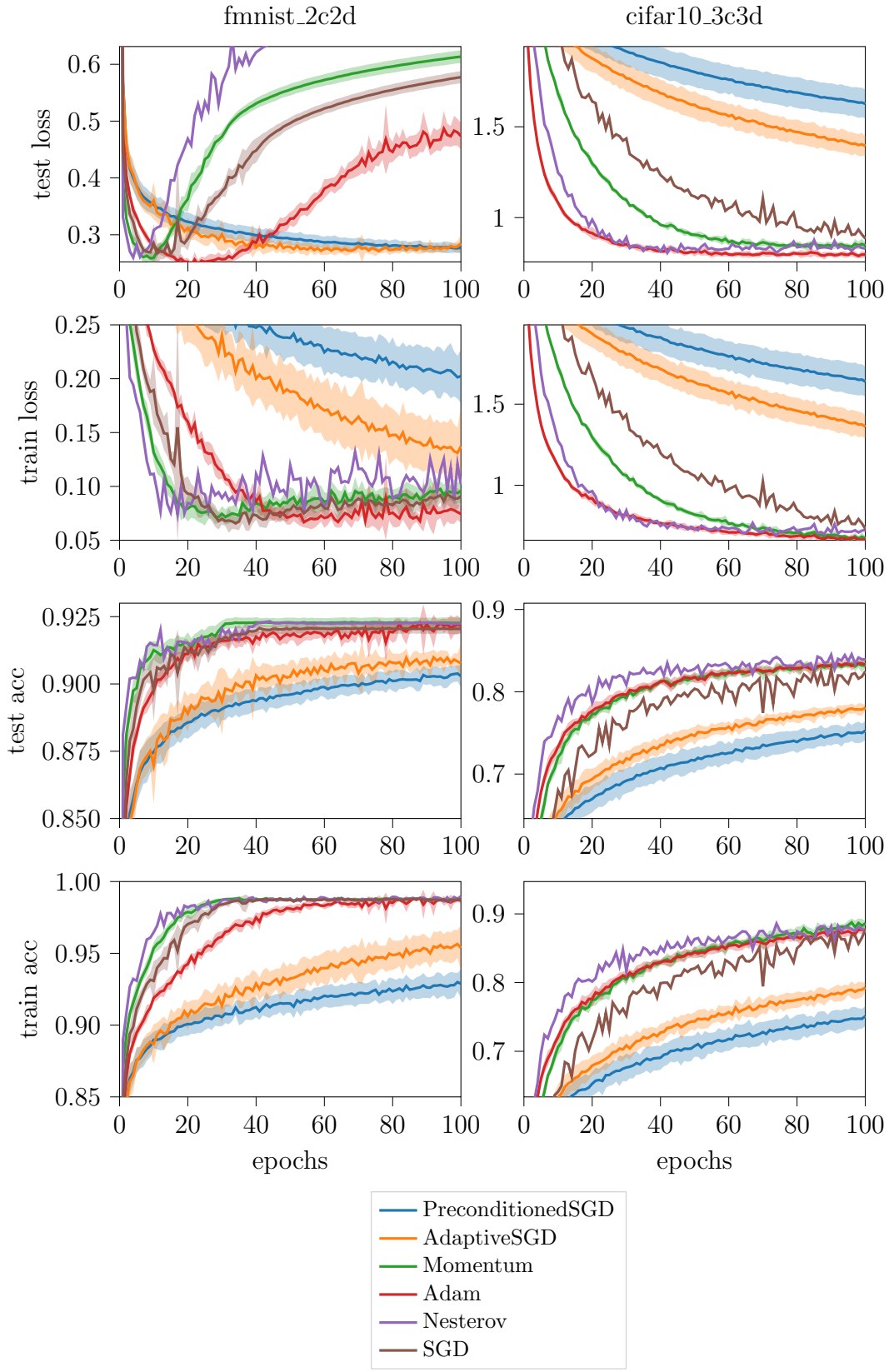


**Figure 4.1:** Both variants of the algorithm perform significantly worse than established alternatives. AdaptiveSGD is better than PreconditionedSGD. On the fmnist model, the baseline optimizers quickly reach a minimum, but then continue to overfit. AdaptiveSGD and PreconditionedSGD arrive at the mimimum much slower, but don't diverge.

## 4.2 Experiment 2: Computational Complexity

Wallclock time is an important metric for practitioners. To the practicioner it is important how much time, energy and money the training of a model costs. It is however difficult to interpret in experiment, because it depends on outside factors like hardware configuration, system load and other sources of statistical noise. DeepOBS provides a script that allows to compare the runtime of an optimizer against the runtime of SGD, run as a baseline on the same problem and the same hardware.

The three investigated optimizers were `PreconditionedSGD` and `AdaptiveSGD`, as defined in the previous experiment, and `OnlyAdaptiveSGD`, which only estimates the prior and constructs a step size. `OnlyAdaptiveSGD` is a more efficient version of `AdaptiveSGD`, which also constructs the estimate for the Hessian and the precondiotioner itself, but does not use this information. Once again, `OnlyAdaptiveSGD` is a subclass of `PreconditionedSGD`, but with some knocked-out functions.

The testproblem used was the default value set by DeepOBS, `mnist_mlp`. Every optimizer was run 5 times for 5 epochs each, on a GPU node of the TCML cluster.

The results are presented in figure 4.2. As expected, the more computations an optimizer does, the more time it takes to run. `PreconditionedSGD` was the slowest optimizer, taking an average of 2.39 ($SD = 0.36$) times as long as SGD, followed by `AdaptiveSGD`, which required 1.87 ($SD = 0.25$) times the runtime of SGD. `OnlyAdaptiveSGD` took about as much time as SGD, with a mean time of 1.00 ($SD = 0.07$) the time of SGD. A standard deviation for SGD is not given in the current version DeepOBS.

## 4.3 Experiment 3: Initialization

[Roos and Hennig, 2019] report that their implementation of the algorithm includes a hyperparameter for the manual first-epoch learning rate, because the learning rate constructed by the algorithm would be so high that the model diverges in the first epochs. Using the proposed implementation, I was not able to replicate this finding. On all tested problems included in DeepOBS, using the constructed learning rate for the first epoch lead the algorithm to eventual convergence. The authors used a model very similar to the one used in DeepOBS' testproblem `cifar10_3c3d`. Upon closer inspection, the main difference was that DeepOBS initializes the weights explicitly, while the authors used the built-in pytorch initialization functions. In order to investigate the effect of initialization, DeepOBS was modied to include the original net,

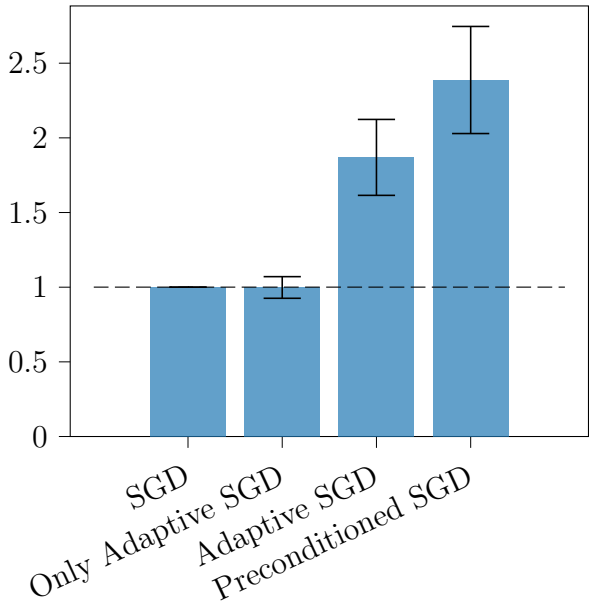


**Figure 4.2:** Wallclock time per epoch in relation to SGD, as tested on the DeepOBS testproblem mnist_mlp . The original algorithm is the slowest, followed by the version that does everything but apply the preconditioner. Finally, the version that only computes the prior and uses the adaptive step size is as fast as the baseline SGD.

either with explicit DeepOBS or implicit pytorch initialization. This model was then trained for both variations, with a batch size of 64 and 128, for five epochs, for each of 10 random seed values. As is presented in figure 4.3, while the initialization method certainly plays a big role in training performance, neither initialization method and neither batch size lead to a diverging run.

## 4.4 Experiment 4: Learning Rate Sensitivity

Maybe multiple problems: quadratic_deep, fmnist_2c2d, cifar10_3c3d, maybe more

After establishing that it's safe to use the automatically constructed learning rate, it is still unclear whether there is an effect on training success of manually setting a first-epoch learning rate versus using the constructed learning rate. A DeepOBS-aided grid search with 10 evaluations on a logarithmic grid between $10^-5$ and $10^2$ on the `fmnist_2c2d` testproblem.

The results are shown in 4.4, together with the baseline SGD reference. Like SGD, the model diverges in the first epoch if the learning rate is set over a certain threshold. Below this threshold, the first-epoch learning rate has next to no effect on the model's final accuracy after training for 100 epochs. For SGD, which uses the same learning rate for all 100 epochs, there is a

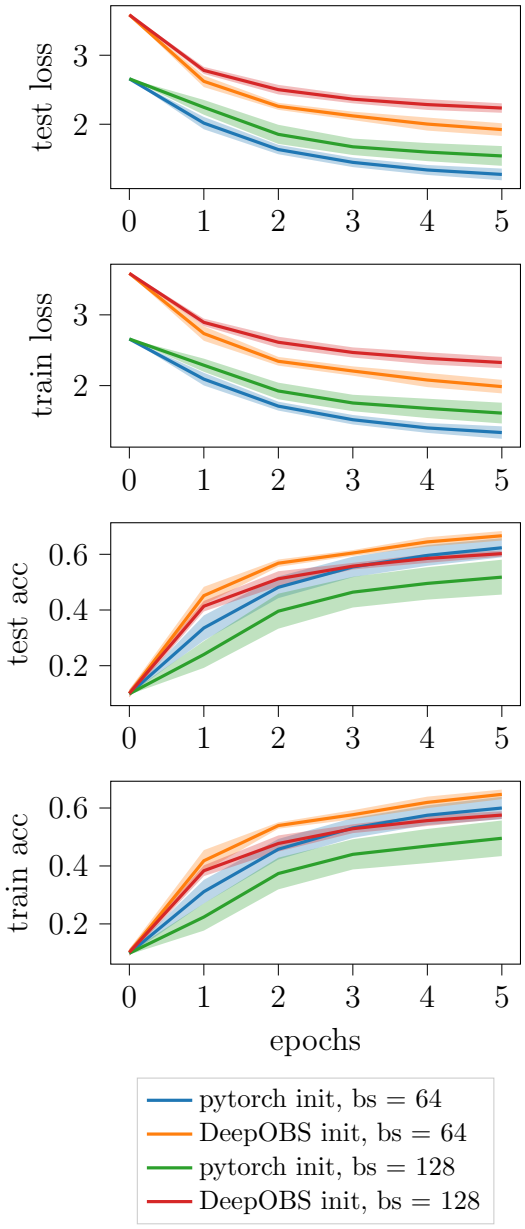


**Figure 4.3:** PreconditionedSGD on a cifar10 net. For both batch size 64 and 128, the preconditioner converges. The initialization method has an effect, but the optimizer handles them fine (on this testproblem.)

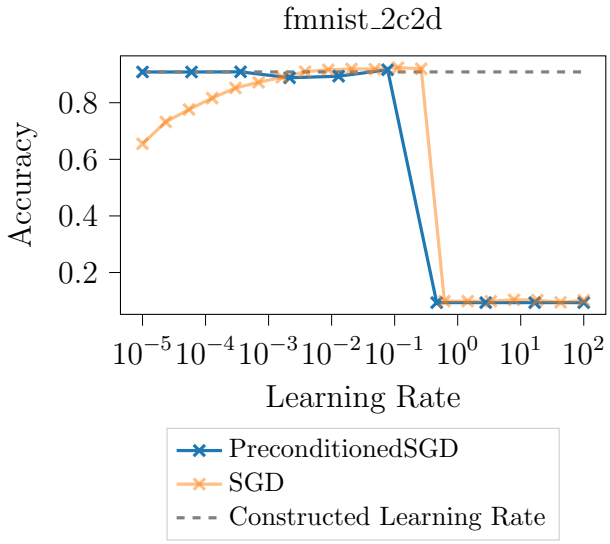


**Figure 4.4:** The originally proposed algorithm has an optional hyperparameter "learning rate", which is used as SGD's learning rate in the first epoch. Below a certain model-specific threshold, it performs just as well as the constructed learning rate. Over that threshold, it diverges in the first epoch.

significant dropoff in training success for smaler learning rates. Both SGD and `PreconditionedSGD` with an optimal choice for learning rate achieve a similar accuracy to `PreconditionedSGD` using the automatically constructed learning rate.

## 4.5 Discussion

The experiments presented in the previous sections show that the algorithm proposed in [Roos and Hennig, 2019] is not generally useful for application in deep learning. In Experiment 1 it is outperformed in nearly every measure of training success by established optimizers. This stands in direct contrast to the findings of the original authors, who report it comparing favourably against SGD in particular. However, the used testing protocol is not thoroughly explained. Test problem, batch size and SGD learning rates were set manually and without giving a reason, which makes the results hard to interpret. In a similar way, the hyperparameters were not tuned in a systematic way for either optimizer.

Given this caveat, the comparatively poor performance of both AdaptiveSGD and PreconditionedSGD compared to the DeepOBS baselines can be explained. While the DeepOBS baselines were exhaustively toned for accuracy, the proposed algorithms were not tuned to the testproblem. While tuning their learning rate is not necessary, other hyperparameters like the number of

steps used for the Hessian estimate rely on guessing and do impact performance. The algorithm uses Bayesian methods to estimate the Hessian, but does not leverage the knowledge about the uncertainty of the estimates.

In order to keep computational overhead of applying the preconditioner manageable, the preconditioner matrix was reduced to rank two. This however does not have a large effect on the condition number. In Deep Learning, cite generally there are many large eigenvalues that would all need to be reduced by a preconditioner in order to see a notable improvement on the condition number. Given that applying the preconditioner actually leads to smaller learning success than ignoring it, applying the rank-2-approximation preconditioner seems to be counterproductive. Because gradients can vary dramatically, the preconditioner is noisy. The algorithm also always re-starts the whole estimation process, while ignoring curvature data gathered in previous iterations. As a consequence the estimate does not become more accurate over time. This is in accordance with the authors' assesment that the good performance is mainly caused by the adaptive step size.

Another possible reason that SGD outperforms the preconditioner is that while estimating the Hessian, the Preconditioner does not use the data for actual parameter updates. The effect of this depends on the number of minibatches.

In experiment 2, the performance penalty was investigated. The authors report that the cost of building the rank 2 approximation accounted for 2-5% of the total computational cost per epoch, using a batch size of 32 on the training subset of CIFAR-10. This number does not tell the whole story. This thesis adds to this finding the performance penalty of estimating the Hessian and applying the preconditioner. As DeepOBS splits the data into three subsets (validation for hyperparameter tuning, training and testing), the training set is smaller. Using the DeepOBS standard batch size of 128, there are fewer minibatches per epoch, which means the estimation of the Hessian on the same number of steps uses a larger proportion of total training time. In this setting, using the preconditioner incurs a computational cost of more than double that of SGD. While the algorithm can be further optimized for performance, for example by moving all computations on the GPU, that's a big difference in computational cost. Calculating only the adaptive step size might be a better idea, as that incurs no significant performance penalty while achieving better accuracies.

Experiment 3 and 4 show that it can be a good idea to rely on the algorithm constructing a learning rate. While the achieved accuracies are a bit worse than those of well-tuned SGD, the trade-off of not having to do the expensive tuning process can be worth the tradeoff. The runs in experiment 3 were only done for a small number of test problem/hyperparameter combinations. There is no guarantee that the found stability translates to other kinds of problems, so it is

probably useful to keep the option of setting an initial learning rate. Generally, the experiments in this thesis were mostly performed on convolutional nets, and it is unclear which properties would translate to other architectures.

The usability goals of the implementation were met, as the experiments required no modifications to the final optimizer class to be run and integrate nicely with DeepOBS.

## 4.6 Further research/development

This thesis can not exhaustively cover all aspects of the presented optimizer implementation. There are open questions the experiments didn't answer and several improvements to be made to the implementation and the algorithm itself.

The relationship between the Preconditioner's remaining hyperparameters and accuracy and computational overhead remains unclear. How much better does the estimate get when using an additional data point? The Hessian is taken as the mean of a Gaussian distribution, so the variance of this distribution could be taken as a measure for uncertainty. The algorithm could keep using new data points until the certainty of the estimate doesn't improve anymore and the remaining variance can mostly be explained by minibatch sampling noise.

A related open question is when to best restart the estimation process. This could be at the start of every epoch, as throughout this thesis, after some fixed number of minibatches or dynamically, once the estimate is determined to be a bad fit at the current point in parameter space. Estimating the likelihood of the Hessian given the observed gradients could also be achieved using the variance of the Hessian estimate, which isn't calculated in the current version.

It is also unclear whether parameter groups are only a convenient feature for practicioners or whether using them does actually impact training success.

In order to make general statements about the robustness and performance of the algorithm, it would need to be either further examined analytically or run on many more, different kinds of model architectures.

# Chapter 5

# Conclusion

In this thesis, I present an implementation of the probabilistic preconditioning algorithm proposed in [Roos and Hennig, 2019]. Using the optimizer benchmarking framework DeepOBS, I show that it performs worse than well-tuned standard optimizers like SGD and Adam on convolutional neural networks. I also show that using the preconditioner incurs a large performance penalty. I propose a middle ground solution, which I call `AdaptiveSGD` throughout this thesis. It uses information about the Hessian in order to propose a good step size, but does not construct a preconditioner.

# Appendix A

# An appendix

Here you can insert the appendices of your thesis.

# Bibliography

[Chaudhari et al., 2017] Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J. T., Sagun, L., and Zecchina, R. (2017). Entropy-sgd: Biasing gradient descent into wide valleys. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[Roos and Hennig, 2019] Roos, F. and Hennig, P. (2019). Active probabilistic inference on matrices for pre-conditioning in stochastic optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1448–1457.

[Schneider et al., 2019] Schneider, F., Balles, L., and Hennig, P. (2019). Deepobs: A deep learning optimizer benchmark suite. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

# Selbstständigkeitserklärung

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

---

Tübingen, 9. September 2019 Ludwig Bald