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

October 7, 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

In recent years, machine learning and more specifically Deep Learning has been becoming more and more relevant. Both in research and in application, it is ubiquitous. It has lead to anything from a better understanding of the brain's structure ([Brown and Hamarneh, 2016]) to major advancements in self-driving cars.

Even though the practical use of the current state of machine learning is unquestionable, there is a lot of opportunity for research and further improvement of the inner workings. One of the areas of interest is the study of optimization algorithms. They have a large effect on the time it takes to find an optimal parametrization while training a machine learning model. Improvements in this area lead to reduced cost, reduced need for data and reduced energy consumption.

The simplest optimizer is Stochastic Gradient Descent, which is a first-order-method and needs access to the gradient of the error function of the model. Second-order methods make use of the Hessian of the error function, which is the second derivation and contains information about curvature. The additional curvature information allows these methods to take much larger and more precise steps, but they require more computational effort which is infeasible for high-dimensional problems like Deep Learning models.

Two recent publications mainly drive this thesis: [Roos and Hennig, 2019] describes a preconditioning algorithm that estimates the Hessian and applies its inverse to the parameters, which is a way of including second-order information. This changes the parameter landscape so that standard first-order optimizers can converge faster to the optimum. The paper shows that it compares favourably in performance over non-preconditioned, standard SGD. The paper also shows results for a single Deep Learning experiment. The second paper, [Schneider et al., 2019], presents the benchmarking suite "DeepOBS" which makes it easier to compare new optimization algorithms in an unbiased way against established baselines.

This thesis makes multiple key contributions. The first is a reimplementation of the preconditioning algorithm, which increases usability and adds functionality, like the possibility to easily switch out the optimizer that takes over after preconditioning. The second contribution is the comparison of the preconditioning algorithm against established benchmarks measured using DeepOBS, which includes both an attempt to reproduce the results, and further investigation of applying the optimizer to Deep Learning models. In parallel with this thesis, a pyTorch version of DeepOBS was being developed by Aaron Bahde. This development is described in detail in [Bahde, 2019]. Another contribution was to provide user feedback to the development team of DeepOBS. A selection of feedback is replicated towards the end of this thesis.

Chapter 2

Fundamentals and Related Work

2.1 Probability Basics

This thesis makes use of some basic concepts of probability theory. Most notably, vector-valued normal distributions, Bayes' Rule and point estimates.

The following concepts are adapted from [Bishop, 2006].

One of the central concepts 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* μ and *variance* σ^2. The mean represents the maximum-a-posteriori estimate for x; it is the maximum of $N(x|\mu,\sigma^2)$

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 μ denotes the mean. The symmetric $n \times n$ matrix Σ 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,Σ is a diagonal matrix.

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

Bayes' Rule tells us what happens to the probability of an event X after observing new evidence E, given a prior probability $P(X)$ and probabilities $P(E)$ and $P(E|X)$. In its basic form it looks like this:

$$P(X|E) = \frac{P(X) \cdot P(E|X)}{P(E)}$$

Bayes' rule can also be applied to vector-valued normal distributions, which leads to vector valued normal posterior distributions.

2.2 Machine Learning

Machine Learning algorithms are everywhere. They control which products we buy, which music we discover and they enable recent advances in industrial automation and autonomous driving. Specialist AI agents routinely outperform the best human players in the hardest games, like Go ([Gibney, 2016]) or Starcraft II ([Vinyals et al., 2019]) This recent rise of machine learning is caused by a number of factors. Enabled by the internet, almost every interaction with a technological system is tracked and recorded. Big Data is available to the masses through Open Data initiatives. Learning algorithms have been around for a few decades, but recent advances in hardware meant it became economically feasable to use them on a large scale.

The general process of machine learning can be summarized in multiple steps:

1. Gathering Data: The gathered data needs to be prepared and split into multiple sets, one for training, a test set for evaluation and a third set for validation of hyperparameter tuning.
2. Chosing the model: Depends on the problem. A possible choice for image classification tasks is a Convolutional Deep Neural Network, for example.
3. Training the model: The model is shown the data, and its parameters are adjusted towards the optimal point. The process is often run multiple times with different optimizer hyperparameters to find the best setting.
4. Application: The model is applied to new data to see how it performs. If it does well, it can be deployed on the designed task.

2.2.1 Loss functions

In order to find an optimal solution, there needs to be a measure of "goodness" of a given parametrization. The loss function usually is a sort of distance

measure between the model's outputs and the true labels of the training data.

$$L : \mathbb{R}^n \to \mathbb{R}$$

One example for a loss function for classification problems, which is used for the experiments in this thesis, is Cross Entropy Loss, where w are the model's parameters, y is the true label, $\hat{y}_w$ is the model's output.

$$L(w) = -\sum_{1}^{K}(y \cdot \log(\hat{y}_w(x)))$$

In order ro prevent the model from overfitting, often a regularization term is added:

$$L(w) = -\sum_{1}^{K}(y \cdot \log(\hat{y}_w(x))) + \lambda\|w\|^2$$

Here λ is a parameter that determines the strength of the regularization. This will penalize large weights during training, which biases the model towards parametrizations that perform better on unseen data. In most cases it is equivalent to weight decay, or decreasing weights after each optimization step. See [Chaudhari et al., 2017] for futher investigations on the differences between these two.

It's worth noting that the choice of loss function and the regularization term have a significant impact on the outcome and convergence speed of training. Ultimately, other measures like classification accuracy might be more important during application.

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. Numerical algorithms are a solution to this problem. They use the available local information in order to iteratively approximate the globally optimal parametrization. The number of iterations needed until the algorithm converges varies based on the implicit prior assumptions about the model. Some of these optimization processes expose *hyperparameters*, which influence the performance of the algorithm. These are different from the model's parameters, which we want to optimize.

For Deep Learning models, it is possible to obtain information about the derivatives $\nabla^n L$ of the loss function L 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 like Gradient Descent use the first derivative, or the gradient. They work under the assumption that the gradient points in the direction of the global minimum and approaches zero while approaching the minimum.

2.3.1 Gradient Descent

If the gradient of the Loss function at a certain point in parameter space is known, this information can be used to update the parameters in the direction towards towards the solution. A widely used family of optimization algorithms is derived from Gradient Descent. Gradient Descent 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.

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

Gradient Descent has only a single hyperparameter, the "learning rate" α, which is multiplied with the step. Its optimal choice depends on the data, the model and is generally not obvious.

2.3.2 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. There is a variation called Stochastic Gradient Descent(SGD). Instead of the true loss, which is the model's performance on the whole training data, it computes only an estimate for the true loss by using a subset or *Minibatch* of the training data. A common choice is between 32 and 256 data items per minibatch. 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.

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

Many variants of SGD have been proposed, for example by adding a momentum term or otherwise adapting the learning rate dynamically.

2.3.3 Second-Order Methods

First-order methods scale their step length by the gradient of the Loss function. This means that they get stuck on flat plateaus, because the gradient is very small. Adaptive Modifications of SGD like `Adam` and `Momentum` keep track of previously seen 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, one can assume that the optimizer is far away from the minimum, where gradients should converge to zero. Therefore, it is now 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.

everything

-RMSprop - 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.

2.4 Preconditioning

SGD takes large steps if the gradient of the loss landscape is steep, and it takes small steps if the gradient of the loss landscape is flat. The underlying assumption is that as the gradient gets flatter (approaches zero), the algorithm approaches the minimum of the loss function. Taking large steps close to the minimum most likely means taking a step away from the minimum. However, SGD is not aware of the scale of the loss landscape. Some problems are generally flatter and have low general curvature than others that might curve more. On these generally flat problems, flat gradients can be observed quite far from the minimum and mandate larger steps. Recalling the update function of SGD, the learning rate parameter α is a scaling factor for the gradient, which can correct for non-optimal general steepness:

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

Tuning the learning rate amounts to finding a global scaling factor for the gradient.

Often, machine learning problems are scaled differently in different directions. In one direction with low curvature, the problem might be quite flat, while in another direction with high curvature, gradients are generally steeper. This means that the optimal step length depends not only on the steepness of the gradient, but also its direction (In the low curvature direction, longer

steps are warranted than in the high curvature direction). This structure is impossible to fully capture and correct for with a simple scalar learning rate. Chosing the appropriate small learning rate for the high curvature direction means the algorithm learns very slowly in the low curvature direction. Chosing anything larger than that runs a risk of exploding gradients in the high curvature direction, as the steps are too large. Using a matrix P (the *Preconditioner*) to scale every entry of the gradient matrix individually, we get the following update rule for Preconditioned SGD:

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

A perfect preconditioner would be the inverse of the true Hessian of the loss function, which exactly captures the curvature of every parameter. If it were available, the true Hessian could however be used for second-order methods, eliminating the need for preconditioning. Computing the true Hessian is very hard and expensive in the high-dimensional and noisy deep learning setting, so various approximations have been proposed.

cite

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

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

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

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, which 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. Activation functions are not necessarily linear. For example, the Rectified Linear Unit is widely used:

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

Different architectures of neural networks have emerged, which are useful for different tasks and types of input data. For example, fully connected Convolutional Neural Networks (CNN) are good at encoding visual or spatial information, while Recurrent Neural Networks have an internal state and are good at encoding information over time.

2.5.2 Automatic Differentiation

For first- and higher-order optimizers, it is required to know the gradient at the current point in parameter space. In neural networks, this is achieved by backpropagation. 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

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

As described in section 2.4, a perfect preconditioner would be the inverse of the Hessian of the loss function, wich is not directly accessible and expensive to compute. The algorithm tested in this thesis is described in detail in [Roos and Hennig, 2019]. This is a high-level overview on the algorithm's structure. The algorithm is made up of four main parts, which will be tested seperately throughout this thesis. It restarts by default every epoch.

Do I need to go into more detail here?

1. Gather observations in-place to construct a multivariate Gaussian distribution as a prior estimate for the Hessian. Calculate alpha and initialize SGD with alpha as its learning rate.
2. Gather more observations and calculate the posterior multivariate Gaussian distribution using Bayes' rule.
3. Take the mean of this distribution as the MAP-estimate. Invert it.
4. For every following minibatch, rescale the gradients by applying the preconditioner and perform an update step of SGD.

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 section 3.2.3), 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 while inverting the Hessian. It however estimated a global step size, which was the

same for every parameter. With this modification, the user can specify which parameters should be grouped together and get a common learning rate.

As described in section 2.4, the learning rate corrects for the scale of the overall curvature of the loss landscape. However, depending on the problem, different parameters require different scale factors. With this modification, scale factors are neither shared for all factors (as in SGD) nor down to an individual parameter precision (Preconditioning). Rather than going for the safest choice, the modified algorithm is able to use larger step sizes if all parameters in a group allow for it. This accelerates learning in the direction of those parameters. The benefit of this approach in practice was not tested.

Automatic Assessment of the Hessian's quality

The algorithm estimates a low-rank estimate for the Hessian. After a number of optimization steps, the surrounding loss landscape has changed and the estimate for the Hessian no longer is useful. It is not clear when this re-estimation should optimally happen. An answer needs to balance the added computational cost of re-estimating the Hessian with the performance benefit that comes from having a better-fitting Hessian. Using the original algorithm, the estimation process is restarted every epoch. The authors don't justify this practical choice. The `Preconditioner` class includes a method `maybe_start_estimate()` which is called before every step of the optimizer and could be used to dynamically assess whether the Hessian is out of date.

Assessment of the Hessian's quality is made especially difficult by sampling noise. A possible approach is to test prediction power of the Hessian. Given the last observed gradient, how well does the Hessian predict the gradient for the next minibatch? Standard statistical techniques like a χ^2 homogenity test could be applied, varying the confidence level to find the balance mentioned before. If the algorithm kept track of a measure of uncertainty for the Hessian and in turn its predictions, Bayesian methods would be possible to apply.

Focusing on the adaptive learning rate

Deep learning problems are very high-dimensional. [Chaudhari et al., 2017] report that around 10% of eigenvalues are large, while 90% are close to zero. A proper preconditioner would need to rescale around 10% of parameters. This means the proper preconditioner's rank would still be very high. Computations would be much more expensive and would probably eat up the performance benefit that comes with preconditioning. In experiment 1 it is tested whether applying the low-rank approximate preconditioner has an effect on convergence speed.

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 [Roos and Hennig, 2019]. It's written in python and made to work with pyTorch. 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
  - `Preconditioner(params, est_rank=2, num_observations=5, prior_iterations=10, weight_decay=0, lr=None, optim_class=torch.optim.SGD, **optim_hyperparams)`
    - `params`: the model's parameters that will be optimized
    - `est_rank` : an Integer, the rank of the preconditioner
    - `num_observations` : an Integer, the number of posterior updates to the estimated Hessian
    - `prior_iterations`: an Integer, the number of iterations to construct the prior
    - `weight_decay` : a float between 0 and 1, the amount of weight decay
    - `lr`: The learning rate. If specified, it will be passed to the inner optimizer as `lr` in the first epoch.
    - `optim_class` : The `torch.optim.optimizer` class to be used as inner optimizer.
    - `**optim_hyperparameters` : any other hyperparameters to be used for the inner optimizer
  - `start_estimate()`
    - (Re)starts the process of estimating the Hessian.
  - `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. The starting point was the original code from [Roos and Hennig, 2019]

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 conceptual modifications to the algorithm are discussed in section **??**. 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 python class**, built for the deep-learning framework **pyTorch**. According to Python conventions, 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* and tells the user a clear interface. All functions have 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` wraps an **inner optimizer** which is responsible for the actual parameter updates. In the original paper and in this thesis, only SGD is studied as an option, but this modification allows to use preconditioning and the adaptive learning rate estimation together with other optimizers which are implemented in a pytorch optimizer class. The Preconditioner class is responsible for 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 is a global task and 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. It can be overwritten if the user wants to see other data. The user then takes care of further processing and writing the data to a file.

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, in the experiments there is a version `AdaptiveSGD` that skips applying the preconditioner every step, but behaves exactly the same as `Preconditioner` in all other ways. `AdaptiveSGD` is a subclass of `Preconditioner` that overwrites the function `_apply_preconditioner()` with an empty function.

Some minor bugs were also fixed. Previously, the algorithm skipped some minibatches during the estimation phase. However, there remains room for improvement. The class assumes the user to input sensible values and won't complain otherwise. For some variables, helpful unit checks are in place. This is fine in a research environment, but not for deployment.

3.3 Test Problems

The optimizers were tested on some of the testproblems that come with DeepOBS and had a stable pytorch implementation at the time of testing. DeepOBS specifies the data set, the model architecture and initialization, the loss function and regularization. It also proposes a standard batch size that is shared by all baselines.

Before training, the data set is split into a validation set for hyperparameter tuning, a training set and a holdout test set.

The DeepOBS testproblem `fmnist_2c2d` uses the Fashion-MNIST data set ([Xiao et al., 2017]), which consists of 60000 labeled greyscale 32×32 pixel images of ten classes of fashion items. The model used consists of two convolutional layers, each followed by max-pooling, a fully connected layer with ReLU activation functions, and a 10-unit softmax output layer. The initial

weights are drawn from a truncated normal distribution with $\sigma = 0.05$ and initial biases are set to 0.05. As a loss function, cross entropy loss is used, without any regularization term. The standard batch size is 128.

The DeepOBS testproblem `cifar10_3c3d` uses the CIFAR10 data set ([Krizhevsky et al., 2014]), which consists of 60000 labeled RGB 32×32 pixel images of ten classes of objects. The model used consists of three convolutional layers with ReLU activation functions, each followed by max-pooling, two fully connected layers with respectively 512 and 256 units and ReLU activation functions, and a 10-unit softmax output layer. The initial weights are initialized using Xavier initialization and initial biases are set to 0.0. As a loss function, cross entropy loss is used, with L2 regularization on the weights (but not the biases) with a factor of 0.002. The standard batch size is 128.

For the experiment investigating computational complexity, DeepOBS uses the built-in testproblem `mnist_mlp`, which is a multi-layer perceptron that consists of four layers with 1000, 500, 100 and 10 unis respectively, using ReLU activation on the first three layers and softmax on the output layer. Initial weights are drawn from a truncated normal distribution ($\sigma = 0.03$)and the initial biases are set to 0.0. As a loss function, cross entropy loss is used, without any regularization term. The standard batch size is 128.

In the experiment investigating different initialization methods, the goal was to replicate the findings from [Roos and Hennig, 2019], so the exact same model was used. It is very similar to the one that comes with DeepOBS. There are three convolutional layers, each followed by ReLU and maxpooling. After that, there are three fully-connected layers preceded by ReLUs. As a loss function, cross-entropy loss is used, with regularization as in `cifar10_3c3d`. Batch size and initialization methods are varied.

3.4 DeepOBS baselines

When using a standardized testing procedure like the one provided by DeepOBS, it is not necessary to run every optimizer, as reproducibility of the results is guaranteed across different hardware. This saves a lot of work and energy that every single researcher otherwise would have needed to spend. DeepOBS comes with results from many established and well-studied optimizers as a baseline, and it is easy to compare a new optimizer against them. Their hyperparameters are tuned for each testproblem, but some more complicated widely used techniques like hyperparameter schedules are not available.

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. Every job had access to 4 CPU cores (Intel XEON CPU E5-2650 v4), 3GB of RAM, and 1 GPU (GeForce GTX 1080 Ti).

During development, I used git on github for distributed version control.

Chapter 4

Experiments

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. Firstly, `PreconditionedSGD`, the full algorithm as previously described uses SGD to do the actual parameter updates. Secondly, `AdaptiveSGD` also computes the preconditioner and constructs a learning rate, but does not apply the preconditioner. It is a subclass of `Preconditioner` that has the `_apply_precenditioner()` method replaced with an empty method. Those two optimizers were tested on the DeepOBS testproblems `cifar10_3c3d` and `fmnist_2c2d`, using the standard settings as provided by DeepOBS. 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`4.1 Another finding is that the variance of `PreconditionedSGD` and `AdaptiveSGD` is larger than the variance of the baseline optimizers.

On every metric, `PreconditionedSGD` is significantly outperformed by the variant without preconditioning, `AdaptiveSGD`.

Investigating further, figure 4.2 shows the performance of `PreconditionedTunedSGD`, which calculates the preconditioner, but does not apply the constructed learning rate. Instead, it uses the exact same learning rate as the well-tuned baseline SGD. Comparing it to SGD isolates the effectiveness of preconditioning without an adaptive learning rate. This version performs comparably to SGD and outperforms the two adaptive versions. On the `cifar10_3c3d` testproblem, it even outperforms SGD.

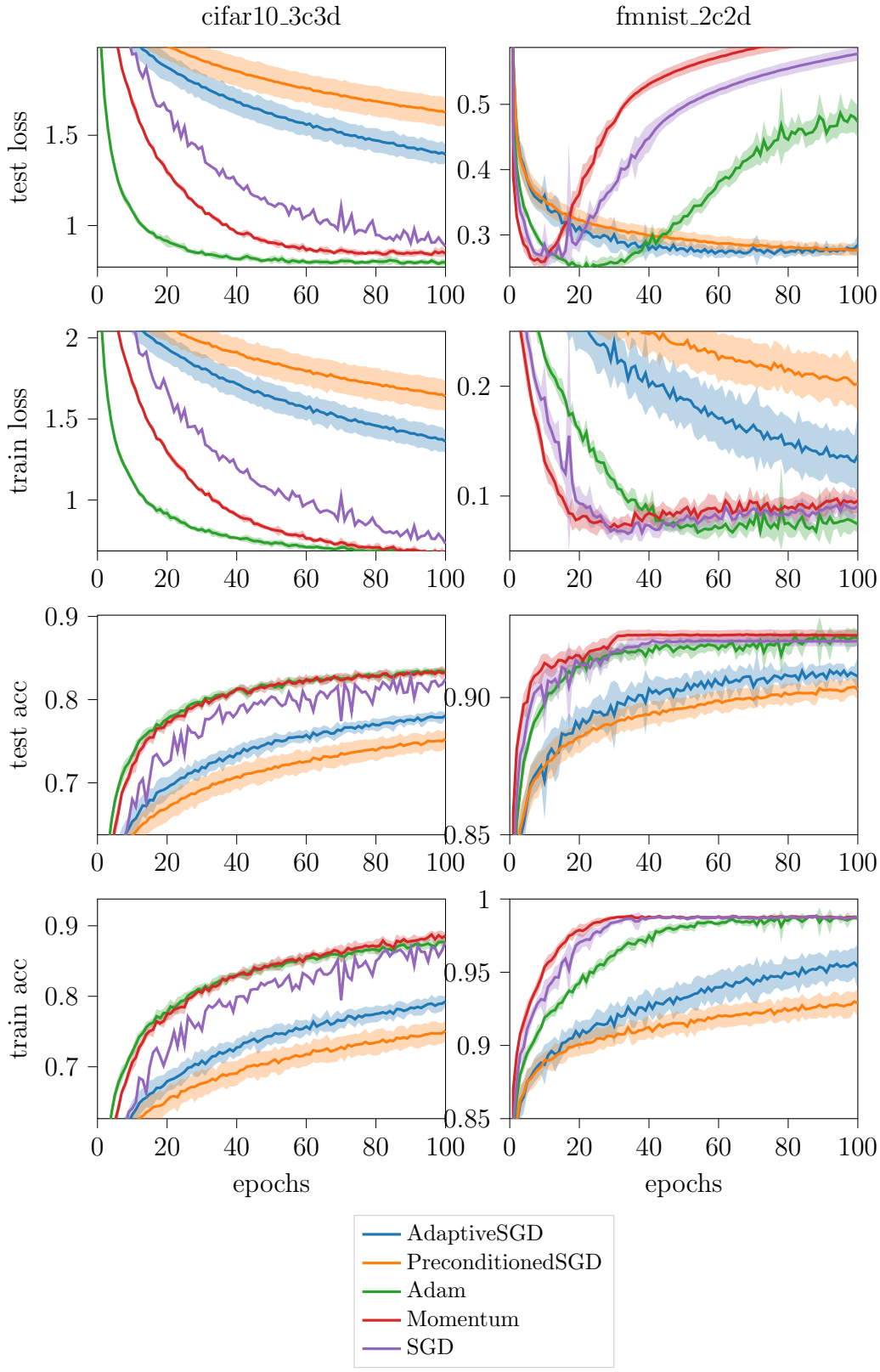

Figure 4.1: Both variants of the algorithm perform significantly worse than established alternatives. AdaptiveSGD is better than PreconditionedSGD. On the fmnist model, the baseline optimizers quickly reach a minimum, but then get worse on the loss. AdaptiveSGD and PreconditionedSGD arrive close the mimimum much later and don't diverge.

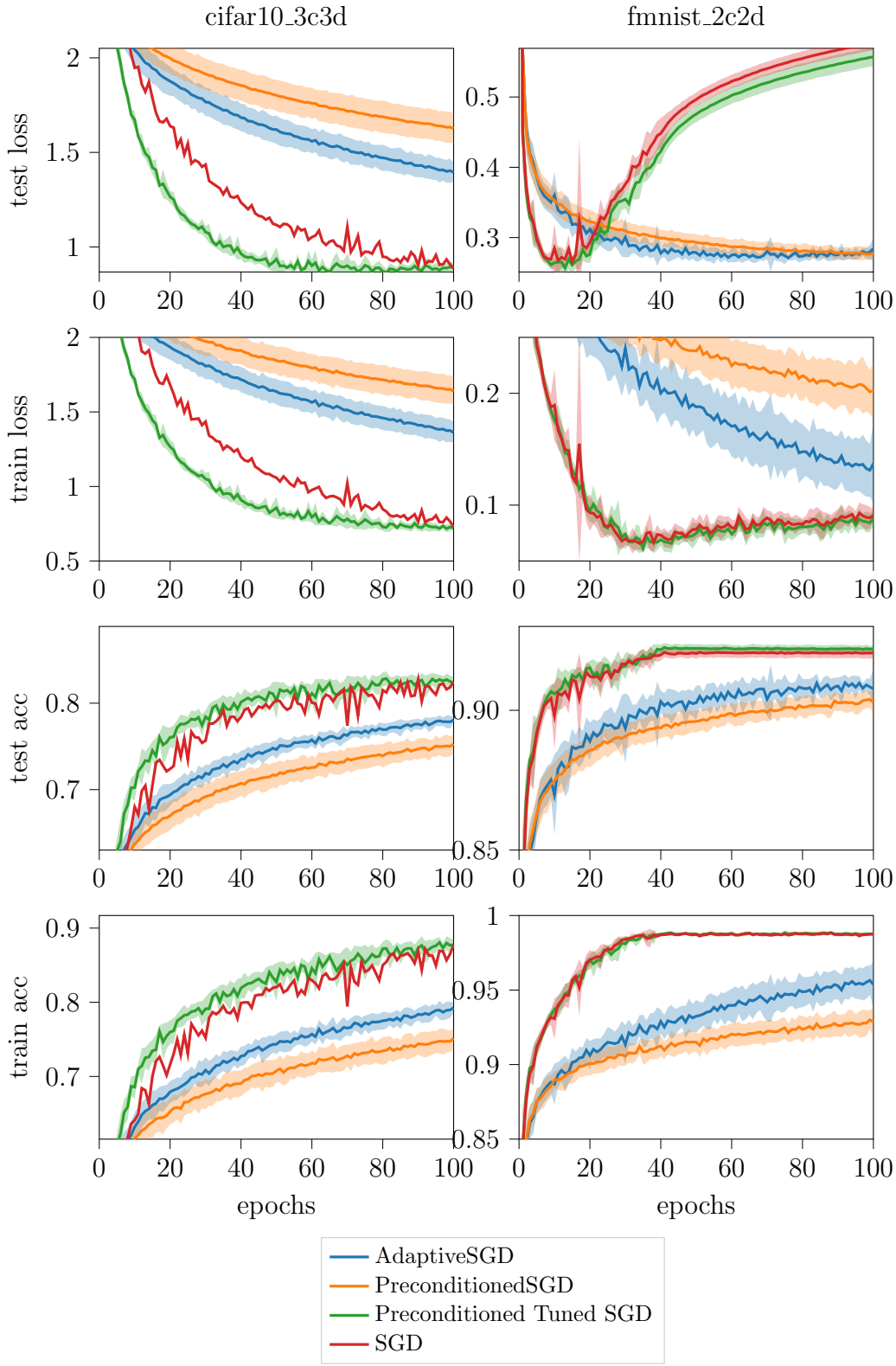

Figure 4.2: This figure shows the preconditioning algorithm, modified to use the same learning rate as well-tuned SGD, for every epoch. It performs better than standard tuned SGD using the same learning rate, and even better than the adaptive versions.

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 4.3. 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. `Adam` took about as long as SGD, with a mean time of 0.99 ($SD = 0.08$).

4.3 Experiment 3: Initialization

In [Roos and Hennig, 2019] it is reported that the studied 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

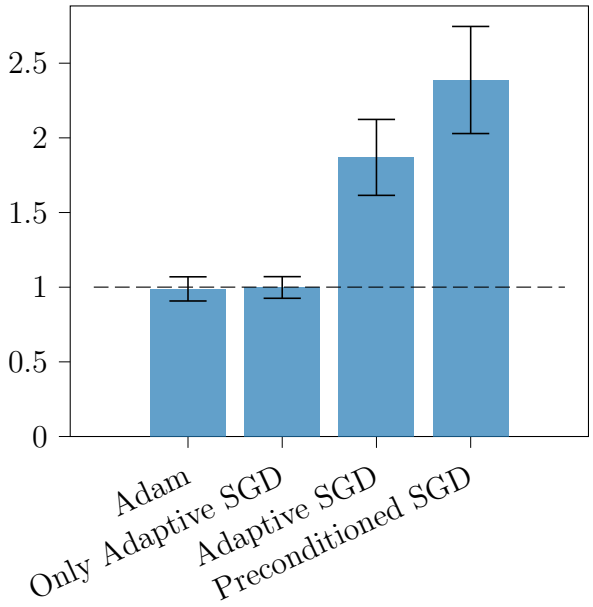

Figure 4.3: Experiment 2: Wallclock time per epoch, as tested on the DeepOBS testproblem mnist_mlp. A score of 1 represents the baseline runtime of SGD. The original algorithm is the slowest. Calculating, but not applying the preconditioner is considerably faster. Only gathering the prior observations is about as fast as SGD and Adam.

to include the original paper's net, either with explicit DeepOBS or implicit pytorch initialization. This model was then trained for both versions, with batch sizes of 32 (as in the original paper), 64 and 128, for five epochs, for each of 10 random seed values.

As is presented in figure 4.4, 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

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 4.5, 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.

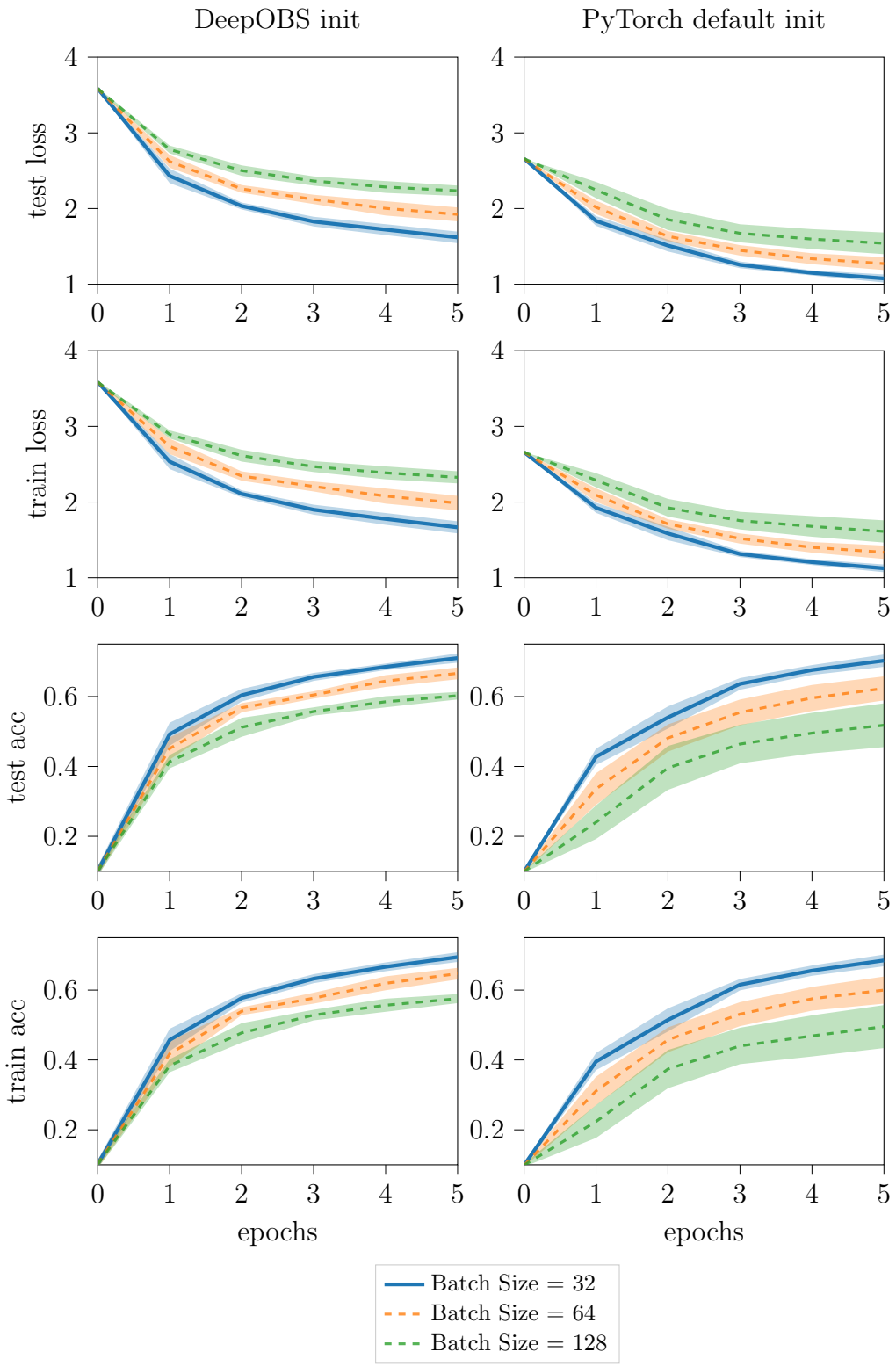

Figure 4.4: Experiment 3: PreconditionedSGD on a cifar10 net, comparing initialization methods. The algorithm is stable regardless of batch size and initialization method.

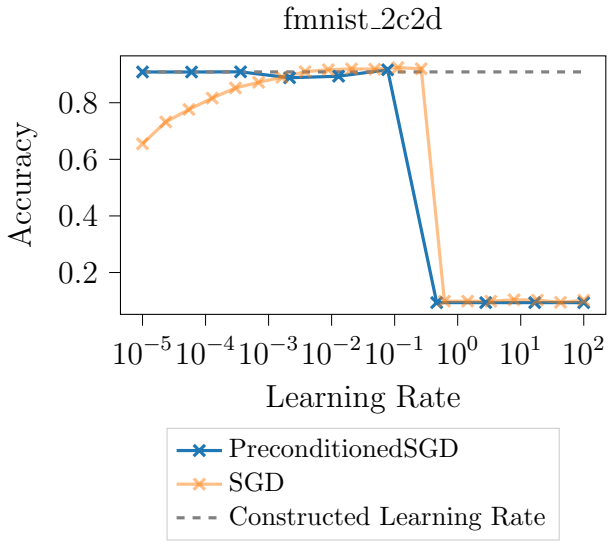

Figure 4.5: Experiment 4: The originally proposed algorithm has an optional hyperparameter "learning rate", which is used as SGD's learning rate in the first epoch. Below a certain model-specific threshold, it performs just as well as the constructed learning rate. Over that threshold, it diverges in the first epoch.

For SGD, which uses the same learning rate for all 100 epochs, there is a 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. When compared to a direct alternative without it, neither the preconditioner nor the adaptive step size show an increase in performance.

This stands in direct contrast to the findings of the original authors, who report the algorithm 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 Preconditioner's hyperparameters were not tuned in a systematic way in the experiments in this thesis.

Given this caveat, the comparatively poor performance of both Adap-

tiveSGD and PreconditionedSGD compared to the DeepOBS baselines can be explained. The Preconditioner is a quite complicated algorithm that has many hyperparameters, so exhaustive tuning was infeasible. While the DeepOBS baselines were tuned 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, 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 mini-batches. *Preconditioning + adaptive learning rate* performs worse than *tuned SGD*, but *Preconditioning + fixed tuned learning rate* performs better than *tuned SGD*. *Adaptive learning rate* performs worse than *tuned SGD* but better than *Preconditioning + adaptive learning rate*. This means that there are interaction effects between applying preconditioning and dynamically adapting the learning rate. On the preconditioned loss landscape, using the adaptive learning rate is beneficial, while on the non-preconditioned loss landscape, it is not optimal.

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. The prior observations add no considerable overhead compared to SGD, while both calculating and applying the Preconditioner add considerable overhead. Adam as another adaptive method is almost exactly as fast as SGD.

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. Two epochs of SGD can be run in the time it takes to compute the preconditioner and apply it during every step in one epoch.

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. CPU time would also be interesting, because it is a more direct measure of the amount of computations required by the algorithm. Unfortunately, DeepOBS does not provide this function as of now.

Experiment 3 and 4 show that it is possible 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 might be worth the tradeoff observed in experiment 2.

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 useful to keep the option of setting an initial learning rate. It is also unclear exactly why the new algorithm is stable.

Generally, the experiments in this thesis were mostly performed on convolutional nets, and it is unclear which of the found 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.

4.7 Feedback for DeepOBS

This thesis is one of the first semi-external projects that use DeepOBS and more specifically the pyTorch version of DeepOBS. Naturally, some issues and unexpected behavior came up while using it. This is a selection of Feedback on DeepOBS.

- Usability:
  - In general, DeepOBS is quite easy to use. The minimal examples provided with the documentation work well and are easy to adapt to other optimizers.
  - Output files used to be single-line strings of json. This was changed to easily human-readable formatting.
  - The way DeepOBS currently handles output files is very transparent (It simply writes json files for each optimizer run in an appropriate file structure). However, there is no unique place where DeepOBS saves and looks for metadata. Sometimes it's in the folder name, sometimes in the file itself, where it should be. It would make more sense to save consistently defined "run"-objects which point to all the runs that logically belong together.
  - When analyzing runs, DeepOBS should not rely on the operating system to determine things like the order of testproblems. The user should be able to define this, for example by manipulating a "run" object before plotting.
- Protocol:
  - The testproblems' hyperparameters like the batch size do affect optimizer performance a lot. DeepOBS provides default values, but it should be clearer if and how they can be left on their default values

to compare optimizers without introducing bias.
  - DeepOBS now uses a web-based issue tracking system where users can easily report bugs and request additional functionality.
- Features:
  - DeepOBS lacks a testproblem that uses parameter groups. While this might be less important for optimizer testing, it would be useful for optimizer development.
  - For measuring computational complexity, CPU time might be a more useful metric than Wallclock time, because that would rely less on the algorithm being tuned to the used hardware setup.
  - In the output files, DeepOBS should record if a run was stopped (and why) or if the algorithm was stable all the way throughout the run.

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.

conclude

Appendix A

Code

A.1 Singularity build recipe

```
#header
Bootstrap: docker
From: ubuntu:16.04

#Sections

%environment
# set environment variables

%post
# commands executed inside the container after os has been installed.
# Used for setup of the container

apt-get -y update
apt-get -y install python3-pip git python3-tk

python3 --version

pip3 install --upgrade pip
pip install torch torchvision
pip install git+https://github.com/abahde/DeepOBS.git@v2.0.0-beta#egg=deepobs
```

This build recipe was used to build the Singularity container by invoking `sudo singularity build`

A.2 DeepOBS Runscript for experiment 1

This code includes the definitions for PreconditionedSGD and AdaptiveSGD. It was run 10 times using different seeds each time.

```
"""Simple run script using SORunner."""

import torch.optim as optim
import deepobs.pytorch as pyt
from sorunner import SORunner
from probprec import Preconditioner
import numpy
import math

# DeepOBS setup

class PreconditionedSGD(Preconditioner):
    """docstring for PreconditionedSGD"""
    def __init__(self, *args, **kwargs):
    super(PreconditionedSGD, self).__init__(
        *args,
        optim_class = optim.SGD,
        **kwargs)

class AdaptiveSGD(Preconditioner):
    """docstring for PreconditionedSGD"""
    def __init__(self, *args, **kwargs):
    super(AdaptiveSGD, self).__init__(
        *args,
        optim_class = optim.SGD,
        **kwargs)

    def _apply_preconditioner(self):
        return;


# specify the Preconditioned Optimizer class
poptimizer_class = PreconditionedSGD

# and its hyperparameters
phyperparams = {'lr': {"type": float, 'default': None}}

# create the runner instances
```

```
prunner = SORunner(PreconditionedSGD, phyperparams)

runner = SORunner(AdaptiveSGD, phyperparams)

runner.run(testproblem='fmnist_2c2d')
runner.run(testproblem='cifar10_3c3d')
prunner.run(testproblem='fmnist_2c2d')
prunner.run(testproblem='cifar10_3c3d')
```

A.3 Slurm Batch Definition File

This file was used to run experiment 1 on the compute cluster.

```
#!/bin/bash

####
#a) Define slurm job parameters
####

#SBATCH --job-name=prec

#resources:

#SBATCH --cpus-per-task=4
# the job can use and see 4 CPUs (from max 24).

#SBATCH --partition=day
# the slurm partition the job is queued to.

#SBATCH --mem-per-cpu=3G
# the job will need 12GB of memory equally distributed on 4 cpus.
# (251GB are available in total on one node)

#SBATCH --gres=gpu:1
#the job can use and see 1 GPUs (4 GPUs are available in total on one node)

#SBATCH --time=03:30:00
# the maximum time the scripts needs to run
# "hours:minutes:seconds"

#SBATCH --array=43-51
```

```
#SBATCH --error=job.%J.err
# write the error output to job.*jobID*.err

#SBATCH --output=job.%J.out
# write the standard output to job.*jobID*.out

#SBATCH --mail-type=ALL
#write a mail if a job begins, ends, fails, gets requeued or stages out

#SBATCH --mail-user=ludwig.bald@student.uni-tuebingen.de
# your mail address

####
#b) copy all needed data to the jobs scratch folder
####

DATA_DIR=/scratch/$SLURM_JOB_ID/data_deepobs/

mkdir -p $DATA_DIR/pytorch
mkdir -p $DATA_DIR/pytorch/FashionMNIST/raw
cp -R /common/datasets/cifar10/. $DATA_DIR/pytorch/
cp -R /common/datasets/Fashion-MNIST/. $DATA_DIR/pytorch/FashionMNIST/raw/

####
#c) Execute the code
####


singularity exec ~/image.sif python3 ~/exp_preconditioning/runscript.py
    --data_dir $DATA_DIR --random_seed $SLURM_ARRAY_TASK_ID

echo DONE!
```

A.4 Repository

Everything that's needed to reproduce the experiments in this thesis is freely accessible. The python and bash source code for the preconditioner, cluster use, experiment setup are available in the following repository: `https://github.com/ludwigbald/probprec/` Raw experiment result files and the LaTeXsources for this thesis and the presentation are in the same repository.

Bibliography

[Bahde, 2019] Bahde, A. (2019). Tuning procedure for deepobs. Master's thesis, Universität Tübingen. Master thesis.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

[Brown and Hamarneh, 2016] Brown, C. J. and Hamarneh, G. (2016). Machine learning on human connectome data from mri. *arXiv preprint arXiv:1611.08699*.

[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*.

[Gibney, 2016] Gibney, E. (2016). Google ai algorithm masters ancient game of go. *Nature News*, 529(7587):445.

[Krizhevsky et al., 2014] Krizhevsky, A., Nair, V., and Hinton, G. (2014). The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55.

[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*.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W. M., Dudzik, A., Huang, A., Georgiev, P., Powell, R., et al. (2019). Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind Blog*.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

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, 7. Oktober 2019 Ludwig Bald