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

Bachelor Thesis Cognitive Science

Title of thesis

Ludwig Bald

August 26, 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:
Title of thesis
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 In this thesis I present my implementation of the probabilistic preconditioning algorithm proposed in [de Roos and Hennig, 2019]. I use DeepOBS as a benchmarking toolbox, examining the effect of this kind of preconditioning on various optimizers and test problems. The results...

theabstract, citing!

cite

Zusammenfassung

Abstract auf Deutsch

Acknowledgments

If you have someone to Acknowledge ;) Aaron, Filip

Contents

Chapter 1

Introduction

What is this all about?

Chapter 2

Fundamentals and Related Work

2.1 Deep learning

2.1.1 Artificial Neural Networks

- Neural Networks exist. There are a few varieties and key developments, but they all share the underlying neural structure. The original idea was to somewhat model biological neurons in order to mimic their capacity for structural memory. As a mathematical object, a neuron is an activation function which takes on a certain value based on the weighted activations of the inputs (usually of their sum), which are either neurons themselves, or outside information given as an input to these neurons. Often there is a bias input, which is always 1 and allows for static activation. Neurons are usually organized in layers, where each neuron has as input all the previous layers' neurons.

2.1.2 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.

2.1.3 Regularization

2.1.4 Optimization

In order to find the optimal values for the parameters of a neural network (usually weights and biases), different methods have been proposed. For small optimization problems (small number of parameters), it is often possible to directly find the optimal values analytically. For large optimization problems, the analytical solution is computationally intractable. As a result, numerical algorithms have emerged. These iterative algorithms use the available limited information in order to gradually infer an approximation for the true best values for the parameters. The number of iterations needed until the algorithm reaches convergence (i.e. the new solution is no longer signigicantly better than the previous one) varies based on the implicit prior assumptions about the data that the algorithm makes. These optimization processes have "hyperparameters" (different from the models "parameters" that we want to optimize). A great deal of research has gone into these numerical optimization processes, both in the general case and more specifically, for deep learining. 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.

2.1.5 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 automatic differentiation: 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.

2.1.6 Stochastic Gradient Descent

Once the gradient at a certain point 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 rolling down a hill and ending up on a local minimum. SGD 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. 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.

2.2 Preconditioning

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".

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.

Condition number vs Spectral Radius

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.

2.3 Something about Probabilistic things

- Probablilty Distribution, normal distribution, multivariate Gaussian distribution

2.4 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.

2.5 Related Work

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

Chapter 3

The algorithm

3.1 Description of the algorithm

The exact inner workings of the algorithm are described in more detail in [de Roos and Hennig, 2019]. Here I will give an overview of the algorithm's structure and steps.

3.2 Modifications of the algorithm

- Seperate learning rates for Parameter groups. This was not needed, but theoretically should improve convergence rates. - Automatic restart of the estimation process, once the old Hessian is determined to be out of date.

Chapter 4

Implementation/Methods

High-Level to low-level details

4.1 Documentation for the class Preconditioner

4.1.1 Methods

4.1.2 Implementation Details

In the following, I will highlight and explain some key software design decisions I took while implementing and refactoring the algorithm.

- Self-contained class with minimal interface, while marking all the internal functions as hidden
- Being able to restart the estimation process from outside, without having to do a new instance of anything
- Class inherits from torch.optim.Optimizer, because that makes it easier to use and easier to investigate using DeepOBS.
- The class functions as a wrapper for other optimizer classes, using the decorator pattern on the `step()` function.
- All the parameter-wise variables and values are stored in the state dict, which means we can save and load backups of the optimizer state.
- There is support for parameter groups, which all pytorch Optimizers should exhibit.
- external logging of internal variables can be done using the `get\_log()` method

4.2 Realization of the Test problems

4.3 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 the compute nodes using the workload manager Slurm.

Chapter 5

Experiment

5.1 Results

- The algorithm greatly depends on the initialization used for the model. For some models, incorrect initialization means we can't find anything. (DeepOBS mnist_vae, Seed 45 failed to provide anything)

5.2 Analysis

5.3 Discussion

5.4 Further research/development

- Make the process more robust using tests and catching exceptions and so on.
- Formulate a probabilistic version of the auto-restart feature.

Chapter 6

Conclusion

Appendix A

An appendix

Here you can insert the appendices of your thesis.

Bibliography

[de Roos and Hennig, 2019] de Roos, F. and Hennig, P. (2019). Active probabilistic inference on matrices for pre-conditioning in stochastic optimization. *arXiv preprint arXiv:1902.07557*.

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, 26. August 2019 Ludwig Bald