Master Thesis

On Scalable Deep Learning and Parallelizing Gradient Descent

Joeri R. Hermans

Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science of Artificial Intelligence

at

Maastricht University
Faculty of Humanities and Sciences
Department of Data Science & Knowledge Engineering
Maastricht, The Netherlands

Preface

This thesis is submitted as a final requirement for the Master of Science degree at the Department of Data Science & Knowledge Engineering of Maastricht University, The Netherlands. The subject of study originally started as a pilot project with Jean-Roch Vlimant, Maurizio Pierini, and Federico Presutti of the EP-UCM group (CMS experiment) at CERN. In order to handle the increased data rates of LHC Run 3 and High Luminosity LHC, the CMS experiment is considering to construct a new architecture for the High Level Trigger based on Deep Neural Networks. However, they would like to significantly decrease the training time of the models as well. This would allow them to tune the neural networks more frequently. As a result, we started to experiment with various state of the art distributed optimization algorithms. Which resulted in the achievements and insights presented in this thesis.

I would like to express my gratitude to several people. First and foremost, I would like to thank my promotors, Gerasimos Spanakis, and Rico Möckel for their expertise and suggestions during my research, which drastically improved the quality of this thesis. Furthermore, I would also like to thank my friends, colleagues and scientists at CERN for their support, feedback, and exchange of ideas during my stay there. It was a very motivating and inspiring time in my life. Especially the support and experience of my CERN supervisors, Zbigniew Baranowski, and Luca Canali, was proven to be invaluable on multiple occasions. I would also like to thank them for giving me the personal freedom to conduct my own research. Finally, I would like to thank my parents and grandparents who always supported me, and who gave me the chance to explore the world in this unique way.

Joeri R. Hermans
Geneva, Switzerland 2016 - 2017

Abstract

Abstract here.

Summary

Summary here.

Contents

Abbreviations and Notation

η	Static learning rate
η_t	Learning rate with respect to time t.
λ	Communication period, or frequency of commits to the parameter server.
$\mathcal{L}(\theta\ ;\ \mathbf{x})$	Loss function with respect to parametrization θ and input $\mathbf{x}$.
τ	Staleness
θ_t^k	Parametrization of worker k at time t.
$\tilde{\theta}_t$	Center variable, or central parametrization maintained by the parameter server.
$\triangleq$	Is defined as
$J(\theta)$	Loss with respect to parameterization θ.
m	Mini-batch size
n	Number of parallel workers.
ADAG	Asynchronous Distributed Adaptive Gradients
ASGD	Asynchronous Stochastic Gradient Descent
CERN	European Organization for Nuclear Research
CMS	Compact Muon Solenoid
EASGD	Elastic Averaging Stochastic Gradient Descent
GD	Gradient Descent
HEP	High Energy Physics
HL-LHC	High Luminosity Large Hadron Collider
LHC	Large Hadron Collider
MNIST	Mixed National Institute of Standards and Technology database
PS	Parameter Server
SGD	Stochastic Gradient Descent

Chapter 1

Introduction

In this chapter we introduce the main concept, and problems surrounding the parallization of gradient descent. We famillliarize the reader with the topic and some notation by providing some context why someone would like to apply said technique. Furthermore, in Section 1.4, we summarize the problem statement and provide several research questions which will guide the research in this work. Finally, we conclude this chapter in Section 1.5 with a brief outline of the thesis.

1.1 Motivation

In recent years it has been shown that being able to train large and deep neural networks result in state-of-the-art performance [10, 3], especially regarding unsupervised feature learning and image recognition. However, consider the required time, and cost of the infrastructure that would be required in order to train a large model in a reasonable amount of time. Furthermore, it is not only the training time and cost of the infrastructure which need to be taken into consideration, but also the volume of the data. The amount of information that will be gathered will be an increasing important factor in the next few years. Not only with respect to big technology companies and government organizations, but also scientific surveys with limited budgets. These scientific surveys will generate more experimental data than ever [1, 5], and will have to process and analyze that data. To solve the problem of increased computational workloads and budget freezes, the High Energy Physics (HEP) community is exploring and researching machine learning approaches to fit physics problems [2, 9, 7] with the intention to improve detection quality, or reduce computational constraints.

However, the sheer size of these datasets severly impacts the training time of the models. In order to resolve this issue, one could sample some representative subset of the data to reduce the training time. The disadvantage of this approach is that some instances, i.e., data points, might not appear in the final training set. This is especially a problem in Deep Learning, where models usually benifit from having access to a lot of training data due to the high dimensionality of the parametrization [3]. To resolve this issue, Dean et al. [3] introduce two new paradigms to decrease the training time of a large model. The two paradigms, *Model Parallelism*, briefly discussed in Section 1.2, and *Data Parallelism*, discussed in Section 1.3, are inherently different ways of decreasing the training time of a model.

The first paradigm, *Model Parallelism*, is intuitively the most straightforward paradigm since it deals with the parallelization of the computations within a *single* model, i.e., how to parallelize the computations of a single model over multiple machines, or multiple processes. The second paradigm, which will be the main focus of this thesis, is *Data Parallelism*. As stated above, the main concept of Data Parallelism will be discussed in detail in Section 1.3. However, for completion, think of Data Parallelism as a technique to *parallelize gradient descent*. This is done by allocating n processes over possibly n different machines, and splitting the training set into n *partitions*, or *data shards*. For further convenience, we will call such a process a *worker*. In the next step, we assign a single distinct partition to a worker. Meaning, the worker will not be able to fetch training data from other partitions

since those have been assigned to different workers. However, in certain data parallel settings, it is benificial to actually consume data from other partitions, once a worker has finished its partition. Finally, the goal of these workers is to work together, and optimize the parameters of a central model.

A lot of different distributed optimization schemes have been suggested in recent years [11, 3, 4]. Most of the recent contributions try to push the limits of asynchronous Data Parallelism, discussed in Section 3.2, by simply *annealing* the gradients with respect to some hyperparameter to improve the convergence when the number of workers increases. This suggests that there is an intrinsic limit to asynchronous Data Parallelism, as suggested by [8]. As a result, why don't we simply reduce the number of parallel workers if we reduce the impact of the gradient updates by means of annealing anyway? The approach of reducing the number of parallel workers in such a situation has been suggested by [4], where they perform a *grid-search* of the training hyperparamers (this includes the number of workers) in order to provide the optimal hyperparameters within a training epoch. However, the disadvantage of this technique is that after every epoch, or a specific number of iterations, a grid-search of the hyperparameters has to be performed in order to obtain the optimal configuration of the hyperparameters to ensure convergence.

This brings us to the main motivation behind this work. We intent to obtain a better understanding of *asynchronous* Data Parallism by building upon previous work, and combine it with novel insights to construct a new distributed optimization scheme without introducing new hyperparameters, or relying on grid-searches to optimize the configuration of existing hyperparameters.

1.2 Model Parallelism

TODO

1.3 Data Parallelism

Data Parallelism is an inherently different methodology of optimizing parameters. As stated above, it is a technique to *parallelize gradient descent*, and thereby reducing the overal training time of a model. In essence, Data Parallelism achieves this by having n workers optimizing a central model, and at the same time, processing n different shards (partitions) of the dataset in parallel over multiple workers[1]. The workers are coordinated in such a way that they optimize the parametrization of a central model, which we denote by $\tilde{\theta}_t$. The coordination mechanism of the workers can be implemented in many different ways. Nevertheless, a popular approach to coordinate workers in their task to optimize the central objective, is to employ a centralized *Parameter Server* (PS). The sole responsibility of the parameter server is to aggregate model updates coming from the workers (*worker commits*), and to handle parameter requests (*worker pulls*). In general, there are several approaches towards data parallelism. However, all approaches can be categorized into two main groups, i.e., *Synchronous Data Parallelism*, and *Asynchronous Data Parallelism*.

[1] As stated in Section 1.1, a worker is a process on a single machine. However, it is possible that multiple workers share the same machine. Nevertheless, one could construct the distribution mechanism (even manually) in such a way every worker will be placed on a different machine.

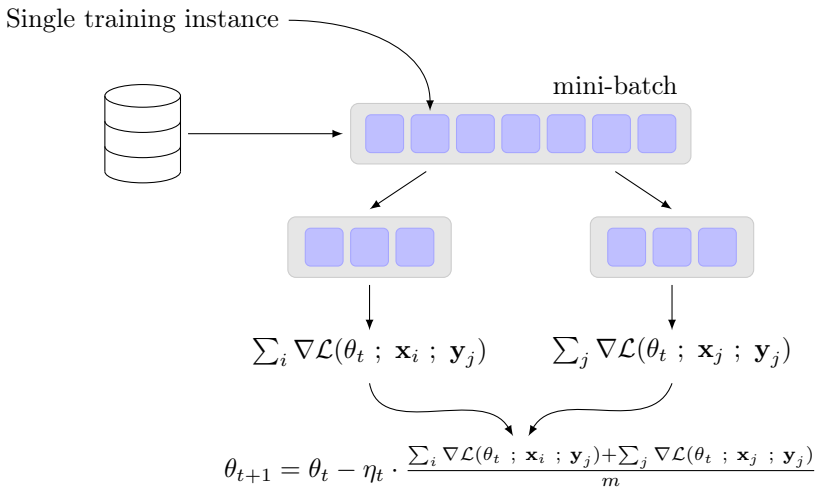

Figure 1.1: Mini-batch parallelism could be viewed as an instance of synchronous data parallelism without a centralized parameter server. Given a mini-batch size of m, we split the mini-batch into several partitions, where a specific worker is responsible for the computation of its own partition. The synchronous nature of this approach lies within the aggregation of the computed gradients, i.e., the results of all workers need to be aggregated, and afterwards averaged in order to integrate the current gradient into the model.

In order to formalize the main concept of Data Parallelism, let us assume we have a dataset D, which contains our training data, and that we are able to distribute dataset D over n different workers $\mathcal{W} = \{w_1, \ldots, w_n\}$. Where every worker $w_i \in \mathcal{W}$ holds a copy of the central model, thus, a copy of the parameterization of the central model $\tilde{\theta}_0$. Furthermore, we denote the parametrization of a particular worker k at time t by θ_t^k. Of course, if a worker wants to contribute to the optimization of the central model, the worker needs to be able to relay update information and retrieve the most recent parameterization of the central model. This is done by instantiating a parameter server, where workers will be able to *commit* their updates, and *pull* the most recent parameterization of the central model. The parameterization of the central model is called the *central variable*, which we denote by $\tilde{\theta}_t$. In the final preperation step, before the actual training starts, $\mathcal{D}$ will be split into roughly n equally sized partitions $\mathcal{P} = \{p_1, \ldots, p_n\}$, where $|p_i| \approx \frac{1}{|\mathcal{D}|}$, and where p_i will be assigned to the corresponding worker w_i.

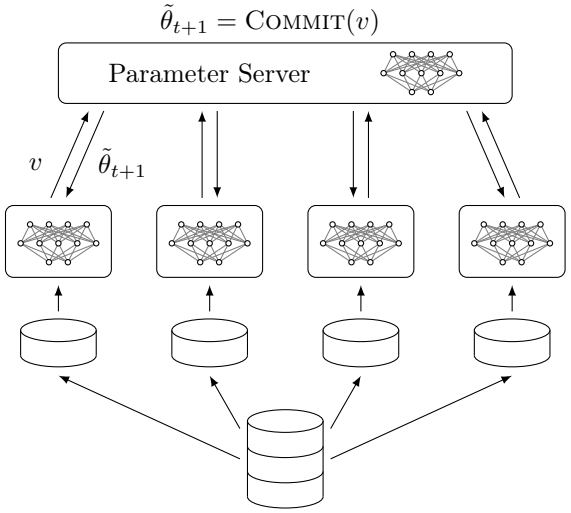

Figure 1.2: Schematic representation of a data parallel approach. In this methodology we spawn n workers (not necessarily on different machines), and assign a data shard (partition) of the dataset to every worker. Using this data shard, a worker i will iterate through all mini-batches to produce a gradient, $\nabla\mathcal{L}_i(x)$, for every mini-batch x. Next, $\nabla\mathcal{L}_i(x)$ is send to the parameter server, which will incorperate the gradient using an UPDATE mechanism.

In general, all data parallel approaches share a similar training procedure, i.e., every worker computes some variable which is communicated with the parameter server to update the central model. In most cases, this variable represents some change $\Delta\theta$ which needs to be applied to the central variable $\tilde{\theta}_t$. However, some approaches such as [11], actually require that the complete worker parametrization θ_t^k is sent to the parameter server. To simplify this specific optimizer detail in this chapter, we denote the variable that is sent to the parameter server by v.

Algorithm 1 Describes the general optimization procedure of a worker in a data parallel setting. The worker will be identified with a certain index k, the other parameter $p_k \in \mathcal{P}$, is the data partition which has been assigned to worker k.

procedure WORKER(k, p_k)
  $\theta_0^k \leftarrow \textsc{Pull}()$
  $t \leftarrow 0$
  while not converged **do**
    $\mathbf{m} \leftarrow \textsc{FetchNextMiniBatch}(p_k)$
    $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \cdot \nabla\mathcal{L}(\theta_t^k\ ;\ \mathbf{m})$ ▷ Optimization step, could be [6], or other optimizer.
    $v \leftarrow \textsc{PrepareCommit}()$
    $\textsc{Commit}(v)$
    $\theta_t^k \leftarrow \textsc{Pull}()$
    $t \leftarrow t + 1$
  end while
end procedure

Algorithm 2 Intialization and variable handling procedures of a parameter server. Before the distributed optimization starts, the IntializeParameterServer procedure is called to initialize the local parameters, given the parametrization θ of the specified model. We would like to note that t maintained by the parameter server, is different from the t variable specified in Algorithm 1.

procedure IntializeParameterServer(θ)
    $\tilde{\theta}_0 \leftarrow \theta$
    $t \leftarrow 0$
end procedure

procedure Commit(v)
    $\tilde{\theta}_{t+1} \leftarrow$ ApplyCommit(v)
    $t \leftarrow t + 1$
end procedure

procedure Pull( )
    return $\tilde{\theta}_t$
end procedure

1.4 Problem Statement

TODO

1.5 Thesis Outline

TODO

Chapter 2

Optimization Algorithms

Chapter 3

Distributed Deep Learning

3.1 Synchronous Data Parallelism

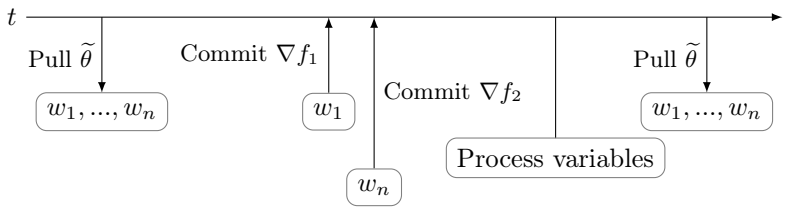

Figure 3.1: Caption here

3.1.1 Model Averaging

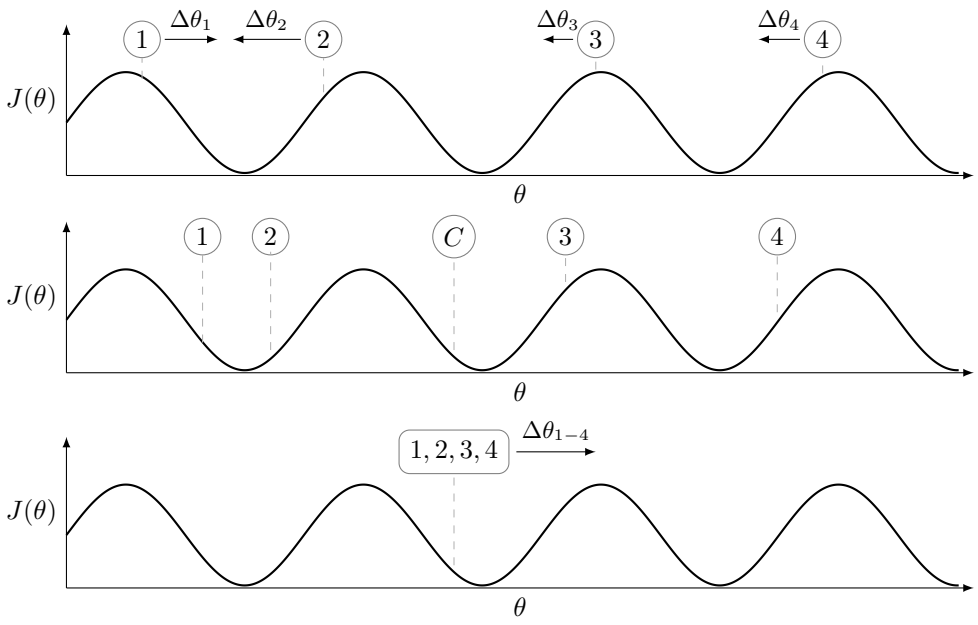

Figure 3.2: Caption here

3.1.2 Elastic Averaging SGD

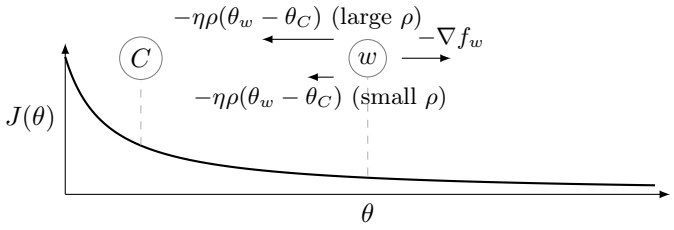

Figure 3.3: EASGD Caption here

3.2 Asynchronous Data Parallelism

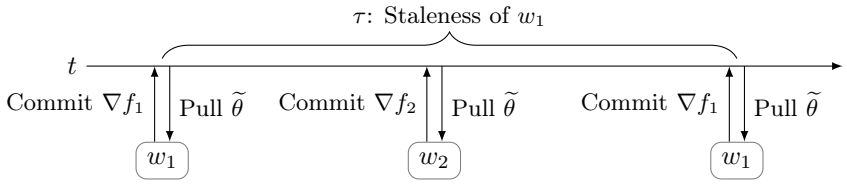

Figure 3.4: Caption here.

3.2.1 Asychrony Induced Momentum

3.2.2 Hogwild!

3.2.3 DOWNPOUR

3.2.4 Asynchronous Elastic Averaging SGD

Chapter 4

Accumulated Gradient Normalization

$$\Delta\theta = -\frac{\sum_{i=0}^{\lambda} \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla f(\theta_i; x_{ij}; y_{ij})}{\lambda} \tag{4.1}$$

Chapter 5

Asynchronous Distributed Adaptive Gradients

In this chapter we introduce a novel optimizer called ADAG. ADAG, or *Asynchronous Distributed Adaptive Gradients*, is an optimization process designed with data parallel methods in mind. We build upon previous work [3, 4, 6, 11] and incorperate new insights backed up by theory and experimental evidence. We start in Section 5.1 by formalizing the problem setting. In Section 5.2, we summarize previous work on distributed (data parallel) optimization. Section 5.3 will describe our algorithm in detail, supported by intuition and theory. Finally, we experimentally show the effectiveness of our approach in Section 5.4 and give some points for future work in Section 5.5.

5.1 Problem setting

5.2 Previous work

5.3 Algorithm

5.3.1 Update rule

5.4 Experiments

5.4.1 Handwritten digit classification

5.4.2 Higgs event detection

5.4.3 Sensitivity to hyperparameters

5.4.4 Sensitivity to number of parallel workers

5.5 Future work

Chapter 6

Distributed Keras

Chapter 7

Experiments

Chapter 8

Conclusion

Bibliography

[1] G Apollinari et al. *High-Luminosity Large Hadron Collider (HL-LHC): Preliminary Design Report*. Geneva: CERN, 2015. URL: `https://cds.cern.ch/record/2116337`.

[2] Jianming Bian. "Recent Results of Electron-Neutrino Appearance Measurement at NOvA". In: *arXiv preprint arXiv:1611.07480* (2016).

[3] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[4] Stefan Hadjis et al. "Omnivore: An optimizer for multi-device deep learning on cpus and gpus". In: *arXiv preprint arXiv:1606.04487* (2016).

[5] Zeljko Ivezic et al. "LSST: from science drivers to reference design and anticipated data products". In: *arXiv preprint arXiv:0805.2366* (2008).

[6] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[7] Gilles Louppe, Michael Kagan, and Kyle Cranmer. "Learning to Pivot with Adversarial Networks". In: *arXiv preprint arXiv:1611.01046* (2016).

[8] Ioannis Mitliagkas et al. "Asynchrony begets Momentum, with an Application to Deep Learning". In: *arXiv preprint arXiv:1605.09774* (2016).

[9] Luke de Oliveira, Michela Paganini, and Benjamin Nachman. "Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis". In: *arXiv preprint arXiv:1701.05927* (2017).

[10] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[11] Sixin Zhang, Anna E Choromanska, and Yann LeCun. "Deep learning with elastic averaging SGD". In: *Advances in Neural Information Processing Systems*. 2015, pp. 685–693.

Appendices