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

Preface ii

Abstract iii

Summary iv

Abbreviations and Notation vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Model Parallelism . . . 2
- 1.3 Data Parallelism . . . 3
- 1.4 Problem Statement . . . 10
- 1.5 Thesis Outline . . . 11

2 Distributed Deep Learning 12
- 2.1 Introduction . . . 12
- 2.2 Synchronous Data Parallelism . . . 15
  - 2.2.1 Model Averaging . . . 15
  - 2.2.2 Elastic Averaging SGD . . . 17
- 2.3 Asynchronous Data Parallelism . . . 17
  - 2.3.1 DOWNPOUR . . . 17
  - 2.3.2 Dynamic SGD . . . 17
  - 2.3.3 Asychrony Induced Momentum . . . 17
  - 2.3.4 Asynchronous Elastic Averaging SGD . . . 17
- 2.4 Hybrids . . . 17

3 Accumulated Gradient Normalization 18

4 Asynchronous Distributed Adaptive Gradients 19
- 4.1 Problem setting . . . 19
- 4.2 Previous work . . . 19
- 4.3 Algorithm . . . 19
  - 4.3.1 Update rule . . . 19
- 4.4 Experiments . . . 19
  - 4.4.1 Handwritten digit classification . . . 19
  - 4.4.2 Higgs event detection . . . 19
  - 4.4.3 Sensitivity to hyperparameters . . . 19
  - 4.4.4 Sensitivity to number of parallel workers . . . 19
- 4.5 Future work . . . 19

5 Distributed Keras 20

6 Experiments **21**
6.1 Handwritten Digit Classification 21
6.2 CMS Track Reconstruction and Event Identification 21
6.2.1 Data Representation 22
6.2.2 Model Development 22

7 Conclusion **23**

References **24**

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.
$\mathcal{L}(\theta\ ;\ \mathbf{x}\ ;\ \mathbf{y})$	Loss function with respect to parametrization θ, input $\mathbf{x}$, and expected output $\mathbf{y}$.
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
HE	Hardware Efficiency
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
SE	Statistical Efficiency
SGD	Stochastic Gradient Descent
TE	Temporal Efficiency

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 [16, 4], 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, 7], 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, 14, 12] 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 [4]. To resolve this issue, Dean et al. [4] 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 [17, 4, 5]. Most of the recent contributions try to push the limits of asynchronous data parallelism, discussed in Section 2.3, 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 [13]. 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 [5], 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

In *model parallelism*, a single model is distributed over multiple machines [4]. The performance benefits of distributing a deep network across multiple machines mainly depends on the structure of the model. Models with a large number of parameters typically benefit from access to more CPU cores and memory, up to the point where communication costs, i.e., propagation of weight updates and synchronization mechanisms, dominate [4].

Let us start with a simple example in order to illustrate this concept more clearly. Imagine having a perceptron, as depicted in Figure 1.1. In order to parallelize this efficiently, we can view a neural network as a dependency graph, where the goal is to minimize the number of synchronization mechanisms, assuming we have unlimited resources. Furthermore, a synchronization mechanism is only required when a node has more than 1 *variable* dependency. A variable dependency is a dependency which can change in time. For example, a bias would be a *static* dependency, because the value of a bias remains constant over time. In the case for the perceptron shown in Figure 1.1, the parallelization is quite straightforward. The only synchronization mechanism which should be implemented resides in output neuron since $y \triangleq \sigma(\sum_i w_i x_i)$ where σ is the activation function of the output neuron.

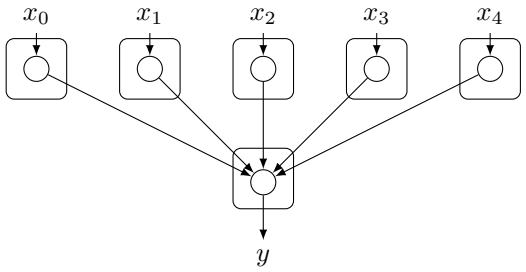


Figure 1.1: A perceptron partitioned using the *model parallelism* paradigm. In this approach every input node is responsible for accepting the input x_i from some source, and multiplying the input with the associated weight w_i. After the multiplication, the result is sent to the node which is responsible for computing y. Of course, this node requires a synchronization mechanism to ensure that the result is consistent. The synchronization mechanism does this by waiting for the results y depends on.

1.3 Data Parallelism

Data parallelism is an inherently different methodology of optimizing parameters. As stated above, it is a technique to reduce the overall 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, where some do not require a parameter server. However, all approaches can be categorized into two main groups, i.e., *Synchronous Data Parallelism*, and *Asynchronous Data Parallelism*.

Synchronous Data Parallelism can be usually identified by the presence of one or multiple locking mechanisms. As in Software Engineering, the purpose of these locking mechanisms is to preserve the consistancy of the state of a model. As an example, let us consider mini-batch parallelism in Figure 1.2 for a moment. Despite it is trivial to implement locally, one could view mini-batch parallelism as an instance of synchronous data parallelism. First and foremost, mini-batch parallelism is a data parallel technique because we split the mini-batch into several partitions where every partition is consumed by its own worker to produce the sum of the gradients, or *accumulated gradient*, as a result. Finally, mini-batch parallelism is synchronous in nature because in order to compute θ_{t+1}, we need to obtain the averaged gradients $\frac{\sum_i \nabla_\theta \mathcal{L}(\theta_t; \mathbf{x}_i; \mathbf{y}_i)}{m}$, which is actually the sum of the accumulated gradients of all workers, divided by the number of training instances m in the original mini-batch. As a result, the synchronization barrier is present right before the averaging of the accumulated gradients, since these are the intermediary results we have to wait for before applying a gradient update.

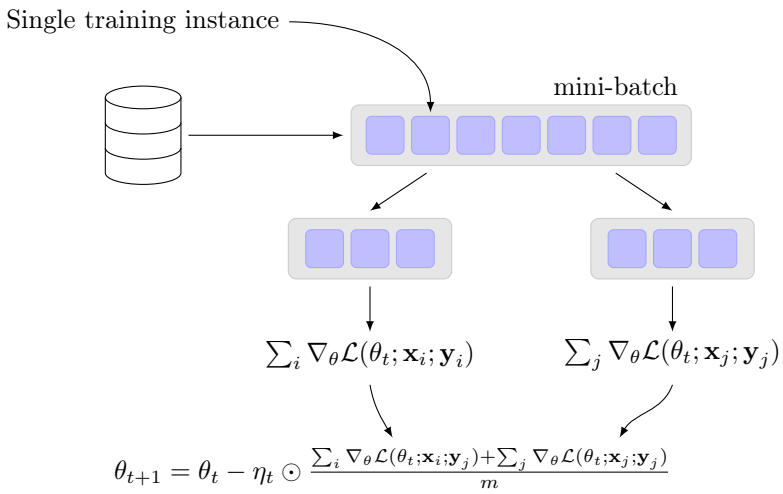


Figure 1.2: Mini-batch parallelism could be viewed as an instance of synchronous data parallelism without a centralized parameter server. Given a mini-batch size of m, we split the mini-batch into several partitions, where a specific worker is responsible for the computation of its own partition. The 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.

[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.

However, consider what happens when the computational resources on your machine are constrained, or even fully utilized? That is, due to the limited amount of available CPU cores (or even GPU's) your parallization of the mini-batch computation doesn't scale very well on your machine. The most straightforward solution would be to purchase more performant hardware, but this is not always possible, not only from a financial perspective, but also from a physical one. An alternative approach would be to solve the problem like a distributed system. In order to make this particular approach work, we need a parameter server to coordinate the workers. Since this still is a synchronous approach, the locking mechanism in this particular case is the parameter server itself, since the parameter server will not be able to send the next parameterization θ_{t+1} of the model to the workers because the parameter server can only compute the θ_{t+1} once it received, and processed all accumulated gradients as shown in Figure 1.3. Yet, if one or multiple machines encouter some unmodeled load, for example, because an other user is running a CPU intensive program, then the synchronization mechanism might actually be a serious bottleneck, because during that time the reserved resources on other machines are not being utilized. This effect becomes even more prominent when the infrastructure is *non-homogeneous*, i.e., multiple machines with different hardware in the same computing cluster cause the workers to be out-of-sync (on top of the unmodeled system behaviour), which in turn results in more waits enforced by the parameter server as it acts as a locking mechanism in synchronous data parallelism.

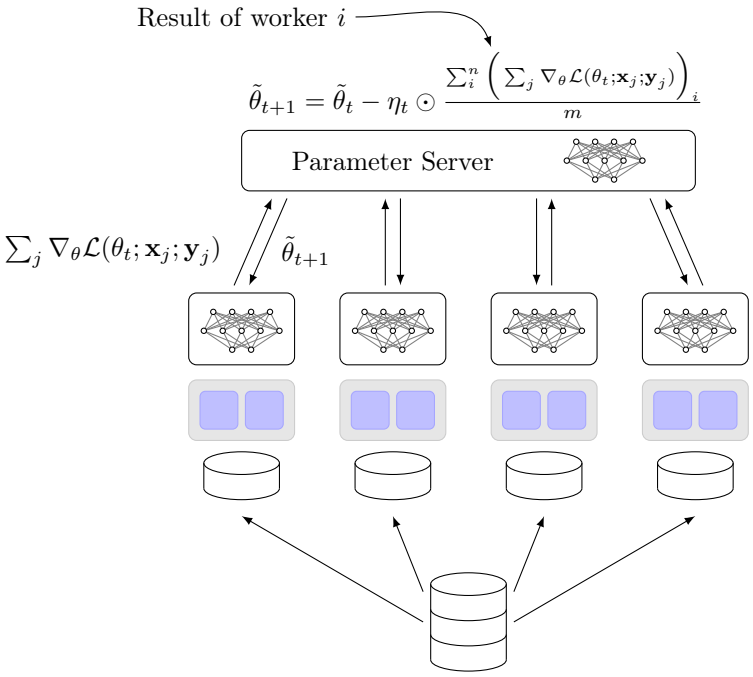


Figure 1.3: Distributed mini-batch data parallelism with n parallel workers, with a total mini-batch size of m. At the start, the data is split into n partitions, and all workers get a copy of the central model with parameterization $\tilde{\theta}_0$. When the training starts, every worker i computes the accumulated gradient $\sum_j \nabla_\theta \mathcal{L}(\theta_t; \mathbf{x}_j; \mathbf{y}_j)$ based on its part of the mini-batch, which is then *commited* (send) to the parameter server. After the parameter server receives all accumulated gradients, it averages them, and then applies the batched gradient to the model in order to produce $\tilde{\theta}_{t+1}$. After the new parametrization is available, the workers will be able to *pull* (download) $\tilde{\theta}_{t+1}$, and repeat the procedure described above.

This of course begs the question, *can we eliminate the need for a locking mechanism to prevent unnecessary waits of the workers*? There are some significant initial counter-arguments against removing the synchronization barrier. The most profound issue by removing the synchronization barrier, and thus obtaining *Asynchronous Data Parallelism*, is that the parameter server will incorperate gradients using a simple queuing model (FIFO) based on older parameterizations of the central variable, as shown in Figure 1.4.

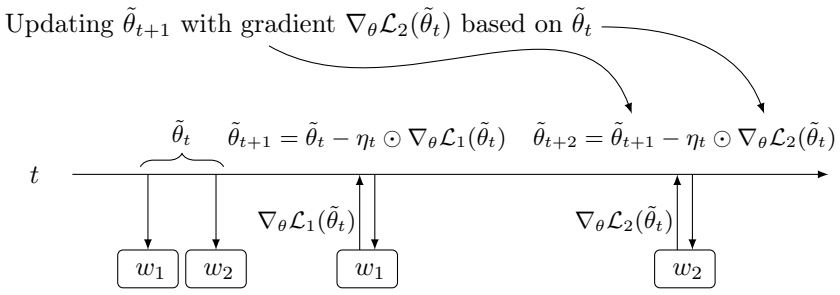


Figure 1.4: In Asynchronous Data Parallelism workers compute and commit gradients to the parameter server asynchronously. This has as a side-effect that some workers are computing, and thus committing, gradients based on old values. These gradients are called *stale gradients* in literature. In this particular example there are 2 workers w_1, and w_2. At the start optimization process, the pull the most recent parameterization from the parameter server $\tilde{\theta}_t$. Now all workers start computing the gradients asynchronously based on the pulled parametrization. However, since the parameter server incorperates gradients into the center variable asynchronously as a simple queuing (FIFO) model, other workers will update the center variable with gradients based on an older value, as shown in the figure above. Finally, assuming that the computing cluster is homogeneous, we can derive from this figure that the expected staleness of a gradient update is $\mathbf{E}[\tau] = (n-1)$.

However, experiments have shown that removing the synchronization barrier actually allows models to converge [4, 17, 5], even when most workers update the central variable using a gradient based on an outdated parameterization of the central variable. An other issue, which only has been formalized recently, is *Implicit Momentum* or *Asynchrony Induced Momentum* [13]. The main reasoning behind implicit momentum, which will be discussed in detail in Section 2.3.3, is based on a very simple but powerful idea. The idea states that *memory arizes from asynchrony*. Intuitively, this implies that "memory" of previous gradients is preserved due to *stale gradient* updates. This can be observed directly from Figure 1.4, where the update of w_2 is actually updating the central variable with a gradient identical to $\nabla_\theta \mathcal{L}_1(\tilde{\theta}_t)$ if we assume that both workers computed the gradient based on the same input data. This is a clear indication that asynchronous data parallelism is *implicitly* (because of the asynchronous nature of the approach) adding *momentum* which is proportional to the number of workers, since adding more workers actually *adds* more "memory" about previous parameterizations. The authors formalize this by probabilistically estimating the expected change between $\tilde{\theta}_{t+1}$ and $\tilde{\theta}_t$. Using some additional additional assumptions, such as the expected staleness $\mathbf{E}[\tau] = (n-1)$, and geometrically distributed staleness, the authors are able to formalize the expected update in an asynchronous setting between update t and $t+1$, which is shown in Equation 2.1.

$$\mathbf{E}[\tilde{\theta}_{t+1} - \tilde{\theta}_t] = \left(1 - \frac{1}{n}\right)\mathbf{E}[\tilde{\theta}_t - \tilde{\theta}_{t-1}] - \frac{\eta}{n}\mathbf{E}\nabla_\theta \mathcal{L}(\tilde{\theta}_t; \mathbf{x}; \mathbf{y}) \tag{1.1}$$

Using Equation 2.1, we can immediatly derrive the term which describes the implicit momentum induced by asynchrony, which is $\left(1 - \frac{1}{n}\right)$. This result actually suggests that there is a limit to asynchronous optimization: since every problem has some optimal momentum term, which implies that there is an optimal number of asynchronous workers for a specific problem. In order to push the limits of asynchronous optimization, the authors propose various techniques to reduce the abundant amount of implicit momentum. One approach is to apply a *grid-search* to find the optimal hyperparameterization for a given epoch, this also includes the number of workers. Despite that this technique finds the optimal hyperparameterization for a given epoch, the disadvantage is that after every fixed number of iterations, a grid-search of the hyperparameters has to be performed to ensure (optimal) convergence. This is actually in accordance with training in non-data parallel settings, where one starts with a smaller momentum hyperparameter because the gradients at the start will be relatively large compared to the gradients near an optimum, where usually one benifits from a larger momentum hyperparameter. From this intuition we can actually deduce that when the gradient updates are large compared to the gradients close to an optimum, an optimizer does not benifit from high parallelism because the

workers are committing gradients which were based on a parametrization which is "far" from the current central variable, thus obtaining implicit momentum. Furthermore, one could eliminate the need for the gridsearch by constructing a distributed optimization scheme which is based on the idea described in Hypothesis 1.

Hypothesis 1 (H1): *Workers only contribute efficiently to the central objective when they commit gradients which are based on variables close to the central variable.*

This implies that in the precense of high parallelism, only gradient updates which are based on variables close to the *current* central variable matter. This intuition is strengthened in Figure 1.5, where a straggler is causing the central variable to converge to a different solution as opposed to the one it was heading to first. A lot of methodologies have been suggested to handle the stragler problem, however, most approaches suggest a synchronous bounded staleness approach [3, 6]. As a result, the error introduced by staleness is limited [6]. Nevertheless, in gradient-based approaches, the straggler problem can be approached from a different perspective. By incorperating Hypothesis 1, we could eliminate additional engineering efforts since stale updates, and thus straglers, are built in the optimization procedure.

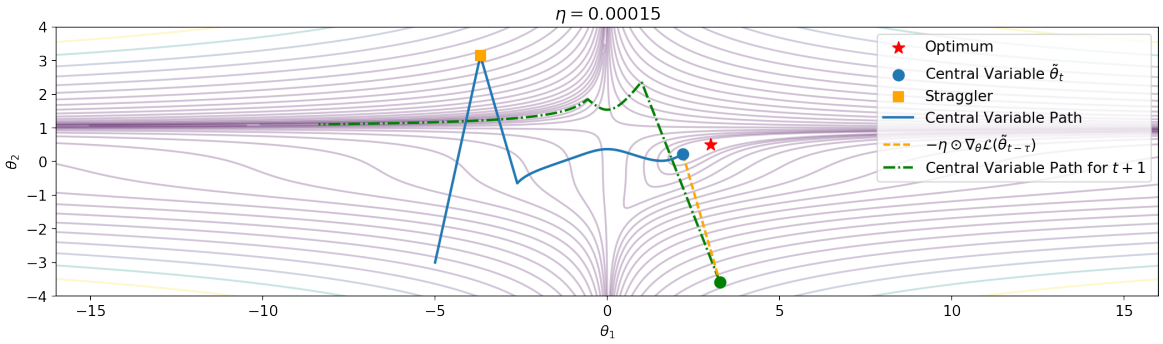


Figure 1.5: Asynchronous optimization procedure applied to Beale's function. In this experiment we introduce a straggler programatically (square) at the start of the optimization process. Despite the fact that this is a low probability event (large staleness compared to the number of workers) in real-world situtations, the effect we describe here is present in any form of asynchronous optimization. When the straggler in this figure *commits* its gradient update to the parameter server, the central variable $\tilde{\theta}_t$ will be updated using $-\eta_t \odot \nabla_\theta \mathcal{L}(\tilde{\theta}_{t-\tau})$ with staleness τ to form $\tilde{\theta}_{t+1}$. This update causes the central variable to converge to a different optimum, and additionially, increases the error of the central variable (green circle). Furthermore, to actually converge to the other optimum, additional computational work has to be performed. This situation drives our intuition presented in Hypothesis 1.

One could of course argue, why not use a smaller number of workers since we are annealing the gradients which are based on non-local variables anyway, thus wasting computational resources on those machines? This is certainly a valid argument. However, let us first consider the hyperparameter grid-search approach suggested by [13]. As mentioned above, despite the fact that the grid-search technique will find the optimal hyperparameters for the current parameterization, it doesn't mean that these hyperparameters are still optimal during the duration of the training process. Furthermore, to actually obtain the optimal hyperparameters, some certain amount of time needs to be spent in order to find them. This is actually quite problematic, since one actually wishes to reduce training time by applying data parallel techniques. In our approach, which will be discussed extensively in Chapter 4, this is not the case since the gradients will be annealed dynamically based on the curvature of the error space and current parametrization of the central variable, i.e., there is no need for a hyperparameter grid-search during the training process.

Now some general approaches and important issues regarding data parallelism have been addressed, and the reader has gained some intuition on the subject, we can formalize data parallelism and use the notation in the following chapters. In order to formalize 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.

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 [17], actually require that the complete worker parametrization θ_t^k is sent to the parameter server. To abstract this specific optimizer detail, we denote the variable that is sent to the parameter server by v. This procedure is described in Algorithm 1 and 2.

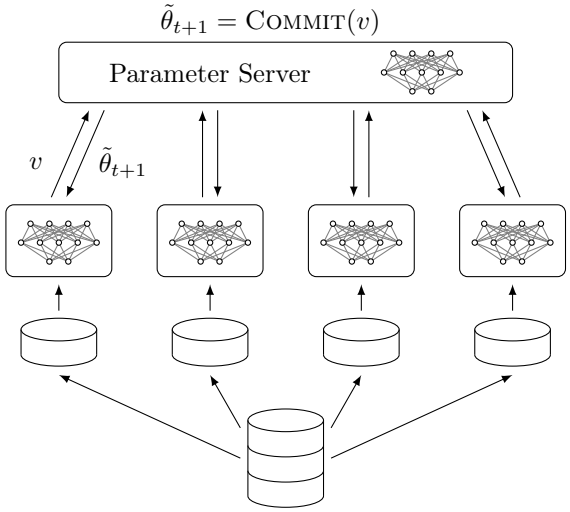


Figure 1.6: Schematic representation of a data parallel approach. In this methodology we spawn n workers (not necessarily on different machines), and assign a data shard (partition) of the dataset to every worker. Using this data shard, a worker i will iterate through all mini-batches to produce a gradient, $\nabla_\theta \mathcal{L}_i(x)$, for every mini-batch $\mathbf{m}$. Next, a variable v is constructed on the worker which is send to the parameter server. The parameter server will incorperate the variable using a COMMIT mechanism to produce the next central parametrization $\tilde{\theta}_{t+1}$.

Nevertheless, the most common and one of the earliest asynchronous optimization algorithm is DOWNPOUR [4]. In this Master Thesis, we use DOWNPOUR as a baseline performance indicator for all our experiments with other distributed optimization algorithms. In essence, workers are continuously committing gradients to the parameter server using Equation 1.2. After a gradient has been committed to the parameter server, the worker will *pull* the most recent parameterization from the parameter server in order to be consistent with the updates of other workers, as in Algorithm 1 and 2.

$$\Delta\theta^k = -\eta_t \odot \nabla_\theta \mathcal{L}(\tilde{\theta}_{t-\tau}; \mathbf{x}; \mathbf{y}) \tag{1.2}$$

Once the parameter server received the update $\Delta\theta^k$ from worker k, the parameter server will simply add (since the worker already negated the gradient) the update to the current central variable in order to produce $\tilde{\theta}_{t+1}$, this is described by Equation 1.3.

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t + \Delta\theta^k \tag{1.3}$$

Furthermore, in order to examine the scaling abilities of the optimization algorithms we discuss in the following chapters, we need a measure to express how well they are performing in a given scenario. To measure this, one could use more traditional metrics such as *statistical efficiency* and *hardware efficiency* [5, 13].

Statistical efficiency (SE) describes the number of iterations that are required to obtain a desired result. In the context of Machine Learning, statistical efficiency describes the number of model updates that have to be performed in order to acquire a desired accuracy. However, in order to obtain some metric about a specific optimization algorithm, we need a baseline to compare against. As stated above, the baseline algorithm we use in this work is DOWNPOUR. Once we evaluated an algorithm ϵ, we compute the statistical efficiency of ϵ compared to DOWNPOUR, as described by Equation 1.4.

$$\frac{SE(\text{DOWNPOUR})}{SE(\epsilon)} \tag{1.4}$$

Hardware efficiency (HE) on the other hand, describes the amount of time it takes to execute a single iteration of a loop. In our work, this denotes the time it takes to complete a *single* epoch. Nonetheless, during this work, we experimented with several optimization algorithms which actually compute several gradients locally, and preprocess them before transmitting the update to the parameter server [17]. In these cases, if someone would employ statistical or hardware efficiency to obtain a performance indicator compared to an other algorithm, they will have a clear advantage since a significantly smaller number of central variable updates occur. Furthermore, usually these algorithms also spend less time consuming all the data since parameter server updates occur less frequent. Moreover, the network is also less saturated due to the reduced number of parameter server updates. In order to have a non-biased metric among different distributed optimization algorithms, we should look at the time it takes to obtain a desired accuracy. We call this *temporal efficiency*.

Temporal efficiency (TE) characterizes the amount of time a process, or a collection of processes, requires in order to obtain a desired accuracy. Temporal efficiency is in effect propertional to statistical efficiency, i.e., $SE(\epsilon) \propto TE(\epsilon)$. However, this is only the case when algorithm ϵ actually transmits an update to the parameter server after a worker computed a gradient (in an asynchronous setting). This is in contrast with algorithms such as [17], where some additional samples are evaluated locally, before an update is transmitted to the parameter server.

Algorithm 1 Describes the general asynchronous 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)
  $\theta_0^k \leftarrow$ PULL()
  $t \leftarrow 0$
  while not converged **do**
    $\mathbf{m} \leftarrow$ FETCHNEXTMINIBATCH(p_k)
    $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{m})$ ▷ Optimization step, could be [9], or other optimizer.
    $v \leftarrow$ PREPARECOMMIT()
    COMMIT(v)
    $\theta_t^k \leftarrow$ 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

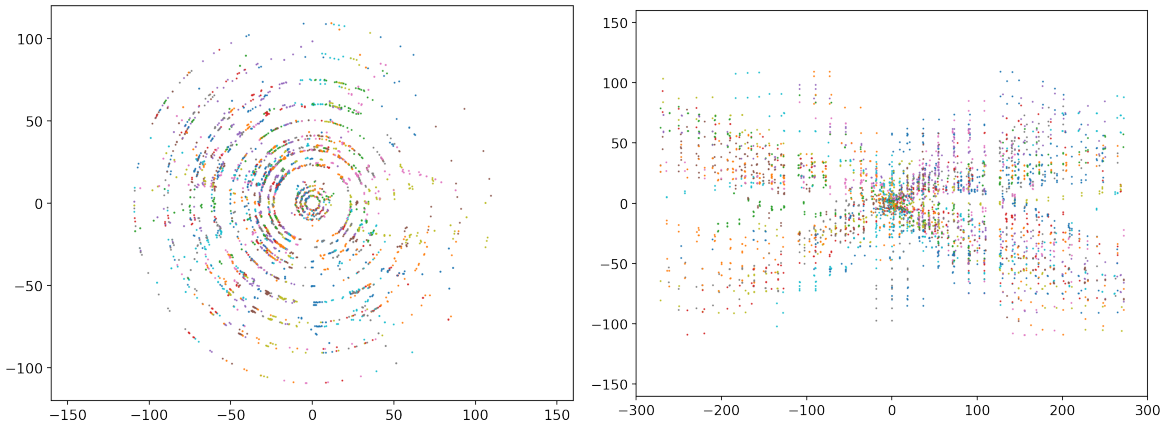
procedure PULL( )
  return $\tilde{\theta}_t$
end procedure

1.4 Problem Statement

In recent years it has been shown that being able to train large deep neural networks on vasts amount of data yield state-of-the-art classification performance. However, training these models usually takes days, or in some cases, even weeks. In order to significantly reduce the training time of these models, Jeff Dean (Google) introduced a new paradigm to train neural networks in a distributed fashion, i.e., model – and data parallelism, which is an initial attempt to tackle this problem.

Despite the relatively old research (2012), few efforts have been made to fully understand the implications, or to significantly improve the parallel gradient descent algorithm (DOWNPOUR) proposed by Dean et al. Furthermore, most research focusses on limiting the error introduced by staleness by introducing some synchronization barrier, and thus limiting the amount of asynchrony. Despite this, only recently a sound theoretical argument has been made to understand asynchronous data parallelism [13]. Using this, and understanding this contribution, we try to push the limits of asynchronous data parallelism even further.

As stated above, being able to train a model on a vast amount of data generally improves the statistical performance of a model since the model will have access to many (different) examples to train on. This is in particular the case at CERN, where the experiments collected in the order of 100 PetaBytes of particle collisions in the past years. Machine Learning approaches, and Deep Learning in particular, could potentially help in data reconstruction in the upcoming runs of LHC where particles will generate a huge amount of hits in the detector where it would be infeasible to reconstruct the particle tracks using traditional techniques (combination of a Kalman filter and Runge–Kutta methods). However, due to the petabyte scale of the data, current data parallelism will not be able to train the model in a reasonable amount of time. Therefore, we think it is important to push the current limits of data parallelism even further in order to reduce the overal training time even further.



(a) Front-view of the CMS detector.

(b) Side-view of the CMS detector.

Figure 1.7: Reconstructed particle hits in the CMS detector. The collision point (vertex) is originated at (0,0,0). The inner part of the detector is called the *pixel silicon* detector. This is a high resolution tracking mechanism which is able to handle the highly saturated environment of the inner part of the detector. The more outward part in this particular figure is the *silicon strip* detector. The silicon strip detector basically consists out of blocks with orthogonally positioned silicon strips which activate when a particle passes through them. Toghether, the strips produce a 3-dimensional coordinate. We would like to note that the hits do not actually represent the track of the particle, but it is rather a set of hits which will be used to compute the helix (track) parameters using the Runge–Kutta method.

1.5 Thesis Outline

In this chapter we introduced the main concept, and problems surrounding the parallization of gradient descent. We famillarized the reader with the topic and some notation by providing some context why someone would like to apply said technique.

Chapter 2 will discuss several distributed optimization methods, i.e., their strenghts, and pitfalls. Furthermore, we give some context in order to understand why some of these methods have been proposed, and how they perform in these situations.

Accumulated Gradient Normalization, our first contribution, appears in Chapter 3. We propose said technique to allow for local exploration in the error space to provide better updates to the parameter server. In general, this technique reduces the communication overhead, thus taking into account communication constraints which other techniques try to solve in a different way. Furthermore, in contrast to previous approaches, our approach is not sensitive to hyperparametrization.

In Chapter 4 we introduce our optimizer ADAG, or *Asynchronous Distributed Adaptive Gradients*, which is an asynchronous data parallel approach which uses the contributions from Chapter 3, and implements Hypothesis 1. We examine how *Accumulated Gradient Normalization* is assisting the optimization process to check for potential synergies.

When all theory has been presented, we validate our results and hypotheses by performing some experiments on traditional datasets such as MNIST, and CIFAR to have a baseline comparison to validate against different methods. But also on CERN-specific problems, such as track reconstruction as mentioned in Section 1.4.

Finally, we conclude the thesis in Chapter 7 by summarizing the contributions that have been made, and the empirical results from our experiments.

We use the following research questions to guide our study of parallelizing gradient descent:

1. When do workers contribute positively to the central variable during training?
2. Why does asynchronous EASGD diverge when a small communication frequency is used, and converges with a large communication frequency?
3. Why does *Accumulated Gradient Normalization* converge faster, compared to equivalently sized mini-batches?

Chapter 2

Distributed Deep Learning

In this chapter, we introduce several concepts and techniques related to Distributed Deep Learning on which this works builts upon. We start in Section 2.1 with a recap of all methods and techniques we have discussed in Chapter 1. Afterwards, we continue with a discussion of synchronous followed by an examination of asynchronous optimization methods such as DOWNPOUR and closely related extensions. Furthermore, we address several issues such as *asynchrony induced momentum* which are related to asynchronous optimization. We also consider several approaches which provide a possible solution to these issues.

2.1 Introduction

For all practical applications, Stochastic Gradient Descent (SGD) and derrivatives are the best tools from the numerical optimization toolbox for neural networks. However, applying SGD in its pure form, that is, updating the parameters after evaluating a training sample, is a computationally intensive process. An initial approach for speeding up SGD in terms of convergence with respect to training time was to compute the gradients of several samples, a *mini-batch*, and average them. This approach has several advantages, the first being that a larger mini-batch will result in less noisy updates, as more "evidence" of the surrounding error space will provide a better gradient update. The second advantage being the increased computational parallelism, since all sub-gradients (gradients of the training samples in the mini-batch) are based upon the same parametrization of the model. As a result, the parallelization of the gradient computation is quite straightforward. For instance, for every training sample in a mini-batch, one could allocate a thread, a process, or even a different machine (see Figure 1.3) to compute the gradients in parallel. However, a blocking mechanism is required in order to sum all gradients, average them, and finally update the parametrization of the model. This process is depicted in Figure 1.2. As discussed in Chapter 1, mini-batch parallelism is an instance of *synchronous data parallelism*. Although many synchronous optimization schemes share a similar structure, we discuss other instances of synchronous data parallelism in particular in Section 2.2 since these optimization schemes incorperate gradients and worker parameterizations into the central variable differently compared to mini-batch parallelism.

Nevertheless, a significant, but albeit technical issue in synchronous optimization is when a single or multiple workers are slowed down for some reason, e.g., due to high CPU load, or bandwidth consumption, other workers will have to wait before they can continue with step $t + 1$. As a result, the allocated resources are not fully utilized. This particular issue is known in literature as the *straggler* problem. However, this problem can be mitigated with by using a *homogeneous* hardware configuration. For instance, when one would employ 2 different GPU's running at different clock speeds, a *heteregenous* hardware configuration, then the CPU will always have to wait for a particulur GPU since it runs at a lower clock speed causing the complete training procedure to be slowed down[1]. Furthermore, we could argue that there is a limit to synchronous data parallelism because simply *adding more workers to the*

[1] A chain is only as strong as its weakest link.

problem implicitly increases the size of the mini-batch. As a result, when applying synchronous data parallelism, one is not parallelizing gradient descent in a typical sense, but rather parallelizing the computations within a step. Of course, one could even increase the parallelism within synchronous data parallelism even further by applying model parallelism as dicussed briefly in Section 1.2. Nevertheless, while such an implementation is definitly possible, it might be more cost-aware from an economical perspective to just let the model train for a longer period of the compared to actually implementing the training procedure described above. Furthermore, even with if one would implement said training method, there is still a limit to the amount parallelism due to the structure of the computation graph, and communication cost between devices which have to be taken into account. This of course begs the question if it is actually possible to push the limits of asynchrony, and thereby reducing the training time even further. Or from a different perspective, is there a more trivial method besides implementing the above training procedure to reduce the training time.

Several approaches [4, 6, 3, 15, 17, 11, 8] have been suggested over the past years which accomplish exactly this. All these methods are instances of *asynchronous data parallelism*, discussed in Section 1.3. In contrast to synchronous data parallelism, asynchronous methods can be identified by the *absence* of a blocking mechanism which is present in synchronous data parallelism. Despite the fact that this method resolves the waiting time induced by stragglers, it introduces a closely related but persistent issue. More formally, the *staleness* issue is due to the fact that all n workers update the central variable in an asynchronous fashion. Meaning, from the moment a worker k is done computing an update $\Delta\theta^k$ based upon parameterization of the central variable $\tilde{\theta}_t$, it will commit $\Delta\theta^k$ to the parameter server, and afterwards continue with the next mini-batch. Because of this behaviour, it is possible that a number of central variable updates τ occurred during the time worker k was computing $\Delta\theta^k$. As a result, instead of obtaining $\tilde{\theta}_{t+1}$ by applying $\Delta\theta^k$, worker k is actually applying $\Delta\theta^k$ to $\tilde{\theta}_{t+\tau}$. Which is not ideal, since $\Delta\theta^k$ is based on parametrization $\tilde{\theta}_t$. From [13] we know that increasing the number of workers actually increases the amount of staleness τ since $\mathbf{E}[\tau] = (n-1)$ under a *homogeneous* hardware configuration and a simple queuing model[2]. This result is validated empirically in one of our experiments, shown in Figure 2.1.

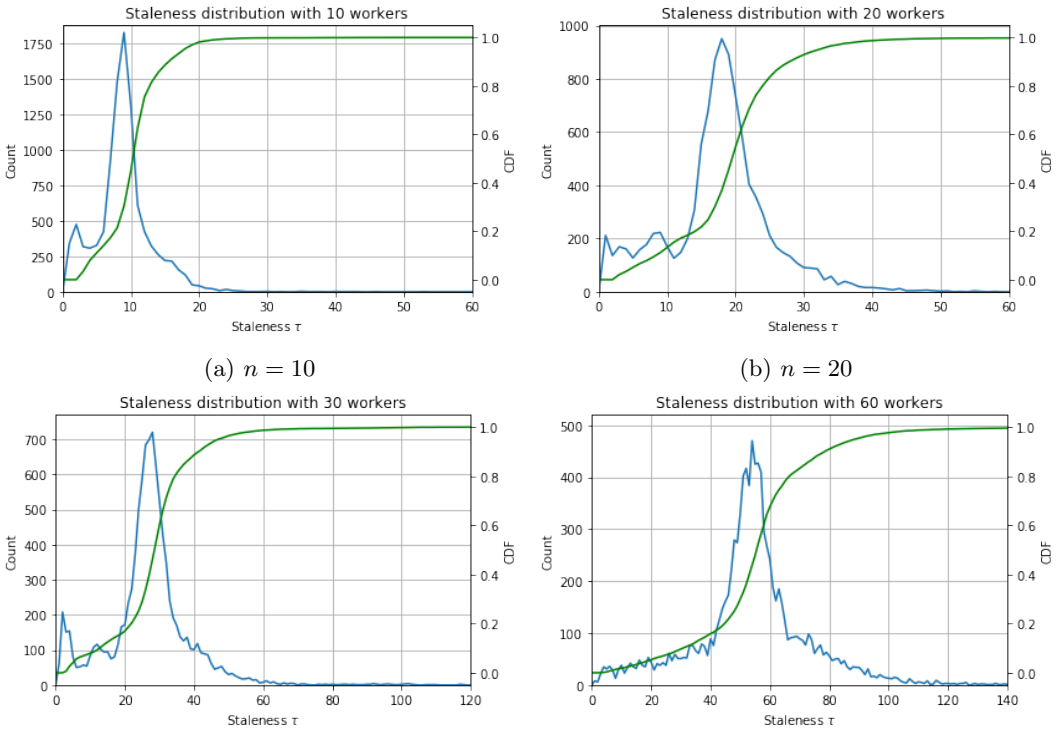
A side-effect of updating the central variable with stale updates in an asynchronous fashion, is that stale updates carry information about previous states of the central variable. Which is to be expected since worker updates are based on older parameterizations of the central variable. Using this intuition, the authors in [13] show formally that in a regular asynchronous SGD setting, like DOWNPOUR, stale updates behave like *momentum*. Furthermore, their formalization can even describe the amount of *implicit momentum*, described in Equation 2.1, which is present in an asynchronous optimization procedure. Furthermore, when applying (Nesterov) momentum in a traditional optimization setting, i.e., sequential parameter updates, one needs to specify the amount of momentum. This is usually denoted by a hyperparameter μ. However, we would like to note that in an asynchronous setting, the hyperparameter μ_s from Equation 2.1, is not explicitly defined in the optimizer, but arises from the number of asynchronous workers. As a result, Equation 2.1 is merely descriptive.

$$\mu_s = \left(1 - \frac{1}{n}\right) \tag{2.1}$$

In a previous paragraph we said that there is a limit to mini-batch parallelism, since adding more workers to the problem implicitly increases the size of a mini-batch. However, we observe in Figure 2.1 in accordance with [13], that there might be a limit to asynchronous optimization as well. However, the authors in [13] assume that gradients coming from the workers are not adaptive[3], as can be detucted from their proof. The question begs, can we push asynchronous optimization even further? We answer this question in Chapter 3, and Chapter 4, by introducing new techniques using a better, and more intuitive understanding of parameter staleness.

[2]With a simple queuing model we intent that updates $\Delta\theta^k$ are incorperated into the central variable in a queuing fashion.

[3]Meaning, they are not modified with respect to some (hyper)parameter.



(a) $n = 10$ (b) $n = 20$

(c) $n = 30$ (d) $n = 60$

Figure 2.1: These figures show the staleness distribution during a training procedure using a differing number of parallel workers. For every central variable update, we record the staleness τ, and increment the number of occurences of this particular staleness by 1. Thus effectievely building a histogram showing the staleness distribution during the training. With this, we experimentally validate the observations of [13] that $\mathbf{E}[\tau] = (n-1)$. Furthermore, the claim that staleness is geometrically distributed during training also holds (right half of the distribution).

2.2 Synchronous Data Parallelism

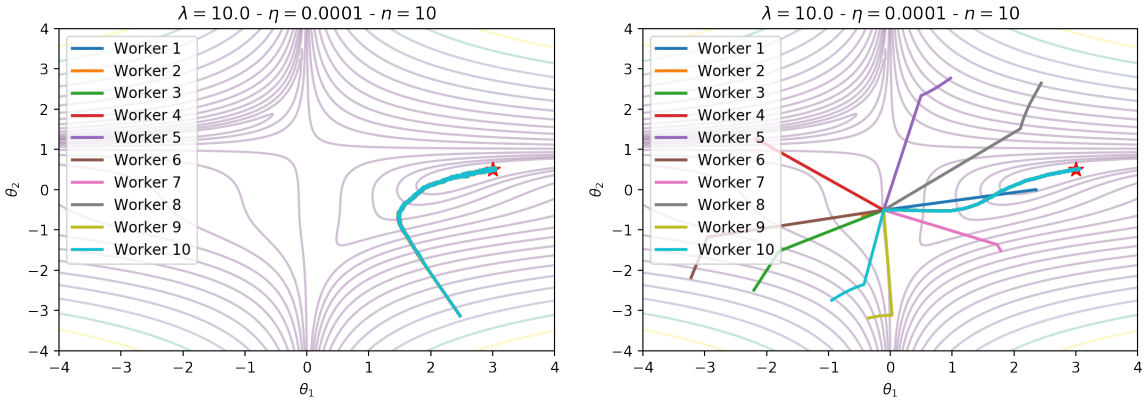
2.2.1 Model Averaging

As the name suggests, model averaging optimizes the central variable by simply averaging the parametrizations of the workers after λ (which could be the amount of data in a single epoch) steps until all data has been consumed. As mentioned before, hyperparameter λ denotes the number of *local* steps that have to be performed before the results are communicated with the parameter server. The optimization procedure in the worker and parameter server are quite straightforward, and are described in Equation 2.2 and Equation 2.3 respectively. However, Equation 2.2 has the disadvantage that a lot of communication with the parameter server has to be performed, since after every *local* worker update all parameters have to be shipped to the parameter server to apply Equation 2.3. Furthermore, this approach has several other issues that will be discussed later. But for now we can resolve this issue by delaying a commit to the parameter server by doing several *local* (λ) updates before sending θ_t^k to the parameter server, this is shown in Algorithm 3.

$$\theta_{t+1}^k = \theta_t^k - \eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) \tag{2.2}$$

$$\tilde{\theta}_{t+1} = \frac{1}{n} \sum_{i=1}^{n} \theta_t^i \tag{2.3}$$

Contrary to other synchronous methods, model averaging does not reduce the training time in general. In fact, it requires more resources to achieve the same results since the central variable is set to be the average of all workers. This is shown in Figure 2.2 (a) since all workers follow the same first-order path, synchronize, and average the parameters after λ steps to start again from the averaged parameterization, which is in this case the central variable. However, what happens if we initialize the parameterizations of the workers randomly? At first, all workers will do some work locally, but after λ steps, the parametrizations of the workers are averaged. As a result, all workers share the same parameterization in the next step, which brings us again to our initial scenario as shown in Figure 2.2 (b). Furthermore, when applying random initialization, we could say a "warmup" period is required since all workers need to converge a particular solution before convergence can take place. This intuition is strenghtened in Figure 2.3.



(a) Identical initialization

(b) Random initialization

Figure 2.2: In this Figure we show the difference between identical initialization (a), and random initialization (b). In essence, both methods require roughly the same amount of time as a sequential optimization algorithm, i.e., not distributed, while using more resources. However, the difference here is that using random initialization requires a "warmup" period (a single parameter server update), before the actual optimization process can start. In order to simulate the stochastic noise of mini-batch gradient descent, we added a noise term to our gradient computations which was sampled from $X \sim \mathcal{N}(\mu = 0, \sigma^2 = 10.0)$ to ensure that some deviation from the central variable was possible.

Algorithm 3 Describes the worker procedure with local worker exploration. The worker will be identified with a certain index k and will be initialized by assigning the central variable ($\tilde{\theta}_t$) to the worker, or the worker variable can be randomized at the start of the optimization procedure. Furthermore, we introduce a hyperparameter λ, which is the number of local updates that have to be performed in order the worker parameterization θ_t^k is communicated with the parameter server.

procedure ModelAveragingWorker(k)
  $\theta_0^k \leftarrow$ Pull() or Random() ▷ Worker variable θ_0^k can be randomized.
  $t \leftarrow 0$
  while not converged **do**
    $i \leftarrow 0$
    for $i < \lambda$ **do**
      $\mathbf{x}, \mathbf{y} \leftarrow$ FetchNextMiniBatch()
      $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ ▷ Optimization step, could be [9], or other optimizer.
      $i \leftarrow i + 1$
      $t \leftarrow t + 1$
    end for
    Commit(θ_t^k)
    WaitForOtherWorkers()
    $\theta_t^k \leftarrow$ Pull()
  end while
end procedure

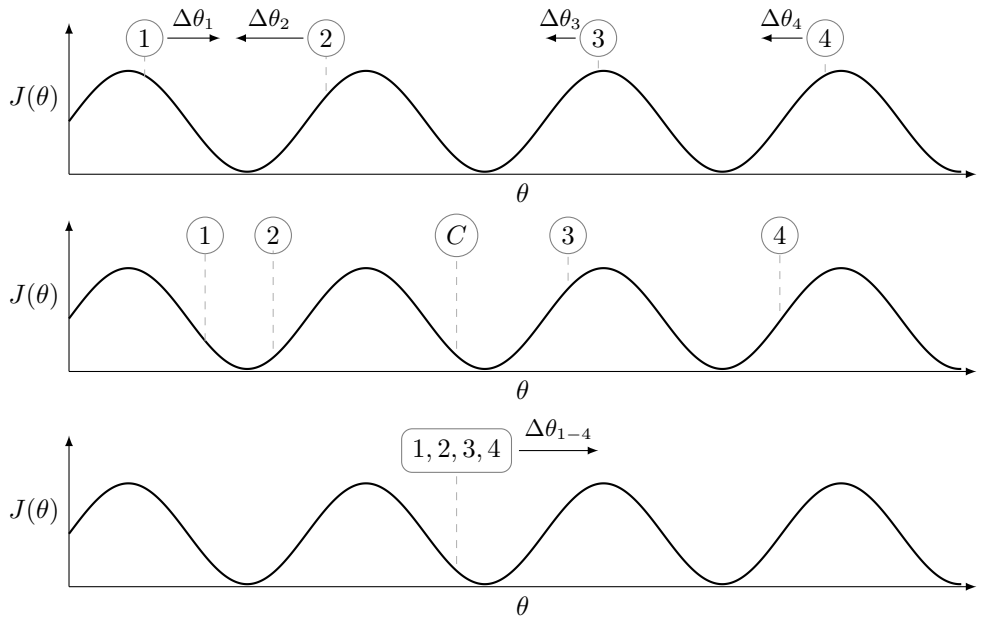
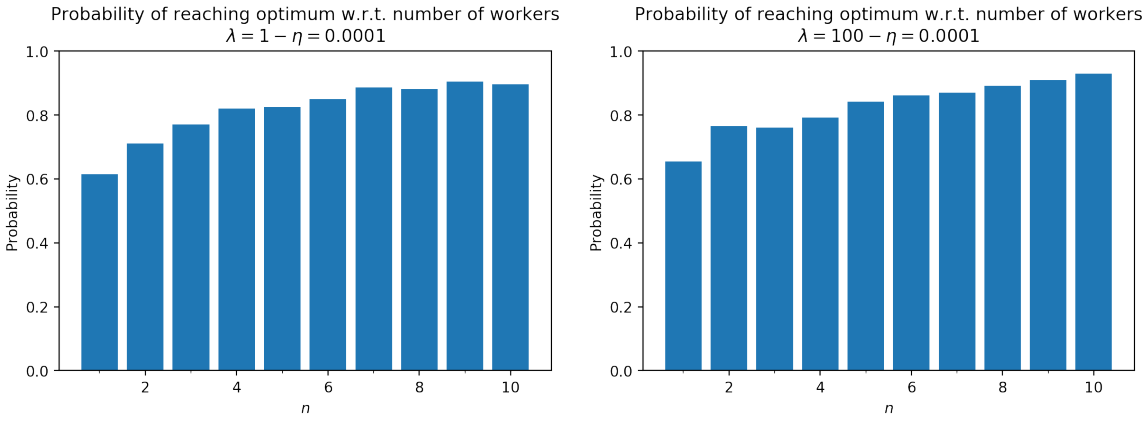


Figure 2.3: This figure explains the intuition behind model averaging. In the first state, all workers, w_1, w_2, w_3, and w_4, are uniformly initialized over the hypothesis space. Using the local parametrizations, every worker obtains an update Δw_i, and applies it locally. Afterwards, all workers send their parametrizations to the parameter server which will average them to obtain a central variable, which is depicted by C in this particular figure. Finally, all workers fetch the most recent central variable, and start computing new gradients based for the following iteration. Furthermore, what can be observed directly from this figure, is that when the workers do not agree on a *local neighborhood*, the central variable will not be able to converge. This is additional support for Hypothesis 1.

Nevertheless, what is interesting about the random initialization of workers, is that when we increase the number of workers, the probability that we will find a better solution (minima) compared to a sequential optimization process increases. However, this also depends on the curvature of the error space. For example, in Figure 2.4 we use Beale's function to obtain our statistic. However, from the plots we can deduce that the curvature of the error space is slightly biased towards the global minimum if first-order gradients are used. If the error space wasn't the case, the statistic for a single worker under different hyperparameterizations should be 50%.



(a) $\lambda = 1$ (b) $\lambda = 100$

Figure 2.4: Probability distribution extracted from several Monte-Carlo simulations under different conditions. We find that the probability of reaching the optimum (Beale's function) increases when the number of random initialized workers increases. Despite the fact that we observe that more exploration (λ) yiels a better statistic, we believe that this result is not significant due to the relatively low number of simulations (1000 per worker per hyperparameter).

2.2.2 Elastic Averaging SGD

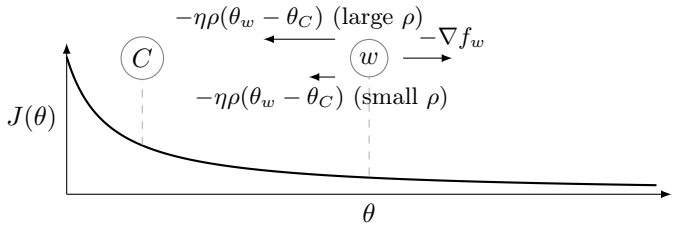


Figure 2.5: EASGD Caption here

2.3 Asynchronous Data Parallelism

2.3.1 DOWNPOUR

2.3.2 Dynamic SGD

2.3.3 Asychrony Induced Momentum

2.3.4 Asynchronous Elastic Averaging SGD

2.4 Hybrids

Chapter 3

Accumulated Gradient Normalization

$$\Delta\theta = -\frac{\sum_{i=0}^{\lambda} \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_\theta \mathcal{L}(\theta_i; \mathbf{x}_{ij}; \mathbf{y}_{ij})}{\lambda} \tag{3.1}$$

$$\lim_{\lambda \to \infty} -\frac{\sum_{i=0}^{\lambda} \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_\theta \mathcal{L}(\theta_i; \mathbf{x}_{ij}; \mathbf{y}_{ij})}{\lambda} \tag{3.2}$$

Chapter 4

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 [4, 5, 9, 17] and incorperate new insights backed up by theory and experimental evidence. We start in Section 4.1 by formalizing the problem setting. In Section 4.2, we summarize previous work on distributed (data parallel) optimization. Section 4.3 will describe our algorithm in detail, supported by intuition and theory. Finally, we experimentally show the effectiveness of our approach in Section 4.4 and give some points for future work in Section 4.5.

4.1 Problem setting

4.2 Previous work

4.3 Algorithm

4.3.1 Update rule

4.4 Experiments

4.4.1 Handwritten digit classification

4.4.2 Higgs event detection

4.4.3 Sensitivity to hyperparameters

4.4.4 Sensitivity to number of parallel workers

4.5 Future work

Chapter 5

Distributed Keras

Chapter 6

Experiments

This chapter describes the experiments, and their results, we conducted to obtain emperical evidence for our hypotheses. We start by applying our theories to a well-studied problem such as MNIST [10]. Afterwards we apply our knowledge to CERN specific problems, such as track reconstruction and event identification.

6.1 Handwritten Digit Classification

6.2 CMS Track Reconstruction and Event Identification

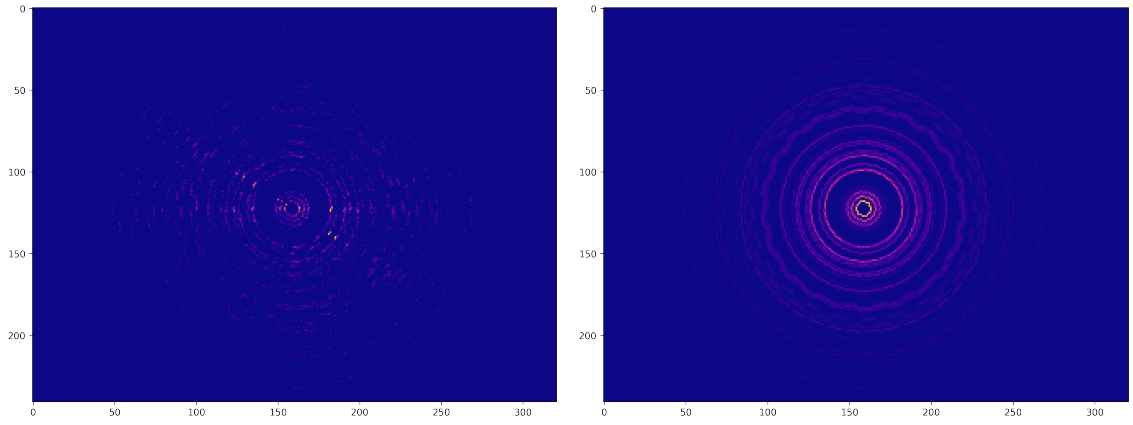
To reduce the computational load of collision reconstructions in future LHC runs, the CMS experiment is exploring Machine Learning techniques as a possible approach for accomplishing this. In this particular case, the experiment is evaluating Deep Learning techniques by fitting them to physics problems, more high-level problems such as deciding which data to keep in the High Level Trigger, and other inference problems. In the following experiments, we mainly occupy ourselves with the reconstruction of *particle tracks*, and identification of *track types* from raw detector *hits*. A particle track, *track* denoted from this point on, is the path that has been traversed by a particle through the detector. The track is reconstructed from a set of hits, which have been triggered (detected) by parts of the detector. The reconstruction of these tracks is a computationally intensive process, since given a set of hits, one needs to minimize the χ^2 error of the track with respect to the set of hits that have been associated with a particular track. However, this is only one aspect of the problem, one first needs to obtain the set of hits which describe a track given all hits within a collision, which also includes the background (false-positives) generated by the detector itself. Currently, the set of hits related to a single track are extracted using a two-pass Kalman filter, with the first pass starting on the outer edges of the detector.

An additional problem of applying Deep Learning, or any other Machine Learning approach to the problem of track reconstruction, or event identification, is the petabyte scale data that needs to be dealth with. A simple, and more common solution would be to sample a more managable fraction of the dataset to train our models on. However, we want the model to be able to extract as many diverse tracks as possible that have been reconstructed over previous years by the reconstruction software. As a result, a distributed (data parallel) approach is necessary. However, the data representation is also an important aspect of the problem. Currently, all collisions are stored in the ROOT format, where every reconstructed track for a particular collision (or set of collisions), including the hits of the track, and track parameters can be extracted from. This specific data format is quite problematic, especially taking the petabyte-scale data into account. Furthermore, depending on the modelling, the data needs to be preprocessed in a particular way. One could of course preprocess the complete physics data in a format and shape the models accept, and simply copy the data to the machines which will be used during training. However, this is not a very space efficient approach.

A more reasonable, and space efficient approach would be to deliver the preprocessed data to the training procedure in a streaming manner. For example, every worker has a buffer in which it will prefetch and preprocess data coming from the ROOT format in a way the model will understand, e.g., NUMPY arrays. This approach does not have the space inefficiencies the previous approach had. However, there is an increased computational load on the worker nodes due to the prefetching, and preprocessing of the data. Nevertheless, compared to computation of gradients, this load is neglecktable. An additional benefit of this approach is that the same architecture can be used during model development, since data representations can be generated on-the-fly.

6.2.1 Data Representation

As stated above, our problem can be described as "*given a set of hits for every collision, identify the (sub)set of hits which belong to a specific track*". In a first step, we simplified the problem as: *given a set of hits of a collision, identify the types of tracks that occurred in said collision.* Using this simplified version of the problem statement, we could experiment with data representation, and model topology quite easily, without having to think about the modelling of the tracks. Our initial idea was to construct a three-dimensional "image" of the detector, where the image is basically a discretized heatmap of the hits in the detector. However, in order to maintain the required resolution in the inner part of the detector, without sacraficing dimensionality of the parameterization of the model, we had to limit the data representation to a two-dimensional approach. In this two-dimensinal approach, we construct two *feature matrices*. One matrix represents the front-side of the detector, along the beam-axis, and the other, perpendicular to the beam-axis, represents the side of the detector. Since every hit is a three-dimensional coordinate, constructing these matrices is quite straightforward.



(a) Subfigure 1 (b) Subfigure 2

Figure 6.1: Pixel-binned data representation of the hits which occurred in two different collisions.

6.2.2 Model Development

Chapter 7

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] James Cipar et al. "Solving the Straggler Problem with Bounded Staleness." In: *HotOS*. 2013, pp. 22–22.

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

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

[6] Qirong Ho et al. "More effective distributed ml via a stale synchronous parallel parameter server". In: *Advances in neural information processing systems*. 2013, pp. 1223–1231.

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

[8] Jiawei Jiang et al. "Heterogeneity-aware distributed parameter servers". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 463–478.

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

[10] Yann LeCun, Corinna Cortes, and Christopher JC Burges. *The MNIST database of handwritten digits*. 1998.

[11] Gilles Louppe and Pierre Geurts. "A zealous parallel gradient descent algorithm". In: (2010).

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

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

[14] 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).

[15] Benjamin Recht et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 2011, pp. 693–701.

[16] 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).

[17] 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