

# 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 promotores, 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

<b>Preface</b>	ii
<b>Abstract</b>	iii
<b>Summary</b>	iv
<b>Abbreviations and Notation</b>	vii
<b>1 Introduction</b>	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
<b>2 Related Work</b>	12
2.1 Introduction . . . . .	12
2.2 Synchronous Data Parallelism . . . . .	15
2.2.1 Model Averaging . . . . .	15
2.2.2 Elastic Averaging SGD . . . . .	18
2.3 Asynchronous Data Parallelism . . . . .	24
2.3.1 DOWNPOUR . . . . .	24
2.3.2 Dynamic SGD . . . . .	26
2.3.3 Asynchronous Elastic Averaging SGD . . . . .	29
2.4 Hybrids . . . . .	31
2.4.1 Stale Synchronous Optimizers . . . . .	31
<b>3 Accumulated Gradient Normalization</b>	32
3.1 Concept and intuition . . . . .	32
<b>4 Asynchronous Distributed Adaptive Gradients</b>	36
4.1 Problem setting . . . . .	36
4.2 Previous work . . . . .	36
4.3 Algorithm . . . . .	36
4.3.1 Update rule . . . . .	36
4.4 Experiments . . . . .	36
4.4.1 Handwritten digit classification . . . . .	36
4.4.2 Higgs event detection . . . . .	36
4.4.3 Sensitivity to hyperparameters . . . . .	36
4.4.4 Sensitivity to number of parallel workers . . . . .	36
4.5 Future work . . . . .	36

<b>5 Experiments</b>	<b>37</b>
5.1 Distributed Keras . . . . .	37
5.2 Handwritten Digit Classification . . . . .	37
5.3 CMS Track Reconstruction and Event Identification . . . . .	37
5.3.1 Data Representation . . . . .	38
5.3.2 Model Development . . . . .	38
<b>6 Conclusion</b>	<b>39</b>
<b>References</b>	<b>40</b>

# Abbreviations and Notation

$\eta$	Static learning rate
$\eta_t$	Learning rate with respect to time $t$ .
$\lambda$	Communication period, or frequency of commits to the parameter server.
$\mathcal{L}(\theta ; \mathbf{x} ; \mathbf{y})$	Loss function with respect to parametrization $\theta$ , input $\mathbf{x}$ , and expected output $\mathbf{y}$ .
$\tau$	Staleness
$\theta_t^k$	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 $\theta$ .
$m$	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 parallelization of gradient descent. We familiarize 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 severely 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 benefit 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 beneficial 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 hyperparameters (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 Parallelism 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  $\sigma$  is the activation function of the output neuron.

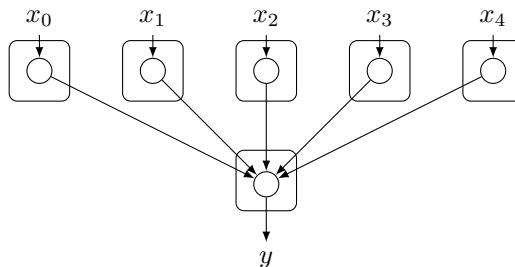


Figure 1.1: A perceptron partitioned using the *model parallelism* paradigm. In this approach every input node is responsible for accepting the input  $x_i$  from some source, and multiplying the input with the associated weight  $w_i$ . After the multiplication, the result is sent to the node which 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<sup>1</sup>. The workers are coordinated in such a way that they optimize the parametrization of a central model, which we denote by  $\bar{\theta}_t$ . 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 consistency 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  $\theta_{t+1}$ , 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.

<sup>1</sup>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 parallelization 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  $\theta_{t+1}$  of the model to the workers because the parameter server can only compute the  $\theta_{t+1}$  once it received, and processed all accumulated gradients as shown in Figure 1.3. Yet, if one or multiple machines encounter 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 *committed* (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 parameterization 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 incorporate gradients using a simple queuing model (FIFO) based on older parameterizations of the central variable, as shown in Figure 1.4.

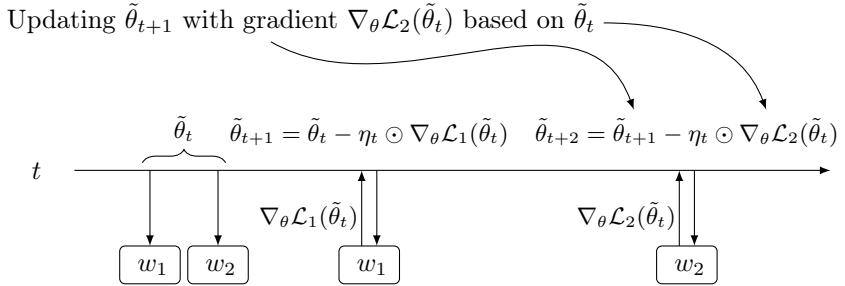


Figure 1.4: In Asynchronous Data Parallelism workers compute and commit gradients to the parameter server asynchronously. This has as a side-effect that some workers are computing, and thus committing, gradients based on old values. These gradients are called *stale gradients* in literature. In this 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 incorporates 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 ??, is based on a very simple but powerful idea. The idea states that *memory arises 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 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.

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

Using Equation 2.1, we can immediately derive the term which describes the implicit momentum induced by asynchrony, which is  $(1 - \frac{1}{n})$ . 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 benefits 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 benefit 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 presence 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 straggler 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 incorporating Hypothesis 1, we could eliminate additional engineering efforts since stale updates, and thus stragglers, are built in the optimization procedure.



Figure 1.5: Asynchronous optimization procedure applied to Beale’s function. In this experiment we introduce a straggler programmatically (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 situations, 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  $\tau$  to form  $\tilde{\theta}_{t+1}$ . This update causes the central variable to converge to a different optimum, and additionally, 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  $\mathcal{W} = \{w_1, \dots, w_n\}$ . 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 parameterization of a particular worker  $k$  at time  $t$  by  $\theta_t^k$ . 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 preparation step, before the actual training starts,  $D$  will be split into roughly  $n$  equally sized partitions  $\mathcal{P} = \{p_1, \dots, p_n\}$ , where  $|p_i| \approx \frac{1}{|D|}$ , 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  $\theta_t^k$  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.

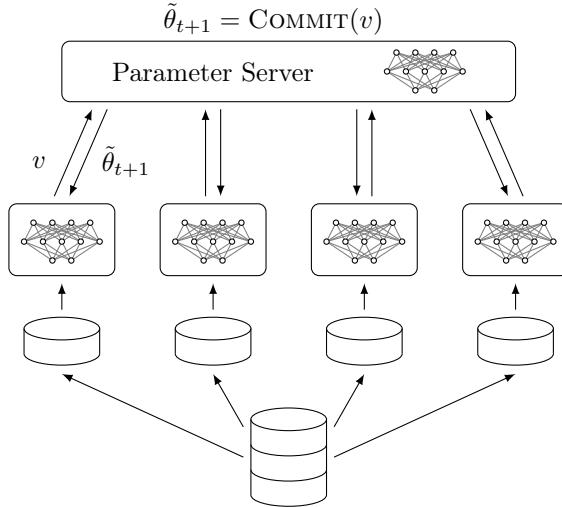


Figure 1.6: Schematic representation of a data parallel approach. In this methodology we spawn  $n$  workers (not necessarily on different machines), and assign a data shard (partition) of the dataset to every worker. Using this data shard, a worker  $i$  will iterate through all mini-batches to produce 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 incorporate the variable using a COMMIT mechanism to produce the next central parameterization  $\tilde{\theta}_{t+1}$ .

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 2.11. 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.

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

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.

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

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  $\epsilon$ , we compute the statistical efficiency of  $\epsilon$  compared to DOWNPOUR, as described by Equation 1.4.

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

**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 proportional to statistical efficiency, i.e.,  $SE(\epsilon) \propto TE(\epsilon)$ . However, this is only the case when algorithm  $\epsilon$  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  $k$ .

---

```

1: procedure WORKER( $k, p_k$ )
2:    $\theta_0^k \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $\mathbf{m} \leftarrow \text{FETCHNEXTMINIBATCH}(p_k)$ 
6:      $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{m})$        $\triangleright$  Optimization step, could be [9], or other optimizer.
7:      $v \leftarrow \text{PREPARECOMMIT}()$ 
8:     COMMIT( $v$ )
9:      $\theta_t^k \leftarrow \text{PULL}()$ 
10:     $t \leftarrow t + 1$ 
11:   end while
12: end procedure

```

---



---

**Algorithm 2** Initialization and variable handling procedures of a parameter server. Before the distributed optimization starts, the INITIALIZEPARAMETERSERVER procedure is called to initialize the local parameters, given the parametrization  $\theta$  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.

---

```

1: procedure INITIALIZEPARAMETERSERVER( $\theta$ )
2:    $\tilde{\theta}_0 \leftarrow \theta$ 
3:    $t \leftarrow 0$ 
4: end procedure
5:
6: procedure COMMIT( $v$ )
7:    $\tilde{\theta}_{t+1} \leftarrow \text{APPLYCOMMIT}(v)$ 
8:    $t \leftarrow t + 1$ 
9: end procedure
10:
11: procedure PULL()
12:   return  $\tilde{\theta}_t$ 
13: 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.

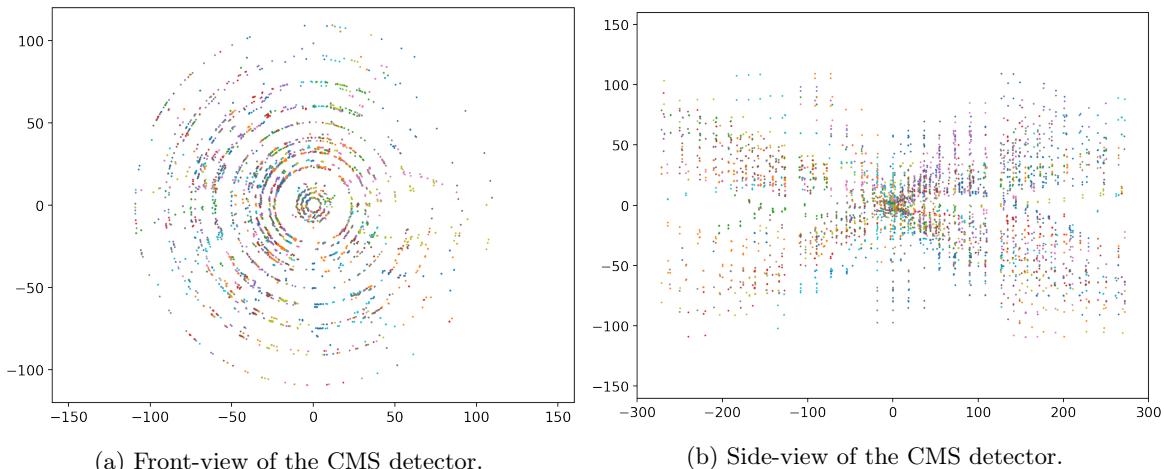


Figure 1.7: Reconstructed particle hits in the CMS detector. The collision point (vertex) is originated at  $(0,0,0)$ . The inner part of the detector is called the *pixel silicon* detector. This is a high resolution tracking mechanism which is able to handle the highly saturated environment of the inner 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 parallelization of gradient descent. We familiarized 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 strengths, 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 6 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

## Related Work

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 incorporate 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 particlur GPU since it runs at a lower clock speed causing the complete training procedure to be slowed down<sup>1</sup>. Furthermore, we could argue that there is a limit to synchronous data parallelism because simply *adding more workers to the*

---

<sup>1</sup>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 discussed briefly in Section 1.2. Nevertheless, while such an implementation is definitely 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 of 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  $\tau$  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  $\tau$  since  $\mathbf{E}[\tau] = (n - 1)$  under a *homogeneous* hardware configuration and a simple queuing model<sup>2</sup>. 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  $\mu$ . However, we would like to note that in an asynchronous setting, the hyperparameter  $\mu_s$  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.

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

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<sup>3</sup>, as can be deducted 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.

---

<sup>2</sup>With a simple queuing model we intent that updates  $\Delta\theta^k$  are incorporated into the central variable in a queuing fashion.

<sup>3</sup>Meaning, they are not modified with respect to some (hyper)parameter.

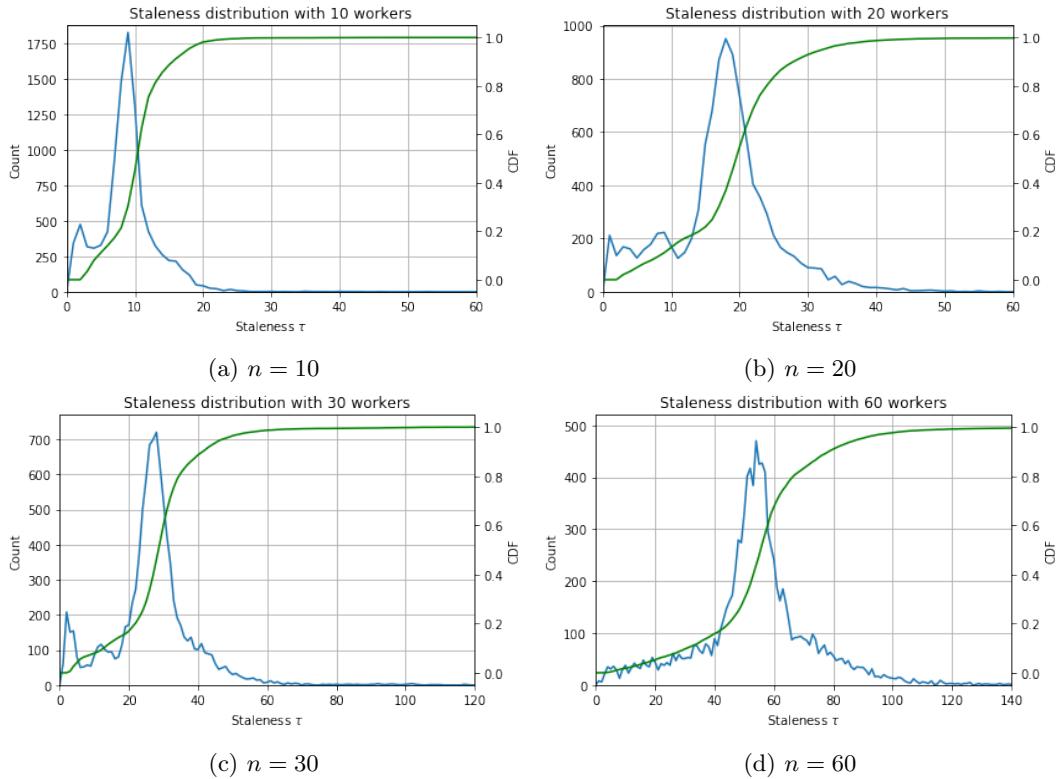


Figure 2.1: These figures show the staleness distribution during a training procedure using a differing number of parallel workers. For every central variable update, we record the staleness  $\tau$ , and increment the number of occurrences of this particular staleness by 1. Thus effectively building a histogram showing the staleness distribution during the training. With this, we experimentally validate the observations of [13] that  $E[\tau] = (n - 1)$ . 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  $\lambda$  (which could be the amount of data in a single epoch) steps until all data has been consumed. As mentioned before, hyperparameter  $\lambda$  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* ( $\lambda$ ) updates before sending  $\theta_t^k$  to the parameter server, this is shown in Algorithm 3.

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

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

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  $\lambda$  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  $\lambda$  steps, the parameterizations 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 strengthened in Figure 2.3.



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  $\lambda$ , which is the number of local updates that have to be performed in order the worker parameterization  $\theta_t^k$  is communicated with the parameter server.

---

```

1: procedure MODELAVERAGINGWORKER( $k$ )
2:    $\theta_0^k \leftarrow \text{PULL}()$  or  $\text{RANDOM}()$                                  $\triangleright$  Worker variable  $\theta_0^k$  can be randomized.
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $i \leftarrow 0$ 
6:     for  $i < \lambda$  do
7:        $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
8:        $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$      $\triangleright$  Optimization step, could be [9], or other optimizer.
9:        $i \leftarrow i + 1$ 
10:       $t \leftarrow t + 1$ 
11:    end for
12:    COMMIT( $\theta_t^k$ )
13:    WAITFOROTHERWORKERS()
14:     $\theta_t^k \leftarrow \text{PULL}()$ 
15:  end while
16: end procedure

```

---

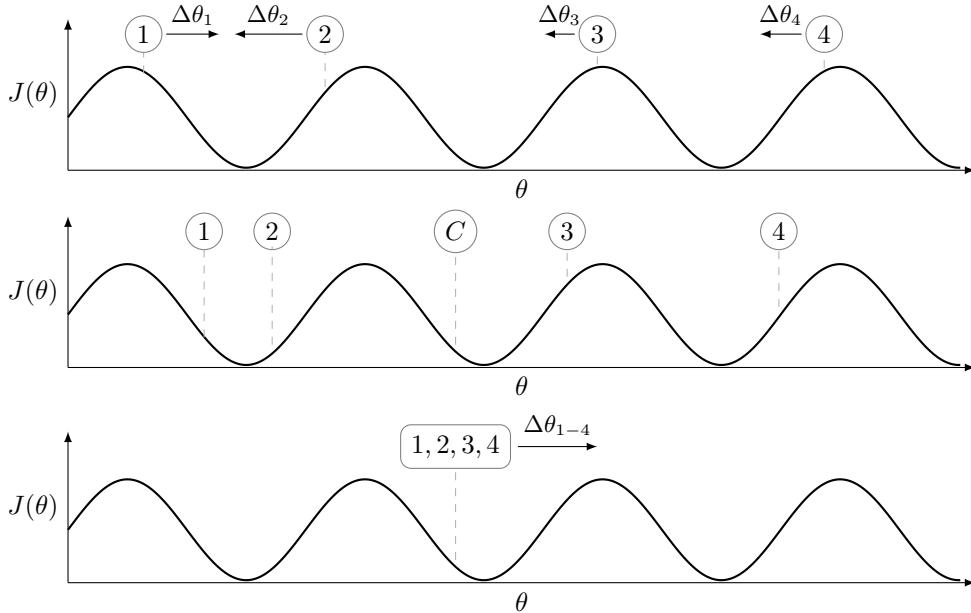


Figure 2.3: This figure explains the intuition behind model averaging. In the first state, all workers,  $w_1$ ,  $w_2$ ,  $w_3$ , and  $w_4$ , are uniformly initialized over the hypothesis space. Using the local parametrizations, every worker obtains an update  $\Delta w_i$ , and applies it locally. 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.

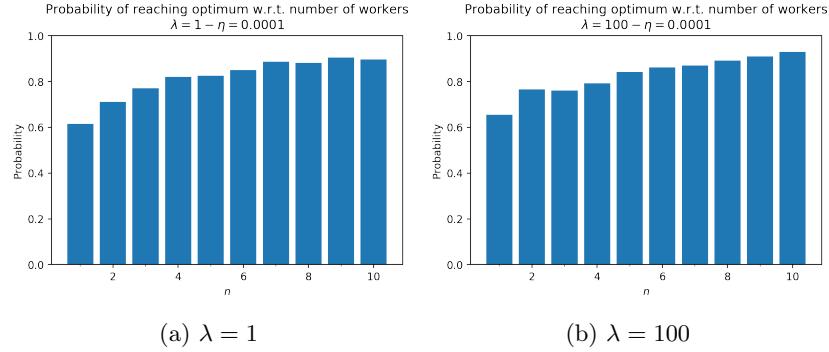


Figure 2.4: Probability distribution extracted from several Monte-Carlo simulations under different conditions. We find that the probability of reaching the optimum (Beale’s function) increases when the number of random initialized workers increases. Despite the fact that we observe that more exploration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

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 was not biased towards a specific minima, the statistic for a single worker under different hyperparameterizations should be 50%.

Furthermore, what is the role of the exploration parameter  $\lambda$ ? Does it contribute to the optimization process besides reducing the amount of communication with the parameter server by increasing the amount of local work? In principle this would help to optimization process since more exploration of the parameter space occurs, and as a result, better updates are applied. Furthermore, remember what we said in Section 2.1 on synchronous data parallelism, that effectively increasing the number of workers implicitly increases the size of the mini-batch. Yet, in this particular case there is a subtle difference, i.e., local exploration of the parameter space. As a result, it is not implicitly increasing the size of the mini-batch since some form of local exploration occurs. As a result, *the averaged central variable will produce a less-noisy consensus based on the (local) exploration of the error space*. However, the problem lies in the fact when different sets of workers enter different minima, possibly because of *too much exploration of the error space*, as depicted in Figure 2.3. Because if this happens, then the averaging step could potentially reset the situation instead of continuing exploring a single minima.



Figure 2.5: Workers which have been randomly initialized (experiments use same initialization) with different values for  $\lambda$  (communication frequency, or number of local worker iterations). Since the workers are initially randomized of the hypothesis space, the central variable in the first averaging step will be shifted towards the global minima because of the amount of local exploration, and due to the bias of the error space as shown in Figure 2.4.

### 2.2.2 Elastic Averaging SGD

Elastic Averaging SGD, or EASGD [17], is a distributed optimization algorithm designed with communication constraints in mind. In essence, EASGD could be viewed as an extension of model averaging described in Section 2.2.1 with the difference that the workers, and the central variable are coordinated using an *elastic force* [17] instead of averaging the workers after a fixed amount of steps. This means that instead of simply transmitting the parametrization or a gradient to the parameter server, the workers commit an *elastic difference* which is defined as  $\eta_t \rho(\theta_t^k - \tilde{\theta}_t)$  where  $\rho$  is the *elasticity* hyperparameter. Intuitively,  $\rho$  describes the amount of exploration that can be performed by the workers with respect to the central variable  $\tilde{\theta}_t$ .

Assuming  $\lambda = 1$ , the worker update and central variable update are described in Equation 2.4, and Equation 2.5 respectively. What is different compared to most other optimization schemes, is that the worker update described in Equation 2.4 has a second component, i.e., the *elastic difference*. Furthermore, note that compared to other distributed optimization schemes, the workers in EASGD do not synchronize their parameterization with the central variable as shown in Algorithm 4, but rather update the central variable using the elastic difference, and than use the new central variable as a new reference point to compute the following elastic differences.

$$\theta_{t+1}^k = \theta_t^k - \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) - \eta_t \rho(\theta_t^k - \tilde{\theta}_t) \quad (2.4)$$

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t + \eta_t \sum_{i=0}^n \rho(\theta_t^i - \tilde{\theta}_t) \quad (2.5)$$



Figure 2.6: In this example we would like to make the intuition behind Elastic Averaging SGD clear by describing it as a classical physics problem. In this particular figure we have 1 worker,  $\theta_w$  ( $n = 1$ ,  $\lambda = 1$ ), and a central variable  $\tilde{\theta}$ . The  $y$ -axis represents the error with respect to a certain parameterization  $\theta$ . Using Algorithm 4, the worker performs  $\lambda$  steps to compute the next value of  $\theta_w$  using Equation 2.4. As stated before in Section 2.2.2, Equation 2.4 holds 2 components, i.e., the regular negated first-order gradient ( $-\eta_t \nabla_{\theta} \mathcal{L}_w$ ), and the *negated* elastic difference ( $-\eta_t \rho(\theta_w - \tilde{\theta})$ ). As always, the negated first-order gradient represents the steepest slope with respect to the current error (loss), and parameterization. However, the *negated* elastic difference actually points towards the central variable. The magnitude of the elastic difference is controlled by hyperparameter  $\rho$ . This implies that a large value of  $\rho$  *strongly* limits the amount of exploration a worker can perform with respect to the central variable since the magnitude of the elastic difference vector will be proportionally larger compared to a small  $\rho$ . Finally, the central variable is updated using Equation 2.5 and the elastic difference coming from all workers. Furthermore, we would like to note that the elastic difference during the central variable update is *not* negated. Meaning that the central variable is optimized with respect to the “pull” of all workers, with the effect that workers which are ahead in the optimization process, will have a larger elastic force, causing the central variable to be influenced more strongly by distant workers. Nevertheless, at the same time, distant workers are pulled towards the central variable using an equal but opposite force *and* the negated gradient, i.e., the net force of the elastic difference and the negated gradient combined results either in a step towards a minimum, a null-operation, or a step in the direction of the central variable.

---

**Algorithm 4** Worker procedure of synchronous EASGD. This algorithms accepts several hyperparameters, the first being the number of local computations  $\lambda$ , the exploration hyperparameter  $\rho$ , and the dynamic learning rate  $\eta_t$ .

---

```

1: procedure EASGDPARTNER( $k$ )
2:    $\theta_0^k \leftarrow \tilde{\theta} \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $i \leftarrow 0$ 
6:     for  $i < \lambda$  do
7:        $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
8:        $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ 
9:        $i \leftarrow i + 1$ 
10:       $t \leftarrow t + 1$ 
11:    end for
12:     $\mathcal{E} = \eta_t \rho (\theta_t^k - \tilde{\theta})$ 
13:     $\theta_{t+1}^k = \theta_t^k - \mathcal{E}$ 
14:    COMMIT( $\mathcal{E}$ )
15:    WAITFOROTHERWORKERS()
16:     $\tilde{\theta} \leftarrow \text{PULL}()$ 
17:     $t \leftarrow t + 1$ 
18:  end while
19: end procedure

```

---

However, there is an interesting property about the elastic difference which is rather problematic for the communication frequency  $\lambda$ . Using the intuition from Figure 2.6, we can deduce that Equation 2.4 has an additional solution for 0 besides when  $\nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) = 0$ . This solution has profound implications on  $\lambda$ , and as a result, on the allocated computing resources. Because if this situation occurs, then all additional computations are wasted. Since during that time, the central variable is not updated, and the worker are not updated. In order to describe this formally, let us consider the cases for which the EASGD worker update rule is 0, as shown in Equation 2.6.

$$-\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) - \eta_t \rho (\theta_t^k - \tilde{\theta}_t) = 0 \quad (2.6)$$

Since we know that when a worker evaluates a 0-gradient, i.e.,  $\nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) = 0$ , then the EASGD worker update rule is also 0. Since no change in the parameterization of the worker took place, and as a result, there is no change in the value of the elastic difference. Unless  $\lambda > 1$ , and in step  $t - 1$ ,  $\nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) \neq 0$ . In this case, a net force will be applied in the direction of the central variable, that is, the worker will move back towards the central variable.

Nevertheless, let us focus our efforts on the case when Equation 2.6 is satisfied. First, let us assume that  $\nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) \neq 0$ . As a result, the only way that the condition described in Equation 2.6 can be satisfied is when the *non-negated* elastic difference equals the gradient update as shown in Equation 2.7.

$$-\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) = \eta_t \rho (\theta_t^k - \tilde{\theta}_t) \quad (2.7)$$

Using Equation 2.7, we can derive a condition, described in Equation 2.8, where the amount of computational work is not wasted, and introduce an early-stopping mechanism to prevent such waste.

$$\|-\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k)\| > \|\eta_t \rho (\theta_t^k - \tilde{\theta}_t)\| \quad (2.8)$$

One might ask why the elastic difference in Equation 2.8 needs to be smaller than the gradient term in order for the worker to move towards a minima? Remember from Figure 2.6 that the worker only moves (backwards or forwards) whenever Equation 2.6 is not satisfied. As a result, we can deduce that a worker only moves towards a minima when Equation 2.8 is met until the *equilibrium* condition

in Equation 2.7 is satisfied. At this point, a worker could stop any further local computations since additional computations would be wasted anyway. Of course, assuming the mini-batch size  $m$  is sufficiently large to eradicate any noise originating from the gradients. In order to proof this result empirically, we conducted several experiments with ordinary hyperparameters ( $\lambda = 30$ ,  $\eta = 0.00001$ ,  $n = 5$ ) for different values of  $\rho$  summarized in Figure 2.7 and Figure 2.8. What we observe in both cases is that the workers initially perform a large amount of exploration due to the relatively large value of  $\lambda$ . Since  $\lambda$  has a large value, the workers do not often synchronize with the central variable causing them to reach the equilibrium condition described in Equation 2.7 because the gradients are nog large enough to satisfy Equation 2.8.

To summarize, if the gradients produced by the workers are not large enough to produce a net force in the direction of a minima, then the workers are in an equilibrium with the elastic difference causing additional computational work to be wasted since the elastic difference will only lower if the distance between the worker and the central variable gets smaller. An obvious solution to this problem would be to lower the value of  $\lambda$  or  $\rho$ . However, lowering the value of  $\lambda$  causes the worker to communicate more frequently with the parameter server, even if the equilibrium condition is not met. Since EASGD is designed with communication constraints in mind, this is not an ideal solution. Furthermore, lowering  $\rho$  is also not ideal since this would reduce the equilibrium condition even further, causing workers to deviate further from the central variable. As mentioned before in Section 2.2.1 this is not desired because the possibility exists that different sets of workers will fall into different minima. This is especially problematic in EASGD since the parameterizations of the workers are not synchronized with the central variable, but are coordinated using the elastic difference. For example, imagine the case when two sets of workers (equal in number) are in two different minima with the central variable being in the middle of these two sets. Furthermore, assume that the elastic difference in these two sets of workers is identical and Equation 2.7 is satisfied. This implies that the central variable and the workers are not able to move, even after a synchronization step, which in turn results in the fact that the central variable is not able to converge.



Figure 2.7: In this experiment we use  $\rho = 5$ , as suggested by the authors [17]. We observe that at some point the elastic difference is starting to influence the first-order gradients causing the workers to bend downwards, which is not present in Figure 2.8. This “bend” is due to a significant elastic force pointing in the direction of the central variable. If we would decompose the vector in its main components, i.e.,  $\theta_1$  and  $\theta_2$ , we would find that the  $\theta_2$  component is significant enough to cause the worker to bend downwards since the elastic force is getting stronger proportional to the distance between the worker and the central variable and hyperparameter  $\rho$ . Furthermore, the equilibrium plot shows that the worker does a lot of initial exploration, while the central variable slowly follows the workers based on the averaged elastic difference. After 40000 steps, we see that the workers reach the equilibrium condition. As a result, any computational work done by the workers within a communication window is wasted since the distance between the workers and the central variable needs to grow smaller for the elastic force to shrink. However, because the elastic force is shrinking proportionally to the distance between the workers and the central variable, it allows the workers to explore *slightly* more of the hypothesis space since the workers already met the equilibrium condition.

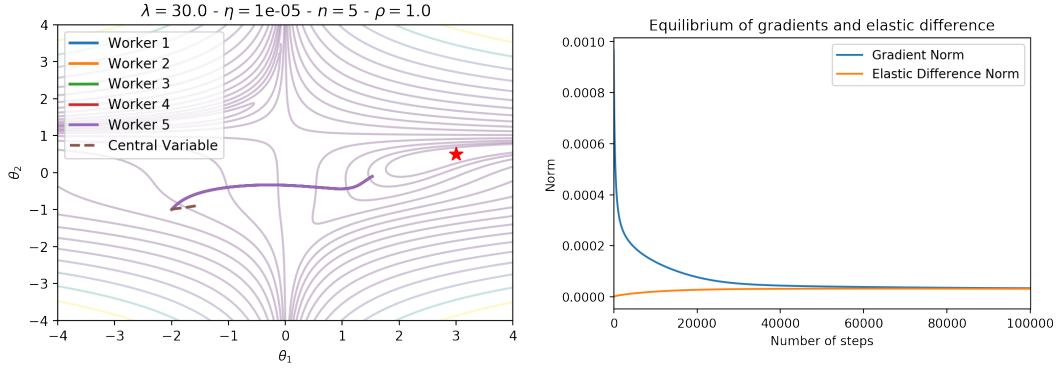


Figure 2.8: In this experiment we reduce the value of the exploration hyperparameter  $\rho$  while all other hyperparameters from our experiment in Figure 2.7 remain the same. After running the experiment we do not observe the ‘‘bend’’ which we initially observed in Figure 2.7. However, this was expected since the elastic force is not that prominent due to the reduced value of hyperparameter  $\rho$ . Nevertheless, we still observe that the workers reach the equilibrium condition, with the difference that the central variable shows slower convergence behavior compared to Figure 2.7.

To improve convergence of the central variable in EASGD, and reduce the amount of wasted computational resources, we implemented Equation 2.9 based on Equation 2.8 as a measure to stop wasting further computations.

$$\|-\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k)\| \leq \|\eta_t \rho(\theta_t^k - \tilde{\theta}_t)\| \quad (2.9)$$

However, we observed from our experiments that this exact condition is rarely satisfied. Of course, we made the assumption in Equation 2.7 that the parameterization of a worker does *not* change if a worker settles in an equilibrium. Of course, before reaching the equilibrium, the gradient is non-zero, this implies that there is a change in the parameterization of a worker causing the elastic difference to be modified. With the result that the net force applied to the worker is *approaching* the equilibrium. As a result, the net force applied to a worker is gradually becoming smaller.

In order to anticipate for a declining net force, and as a result, the gradually approaching equilibrium, we communicate the elastic difference with the parameter server if Equation 2.10 is met.

$$\left( \|\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k)\| - \|\eta_t \rho(\theta_t^k - \tilde{\theta}_t)\| \right) < \eta_t \quad (2.10)$$

Using the same experimental configuration as in Figure 2.7, with the difference that Equation 2.10 has been implemented as an early-stopping mechanism to communicate the elastic difference with the parameter server. Using Equation 2.10, we can see that the proposed early stopping mechanism benefits the central variable in terms of convergence with respect to the default experimental configuration in Figure 2.7.

What is remarkable about the equilibrium plot in Figure 2.9 is the presence of a sudden drop in the norm of the elastic difference vector which is absent in Figure 2.7 despite the fact that the workers follow similar paths. This is due to the early stopping mechanism which pulling the central variable closer towards the workers causing the decline in the norm of the elastic difference. Furthermore, since at this point the gradients are relatively small, a lot of local work can be done while at the same time the central variable will be able to keep the elastic distance small.

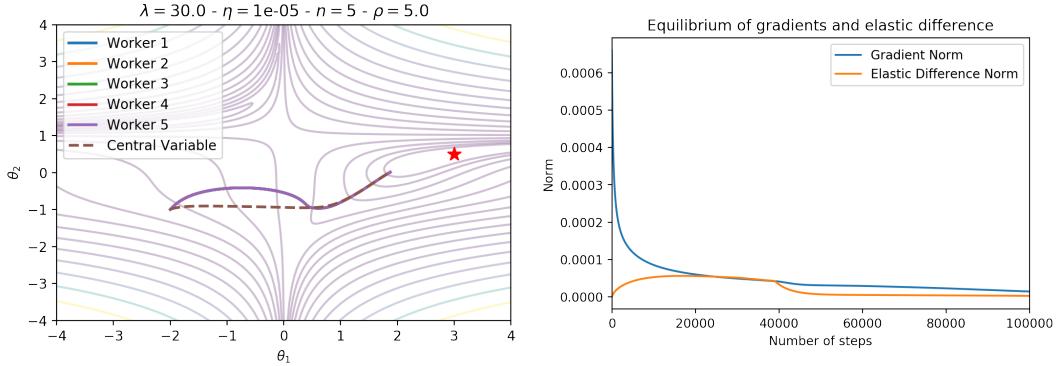


Figure 2.9: EASGD using the early stopping mechanism described in Equation 2.10. Compared to Figure 2.7, which uses the same experimental configuration, we observe that the central variable benefits in terms of convergence from increased communication when the early stopping condition is met.

Since the early stopping condition effectively communicates the elastic difference with the parameter server, is it not in essence increasing the communication frequency, i.e., lowering  $\lambda$ ? In effect it isn't, since communication only takes place when the workers and the central variable approach an equilibrium condition. To show this, we conducted several experiments with and without the early stopping mechanism shown in Figure 2.10 and Figure 2.11.

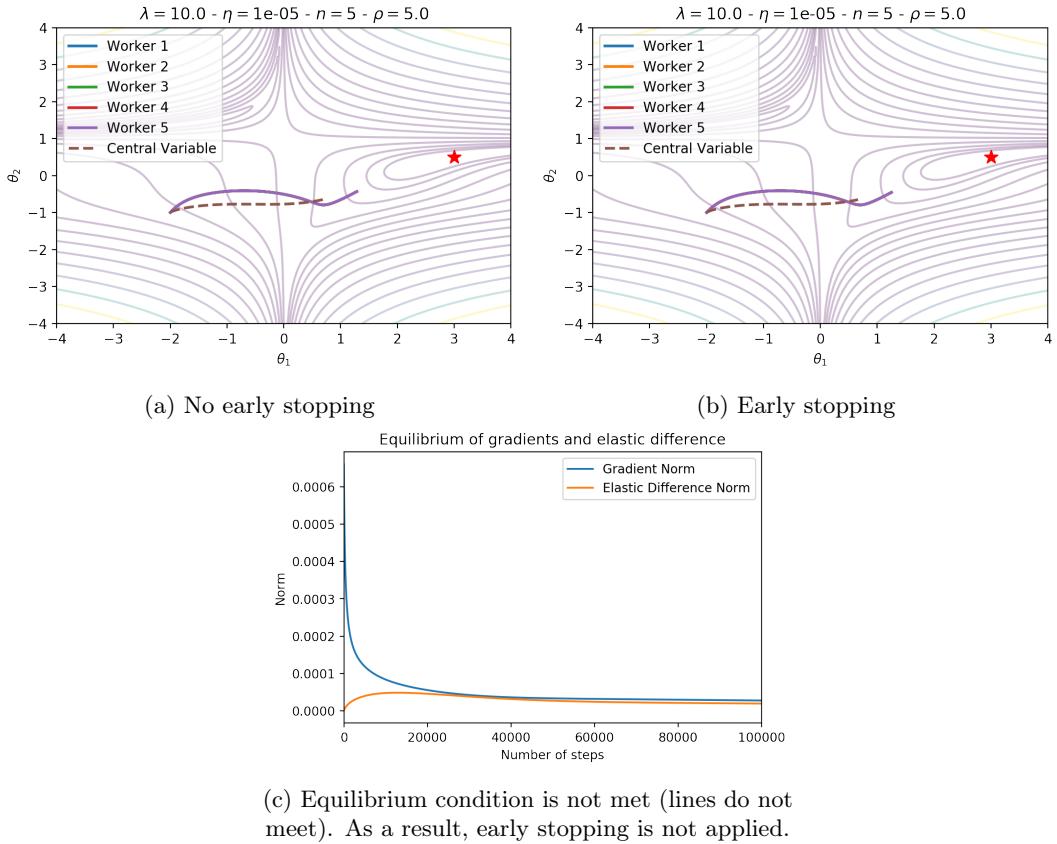
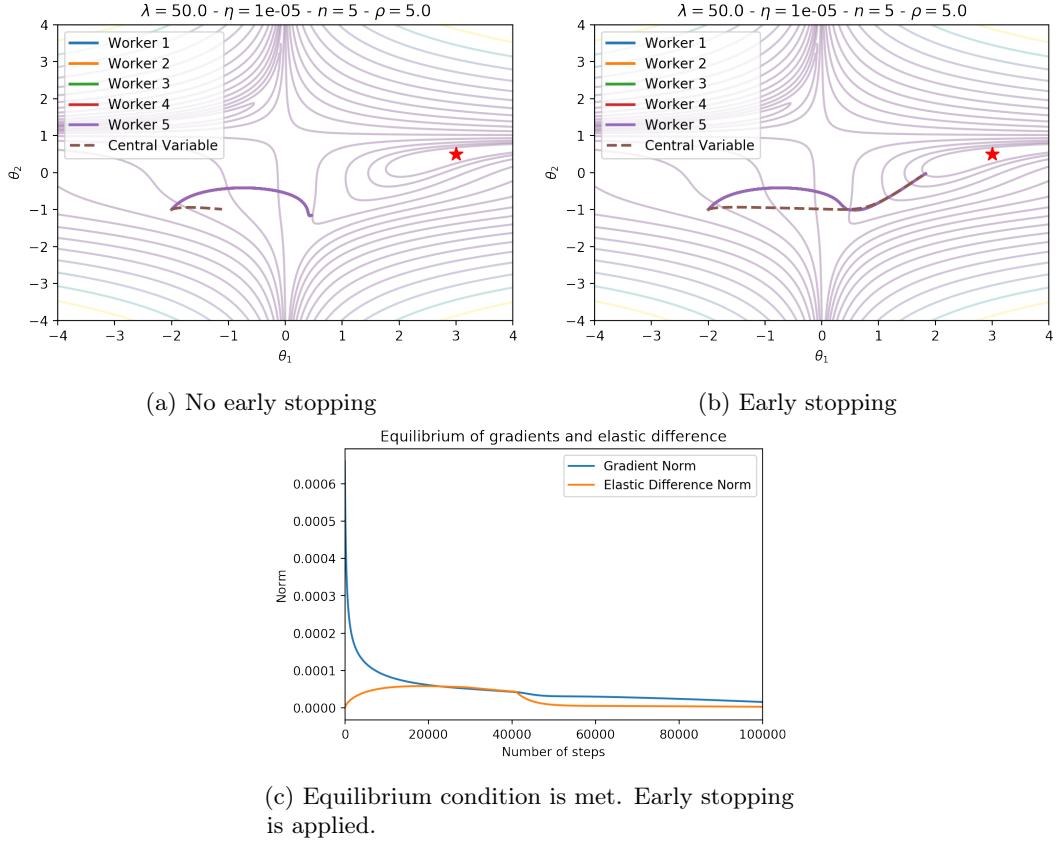


Figure 2.10:  $\lambda = 10$

Figure 2.11:  $\lambda = 50$ 

An other interesting phenomena is when EASGD is approaching a minima. Compared to other synchronous optimizers such as mini-batch parallelism, or model averaging, EASGD requires a significant amount of additional steps to actually converge to the minima because of the equilibrium condition as shown in Figure 2.12. Remember, if the workers are approaching the equilibrium condition, a small net force is applied to the worker because the *negated* elastic difference counter-acts the advances of the workers to minimize the variance of the workers with respect to the central variable.

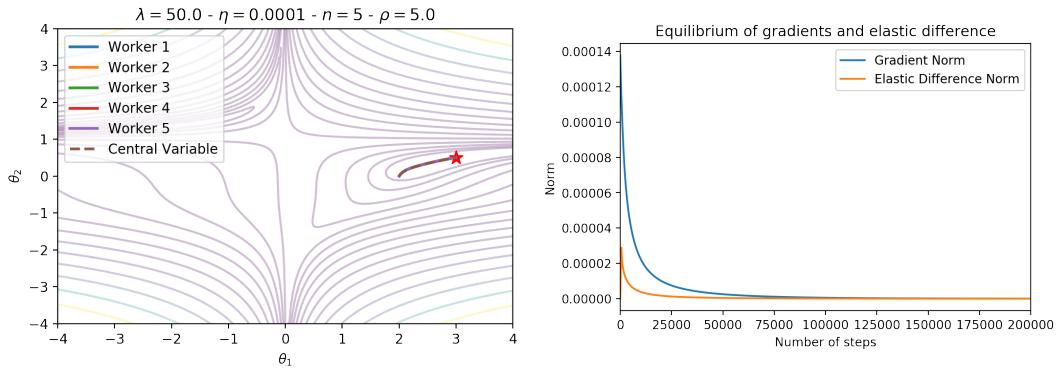


Figure 2.12: Slow converge of EASGD at minima due to the equilibrium condition. In this figure we apply early stopping but due to the small gradients close to the minima it does not have any significant effects.

## 2.3 Asynchronous Data Parallelism

### 2.3.1 DOWNPOUR

Waits induced by blocking mechanisms in synchronous data parallelism significantly reduce the hardware utilisation during training, especially in non-homogeneous hardware configurations. As mentioned in Section 2.1, this can be resolved by simply removing the synchronizations barriers, as suggested by [4]. The algorithm the authors proposed is called DOWNPOUR, and is shown in Algorithm 5.

DOWNPOUR is an intuitively a very simple algorithm. As before, we have  $n$  different workers optimizing a central variable  $\tilde{\theta}_t$  with the difference that every worker will *commit* a gradient after every mini-batch in an asynchronous fashion, and after a commit has been performed, the worker will synchronize with the parameter server by *pulling* the most recent parameterization of the central variable.

$$\theta_{t+1}^k = \theta_t^k - \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}^k) \quad (2.11)$$

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t - \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}^k) \quad (2.12)$$

As mentioned in Chapter 1, due to the asynchronous optimization one does not have the issue with idle workers due to synchronizations mechanisms. Instead, there is a more severe problem related to the parameterizations of the workers. Since workers will commit gradients to the central variable in an asynchronous manner, other workers will commit gradients based on older parameterizations of the central variable. As a result, the gradients of these workers are *stale* as shown in Figure 1.4 and Figure 2.13. An additional, but related issue to parameter staleness, is that increasing the amount of asynchronous workers, increases the staleness in the system as shown in Figure 2.1.

---

**Algorithm 5** Worker procedure of DOWNPOUR, which is a parallelized extension of SGD.

---

```

1: procedure DOWNPOURWORKER( $k$ )
2:    $\theta_0^k \leftarrow \tilde{\theta} \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
6:      $g \leftarrow -\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ 
7:     COMMIT( $g$ )
8:      $\theta_{t+1}^k \leftarrow \text{PULL}()$ 
9:      $t \leftarrow t + 1$ 
10:   end while
11: end procedure
```

---

Nevertheless, contrary to EASGD, DOWNPOUR is not designed with communication constraints in mind. As a result, increasing the number of workers in an already saturated environment will not reduce the training time since whenever a worker computes a gradient, it will commit the result to the parameter server. To illustrate this, imagine having several workers committing highly dimensional gradients to the parameter server. Since it takes some time to incorporate these gradients into the central variable, other workers will have to wait for their turn in the queue. To reduce the side-effects (waits) in this particular situation, [11] proposes to before committing the gradient, a worker should send a small control message to check if the queue is currently occupied, i.e., gradients of other workers are being incorporated into the central variable. If this is the case, do not commit the gradient, but fetch the next mini-batch, compute the gradient, and accumulate the computed gradient with the previously computed gradient, and send a new control message

An obvious problem with this method is that the possibility exists that after  $x$  number of retries, the central variable queue is still occupied by other workers. Following the naive procedure, the zealous method should keep accumulating new gradients. However, this is problematic at step

$x + 1$ , when the queue is available. Imagine the possibility that other workers, and the central variable converged to a different minima. If our worker would commit the accumulated gradient in this particular situation, then the central variable would not be close to its minima as it was before. In order to reduce the amount of staleness in such a situation, one could introduce a limit to the amount of local steps that can be made in an attempt to reduce staleness as shown in Algorithm 6.

However, this technique introduces an additional hyperparameter. Several other approaches which handle this problem in a different way without introducing new hyperparameters, and additionally giving some intuition and insight into parameter staleness in Deep Learning are discussed in detail in Chapter 3 and Chapter 4.

---

**Algorithm 6** Zealous worker procedure of DOWNPOUR, inspired by [11].

---

```

1: procedure ZEALOUSDOWNPOURWORKER( $k$ )
2:    $\theta_0^k \leftarrow \tilde{\theta} \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $i \leftarrow 0$ 
6:      $g \leftarrow 0$ 
7:     while  $i < \lambda$  do
8:        $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
9:        $g \leftarrow g - \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ 
10:      if not QUEUEOCCUPIED( ) or  $i = (\lambda - 1)$  then
11:        COMMIT( $g$ )
12:         $\theta_{t+1}^k \leftarrow \text{PULL}()$ 
13:         $i \leftarrow \lambda$                                  $\triangleright$  Satisfy stopping condition.
14:      end if
15:       $t \leftarrow t + 1$ 
16:       $i \leftarrow i + 1$ 
17:    end while
18:  end while
19: end procedure

```

---

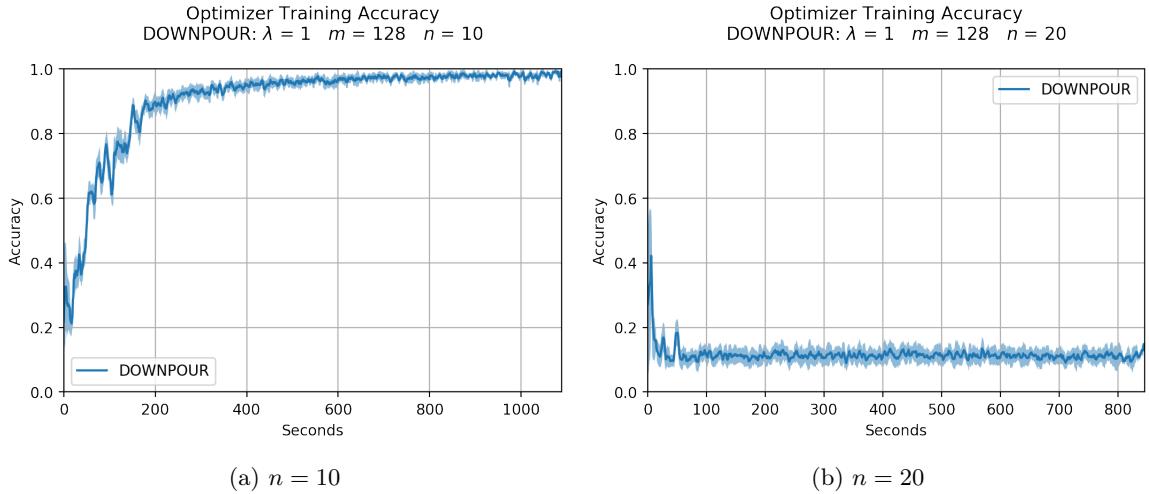


Figure 2.13: Divergence due to number ( $n = 20$ ) of asynchronous workers in the optimization process [13] and not dealing with parameter staleness in a more intelligent way. Lowering the number workers ( $n = 10$ ) causes the central variable to converge.

### 2.3.2 Dynamic SGD

The intrinsic reason that asynchronous methods like DOWNPOUR cannot guarantee convergence is that these method directly incorporate local updates (good) and stale updates (bad) in the central variable [8]. To combat this, one could incorporate staleness information of a worker to have a per-worker learning which decays proportionally stale gradients proportional to the staleness [8]. As a result, Equation 2.13 will better cope with parameter staleness in contrast to DOWNPOUR.

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t - \frac{\eta_t}{\tau_i} \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) \quad (2.13)$$

However, we would like to note that in a homogeneous hardware configuration the gradients will be scaled down proportional to the number of asynchronous workers as  $\mathbf{E}[\tau] = (n - 1)$  [13]. As a result, the expected scalar that will be applied to a worker gradient in a homogeneous setting is summarized in Equation 2.14.

$$\mathbf{E}\left[\frac{1}{\tau}\right] = \left(\frac{1}{n - 1}\right) \quad (2.14)$$

Furthermore, as we will see in Chapter 4, this technique scales down gradients with respect to the number of stale steps, while the issue of staleness mainly arises from the *distance between parameterizations*. For example, imagine a plateau where the workers have to pass through in order to converge to a minima. Since progress is slow in a plateau, thus the distance between the workers and the central variable will remain relatively small, it is non-sensical to scale down the gradients with respect to the number of stale steps, since these gradients provide good information because they are *close to the “old” central variable*.

Nevertheless, in order to capture the staleness information, some additional engineering is required to keep track of worker staleness since the worker themselves do not possess this information. Contrary to [8], we propose a different architecture to capture the staleness information. In our approach we exchange messages between the parameters server and workers. These messages can obtain additional information beside the parameterization of a worker or the central variable. During the initialization phase of the training procedure, we keep track of a *parameter server clock*. Basically, this clock holds the number of updates that have been applied to the central variable. Furthermore, in order to obtain the staleness of every worker, and apply the scaling factor described in Equation 2.13, the parameter server needs to keep track of the last *clock tick* when a worker  $i$  sent an update. It does so by maintaining a hashmap, or a different datastructure, to associate the last update clock tick with a particular worker. Furthermore, we scale the parameterizations on the parameter server since this will ensure the consistency of the staleness information because it is possible that an other worker might commit an update in the meantime (CAP theorem), and reduce the amount of blocking mechanisms. Nevertheless, for (very) large networks there might be an issue when insufficient processing power is allocated to preprocess the queue in an efficient manner.

An additional requirement of this architecture, is that the parameter server needs to update the worker datastructure whenever a worker *pulls* the central variable (because the worker will compute a gradient based on the parameterization of the central variable it just pulled, and this is required to compute the number of updates that happened in between). Then, whenever a worker commits a gradient to the central variable, the parameter server just needs to compute the difference between the last recorded clock time of this particular worker and the current value of the parameter server clock. Where the last recorded clock tick of a particular worker can be retrieved from the data structure mentioned above. The complete pseudo-code for these procedures can be found in Algorithm 7 and Algorithm 8. However, an alternative approach which does not require an additional data structure to be maintained by the parameter server, is to simply send the current clock tick together with the parameterization of the central variable to the worker during a *pull*. Similar to our previous technique, the scaling will still happen at the side of the parameter server in order to preserve staleness consistency. However, when the worker commits a gradient to the parameter server, the parameter server does not have to do additional searches in the worker dictionary since the value of the parameter server clock

at the time of the central variable pull is added to the commit message. As a result, the difference between the current clock value and the given clock value, thus the number of stale updates, can be derived easily. However, this requires the worker procedure to be slightly modified whereas the worker procedure described in Algorithm 7 does not contain any modifications to extract staleness information.

---

**Algorithm 7** Worker procedure of DYNSGD.

---

```

1: procedure DYNSGDWORKER( $k$ )
2:    $\theta_0^k \leftarrow \tilde{\theta} \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $i \leftarrow 0$ 
6:      $a \leftarrow 0$ 
7:     while  $i < \lambda$  do
8:        $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
9:        $g \leftarrow \eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ 
10:       $a \leftarrow a - g$ 
11:       $\theta_{t+1}^k \leftarrow \theta_t^k - g$ 
12:       $t \leftarrow t + 1$ 
13:       $i \leftarrow i + 1$ 
14:    end while
15:    COMMIT( $a$ )
16:     $\theta_{t+1}^k \leftarrow \text{PULL}()$ 
17:  end while
18: end procedure

```

---

**Algorithm 8** Parameter server procedures of DYNSGD.

---

```

1: procedure DYNSGDPARAMETERSERVER
2:    $\tilde{c} \leftarrow 0$                                  $\triangleright$  Parameter server clock
3:    $\tilde{m} \leftarrow \tilde{c}$                           $\triangleright$  Initialize staleness datastructure
4:
5:   procedure HANDLEPULL( $k$ )                       $\triangleright k$  denotes the worker identifier
6:      $\tilde{m}[k] = \tilde{c}$ 
7:     return  $\tilde{\theta}_{\tilde{c}}$ 
8:   end procedure
9:
10:  procedure HANDLECOMMIT( $k, \Delta\theta^k$ )
11:     $\tau \leftarrow \tilde{c} - \tilde{m}[k]$ 
12:     $\tilde{\theta}_{\tilde{c}+1} = \tilde{\theta}_{\tilde{c}} + \frac{1}{\tau+1} \odot \Delta\theta^k$            $\triangleright +1$  to prevent division by 0
13:     $\tilde{c} \leftarrow \tilde{c} + 1$ 
14:  end procedure
15:
16: end procedure

```

---

### Experimental Validation of Dynamic SGD

To validate the claims made by [8], we conducted several experiments on the MNIST [10] dataset with different hyperparameterizations. Using an equivalent experimental configuration as with DOWNPOUR in Section 2.3.1, and identical hyperparameterizations, we confirm that DYNSGD is able to cope with an increased amount of staleness due to asynchrony compared to DOWNPOUR as shown in Figure 2.14. Furthermore, due to the high communication frequency in Figure 2.14, a significant amount of communication took place which reduced the throughput of the workers. To combat this, we decreased the communication frequency (thus increasing  $\lambda$ ) which allows for more local work to be

done before incorporating the *accumulated gradient* into the central variable, as shown in Figure 2.15. As expected, the training time was reduced significantly. However, due to the increased amount of local work, the gradients that were submitted to the parameter server became proportionally larger as well. Contrary to the expectations of the authors in [8], DYNSGD is not able to handle this amount of staleness. As a result, this is an indication that staleness is a more “deeper” problem, especially in the context of Deep Learning, which will be addressed in Chapter 4.

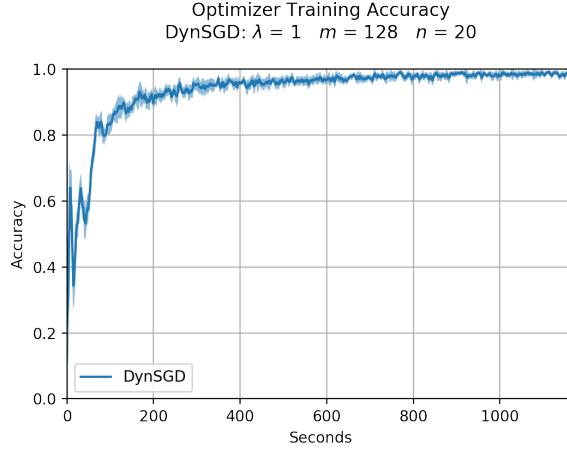


Figure 2.14: In this experiment we applied 20 asynchronous workers on equally shared 5 epochs worth of MNIST training data. Since this is an identical experimental configuration for which DOWNPOUR was used, this experiment verifies the claims of the authors that DYNSGD is able to handle staleness better.

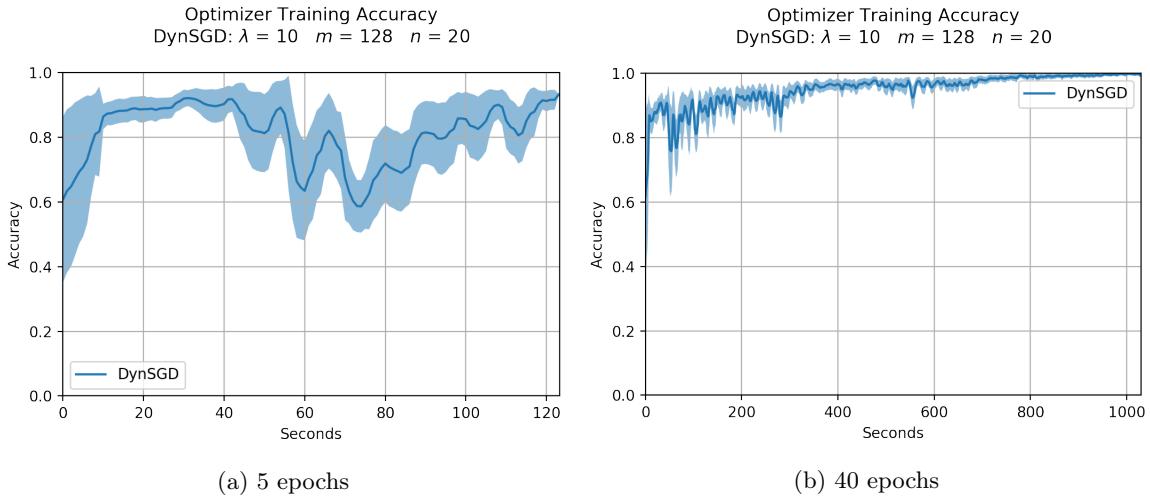


Figure 2.15: Since network-communication is a blocking factor in terms of CPU usage, we decided to test several iterations of local exploration before committing the accumulated gradient to the parameter server. In our first experiment (a), we copied the experimental configuration from Figure 2.14, with the difference that we decreased the communication frequency. Due to the decreased communication frequency, more local work was done, which had the result that the norm of the committed gradients to the parameter were larger. This had profound implications on convergence stability of the central variable, which only reached 90% training accuracy while using the same amount of data, but, spending less time. In order to have a “fair” comparison with Figure 2.14 in terms of training time, we increased the total amount of training data to 40 epochs, which resulted in a training accuracy of 99.7%, compared to 98.1% in Figure 2.14.

### 2.3.3 Asynchronous Elastic Averaging SGD

In Section 2.2.2 we discussed the synchronous version of EASGD. To reduce the effect of blocking mechanisms, which is profound in synchronous data parallelism, the authors proposed [17] an asynchronous extension of EASGD which is called AEASGD, or Asynchronous Elastic Averaging SGD. In essence, nothing changed for the worker procedure shown in Equation 2.15 with the exception of the blocking mechanism. However, contrary to other optimization algorithms discussed in this chapter (which commit first-order gradients), AEASGD is optimizing the central variable by incorporating the elastic differences of the workers in an asynchronous fashion, as shown in Equation 2.16 (for intuition, see Figure 2.6).

$$\theta_{t+1}^k = \theta_t^k - \eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{x}_t^k; \mathbf{y}_t^k) - \eta_t \rho(\theta_t^k - \tilde{\theta}_t) \quad (2.15)$$

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t + \eta_t \rho(\theta_t^k - \tilde{\theta}_t) \quad (2.16)$$

Since EASGD and its variants are designed with communication constraints in mind, and allow for more exploration of the local hypothesis space [17], we can significantly reduce the communication frequency, and thus further reducing the training time. However, as in any asynchronous optimizer, AEASGD mitigates staleness by pulling the most recent central variable in order to compute a more up-to-date elastic difference as shown in Algorithm 4.

---

**Algorithm 9** Worker procedure of AEASGD. Note how the pull of the central variable occurs *before* the computation of the elastic difference.

---

```

1: procedure AEASGDWORKER( $k$ )
2:    $\theta_0^k \leftarrow \tilde{\theta} \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $i \leftarrow 0$ 
6:     for  $i < \lambda$  do
7:        $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
8:        $\theta_{t+1}^k \leftarrow \theta_t^k - \eta_t \odot \nabla_\theta \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ 
9:        $i \leftarrow i + 1$ 
10:       $t \leftarrow t + 1$ 
11:    end for
12:     $\tilde{\theta} \leftarrow \text{PULL}()$ 
13:     $\mathcal{E} = \eta_t \rho(\theta_t^k - \tilde{\theta})$ 
14:     $\theta_{t+1}^k = \theta_t^k - \mathcal{E}$ 
15:    COMMIT( $\mathcal{E}$ )
16:     $t \leftarrow t + 1$ 
17:  end while
18: end procedure
```

---

What is interesting about AEASGD in our experiments, is that given identical hyperparameterizations as other optimizers in this chapter AEASGD is not able to get closer to a minima as can be seen in Figure 2.16. This could potentially be due to the equilibrium condition described in Section 2.2.2. Nevertheless, the claim the authors make that EASGD benefits from more exploration is validated in Figure 2.16 as well. Furthermore, during our experiments we have encountered several interesting observations on AEASGD. The first being when a high communication frequency is used, i.e.,  $\lambda = 1$ , AEASGD shows sign of *divergence*. However, this effect is not always present, and our suspicions are that numerical error might be the root cause of this issue. In order to better understand the properties of the divergence in AEASGD, we conducted several simulations which show similar behaviour. To this date, we do not really have an idea why this is exactly happening, and is subject to further investigation. Furthermore, we would like to declare the following observations, as a result of these simulations:

- Asynchronous optimizers which commit gradients: central variable is “pushed” by workers towards a minima.
- EASGD: central variable is “pulled” by the workers towards a minima as described in Section 2.2.2.
- AEASGD: central variable acts as a “mean” of a distribution of workers, where the variance of the workers is controlled by  $\rho$ .

The final remarkable observation, shown in Figure 2.17, was that in almost all tests, the validation accuracy of the optimizer was consistently better than the training accuracy. According to us, this is because of the observation mentioned above, i.e., because AEASGD behaves like a moving distribution (more like a particle filter), it will tend to find a flat minima which generalizes better than the training accuracy of the central variable. However, this is pure speculation and is subject to further research.

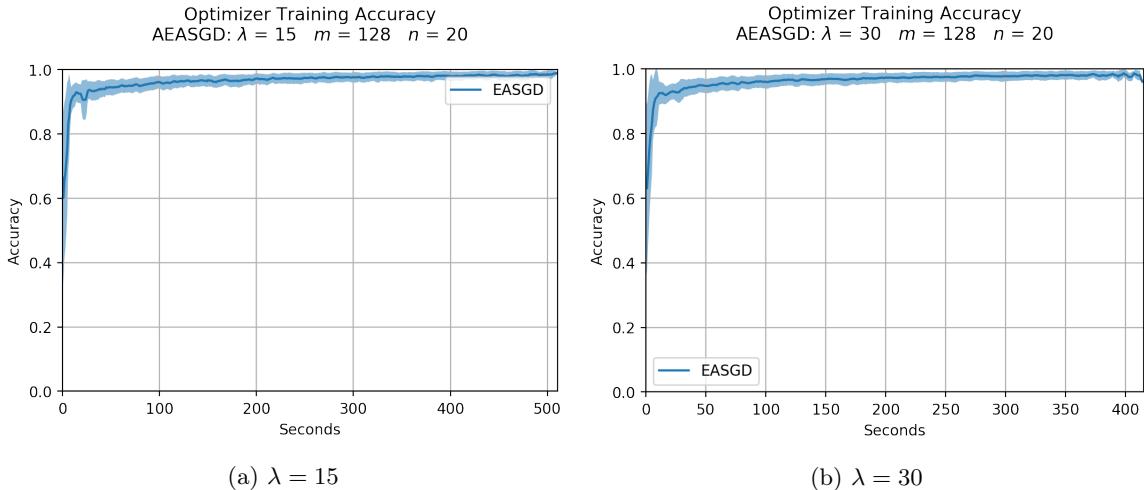


Figure 2.16: Accuracy plots which show the training accuracy of the central variable for different communication frequencies. These results validate the claims of the authors that in general, the central variable benefits from more exploration. However, we would argue that this is true only when sufficient amount of data is available. Furthermore, we would like to note that compared to other optimizers discussed in this Chapter, EASGD seems to converge slowly close to a minima, this might be due to the equilibrium condition described in Section 2.2.2. However, further evidence is required to confirm this.

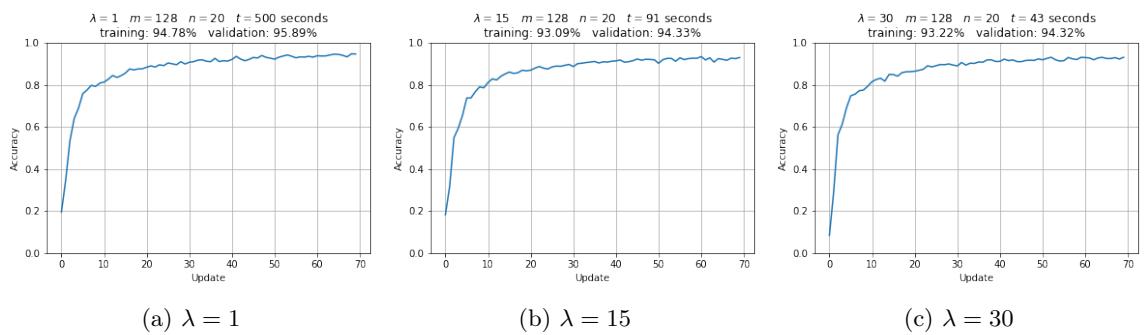


Figure 2.17: Results of a small experiment to show the consistently higher validation accuracy compared to the training accuracy in AEASGD for different values of  $\lambda$ .

## 2.4 Hybrids

### 2.4.1 Stale Synchronous Optimizers

Other approaches to deal with staleness in asynchronous optimization have been suggested in the past years [3, 6]. These approaches, typically called Stale Synchronous Optimizers, are based on the concept that staleness needs to be limited within a system. In order to limit the amount of staleness within a system, asynchronous workers can only be a certain number of steps ahead compared to the slowest worker. Hence the name stale synchronous optimizers, since they allow for a limited amount of staleness, and if this particular condition is not met by some workers, then these workers will have to wait until the slower workers caught up. However, as mentioned before, we know from [13] that the expected staleness in a homogeneous system is  $E(\tau) = (n - 1)$ . As a result, adding more workers to the problem in stale synchronous optimizers is non-sensical, since adding more workers to the problem implicitly increases the amount of staleness. As a result, the limit which is specified will have to be changed whenever a couple of workers are added to the optimization process. For this reason, we will not consider stale synchronous optimizers in this thesis.

# Chapter 3

## Accumulated Gradient Normalization

This chapter addresses the first contribution of this thesis, which is *accumulated gradient normalization*, or AGN in short. We start by laying out the concept and intuition behind accumulated gradient normalization, and show why AGN provides the central variable with more efficient updates. Finally, to show our claims, we provide several experiments where we compare AGN with different distributed optimizers such as DOWNPOUR [4], AEASGD [17], and DYNNSGD [8].

### 3.1 Concept and intuition

The main issue with DOWNPOUR [4] is the requirement of constant communication with the parameter server after every gradient computation. Furthermore, as the number of parallel workers increases, DOWNPOUR fails to converge due to the amount of *implicit momentum* [13], as shown in Figure 2.13. To reduce the amount of communication with the parameter server, one could take ideas from EASGD [17], and perform several iterations of local exploration before committing the gradients to the parameter server. However, in the case of algorithms like DOWNPOUR, that is, where gradients are committed to the parameter server in an asynchronous fashion, more local exploration results in proportionally larger gradients, and as a result, complicate the staleness and the implicit momentum problem even further as addressed in Chapter 4. To intuitively show why this is an issue, let us consider Figure 3.1. In a regular DOWNPOUR setting, first-order gradients such as in Subfigure (a) are committed to the parameter server. However, when an algorithm allows for a certain amount of local exploration, such as Algorithm 6, the gradient that is committed to the parameter server is an *accumulated gradient* as shown in Subfigure (b).

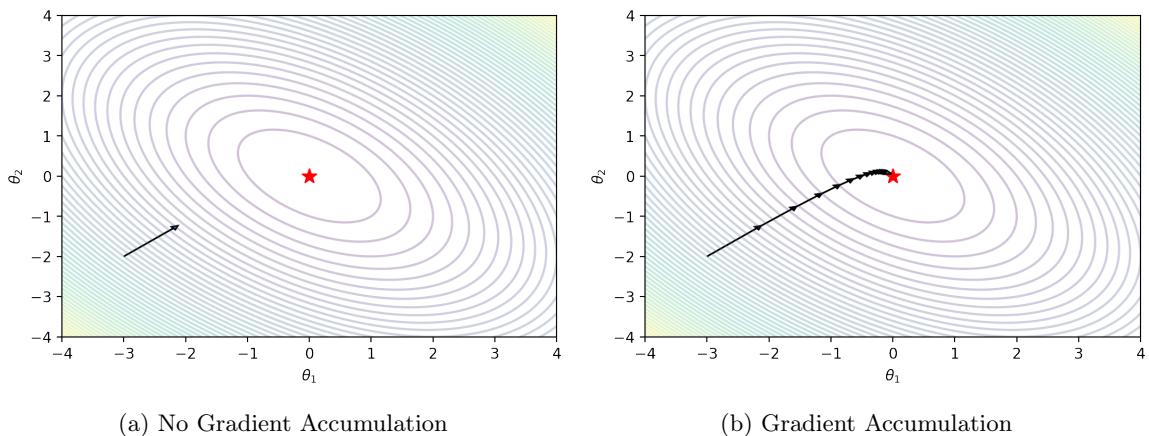


Figure 3.1: This figure shows the difference between regular first-order gradients (a), and accumulated gradients (b). We observe that *accumulated gradients are proportionally larger to the number of exploration steps*. However, they do provide a better direction compared to first-order gradients.

Now, imagine two asynchronous environments, in the first no gradient accumulation is performed, and in the last gradient accumulation takes place. In the environment where no gradient accumulation is performed, as in regular DOWNPOUR, first-order gradients are committed to the parameter server. However, as we have seen in Chapter 2, and in particular Figure 2.13, we saw that DOWNPOUR diverges when the number of asynchronous workers is too high due to the amount of implicit momentum [13]. As a result, careful tuning is required when no adaptive methods are applied. Nevertheless, given the fact that DOWNPOUR converges with  $n = 10$  workers in Figure 2.13 and our knowledge about gradient accumulation, i.e., *accumulated gradients that are committed are proportional to the number of exploration steps for every worker, and provide better directions to a minimum*, we would expect that for some amount of local exploration while using the same hyperparameterization (with the exception of local exploration steps  $\lambda$ ) DOWNPOUR would diverge again due to the magnitude of the accumulated gradients. This behaviour is illustrated in Figure 3.2.

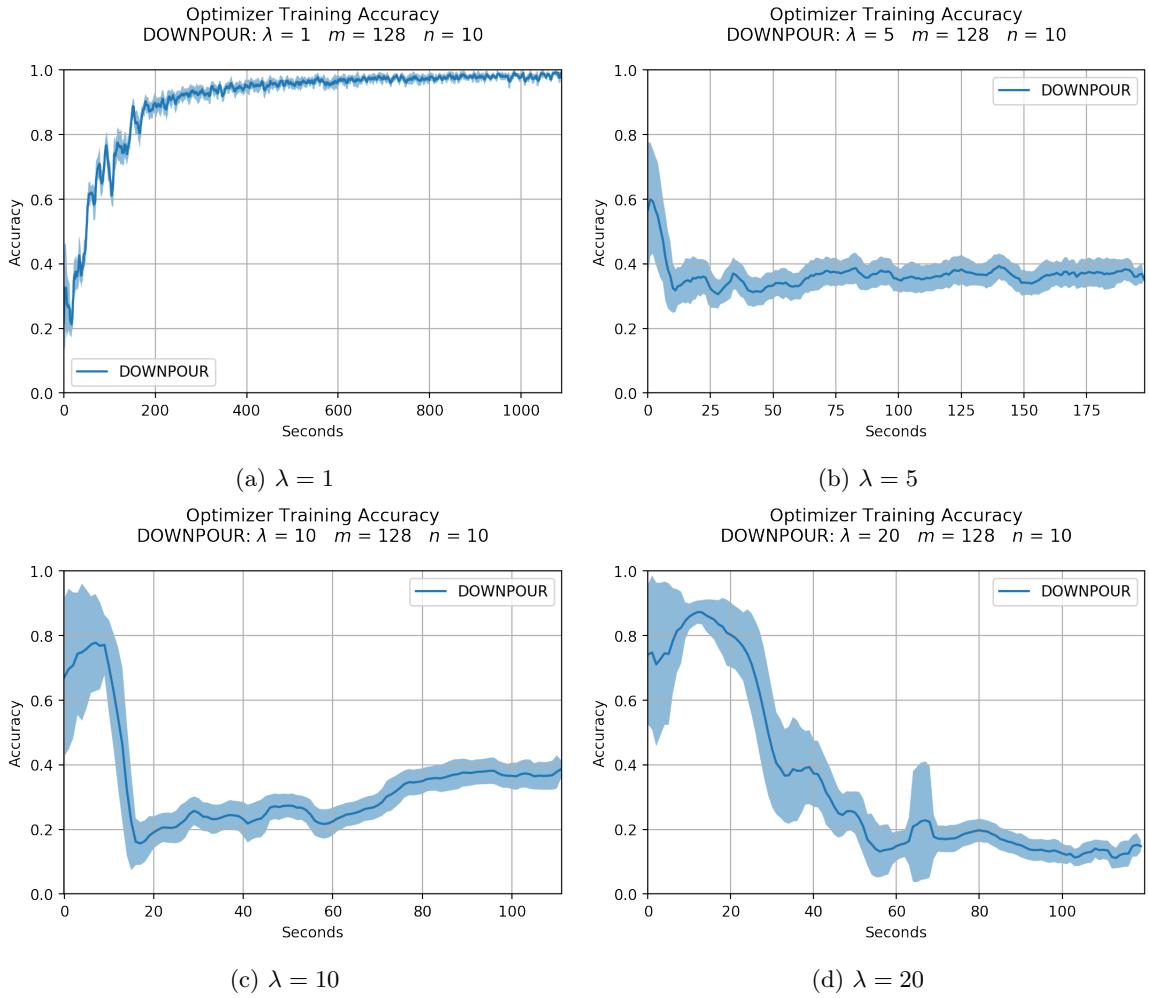


Figure 3.2: Illustration of divergence due to gradient accumulation in DOWNPOUR. In Figure 2.13, we say that for  $n = 10$  DOWNPOUR converged to a good solution. In order to reduce the training time, we decrease the communication frequency (increasing  $\lambda$ ). However, due to the larger gradients that are committed to the parameter server, which increases the amount of implicit momentum, the central variable is not able to converge as before.

To reduce the magnitude of the accumulated gradients, and thereby reducing the amount of implicit momentum, while at the same time preserving the better direction that has been provided due to the amount of local exploration, we propose to normalize (average) the accumulated gradient with

the amount of local steps that have been performed by the workers ( $\lambda$ ), shown in Equation 3.1<sup>1</sup>. We call this technique of normalizing the accumulated gradient *Accumulated Gradient Normalization* or AGN. An initial critique of this technique would be that by normalizing the accumulated gradient, AGN would in effect be undoing the work that has been done by a single worker. This seems at first a valid criticism, however, one needs to take into account that AGN is actually using the worker exploration steps to compute a better gradient based on first-order gradients.

$$\Delta\theta = -\frac{1}{\lambda} \sum_{i=0}^{\lambda} \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_{\theta} \mathcal{L}(\theta_i; x_{ij}; y_{ij}) \quad (3.1)$$

Since AGN is using local steps to compute a better gradient compared to first order gradients, it can also be used under communication constraints like EASGD since less communication with the parameter server is required. In Figure 3.3, we show how a *Normalized Accumulated Gradient* is obtained and applied to the central variable using Equation 3.1 as described in Algorithm 10.

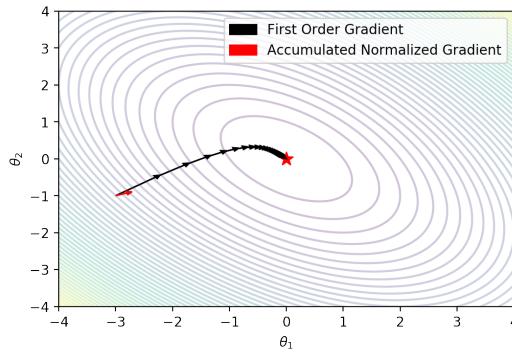


Figure 3.3: After pulling the most recent parameterization of the central variable from the parameter server, the worker starts accumulating  $\lambda$  first order gradients, and applies those gradients locally to explore the surrounding error space. Finally, after  $\lambda$  exploration steps have been performed, the accumulated is normalized w.r.t.  $\lambda$  and send to the parameter server.

---

**Algorithm 10** Worker procedure of AGN.

---

```

1: procedure AGNWORKER( $k$ )
2:    $\theta_0^k \leftarrow \tilde{\theta} \leftarrow \text{PULL}()$ 
3:    $t \leftarrow 0$ 
4:   while not converged do
5:      $i \leftarrow 0$ 
6:      $a \leftarrow 0$ 
7:     while  $i < \lambda$  do
8:        $\mathbf{x}, \mathbf{y} \leftarrow \text{FETCHNEXTMINIBATCH}()$ 
9:        $g \leftarrow -\eta_t \odot \nabla_{\theta} \mathcal{L}(\theta_t^k; \mathbf{x}; \mathbf{y})$ 
10:       $a \leftarrow a + g$ 
11:       $\theta_{t+1}^k = \theta_t^k + g$ 
12:       $i \leftarrow i + 1$ 
13:       $t \leftarrow t + 1$ 
14:    end while
15:     $a \leftarrow \frac{a}{\lambda}$                                  $\triangleright$  Accumulated Gradient Normalization step.
16:     $\text{COMMIT}(a)$ 
17:     $\theta_t^k \leftarrow \text{PULL}()$ 
18:  end while
19: end procedure

```

---

<sup>1</sup>Note if  $\lambda = 1$ , AGN is in essence equivalent to DOWPOUR.

An interesting thought experiment would be what would happen in the case that the workers would not communicate with the parameter server at all, that is,  $\lambda = \infty$ . How would the normalized accumulated gradients look like in such a situation, described by Equation 3.2?

$$\lim_{\lambda \rightarrow \infty} -\frac{\sum_{i=0}^{\lambda} \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_{\theta} \mathcal{L}(\theta_i; x_{ij}; y_{ij})}{\lambda} \quad (3.2)$$

In order to completely understand how the worker deltas would look like after  $\lambda = \infty$  steps, one first needs to understand the individual components of Equation 3.2. The most inner component,  $\eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_{\theta} \mathcal{L}(\theta_i; x_{ij}; y_{ij})$ , is just the computation of a mini-batch using  $m - 1$  samples, where index  $i$  denotes the current step in the gradient accumulation. Please note that a mini-batch can differ for different values of  $i$  as training samples are randomly retrieved from the dataset. After computing the gradient based on the mini-batch, the local model will be updated as  $\theta_{i+1} = \theta_i - \eta_t \frac{1}{m} \sum_{j=0}^{m-1} \nabla_{\theta} \mathcal{L}(\theta_i; x_{ij}; y_{ij})$ . This process goes on for  $\lambda$  steps, while at the end, the accumulated is normalized with respect to  $\lambda$ .

Let us assume we have a convex error space, or a non-convex error space with at least a single minima. Due to the existance of a minima, first order gradients will eventually converge.

## 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 incorporate 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

# Experiments

This chapter describes the experiments, and their results, we conducted to obtain empirical 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.

## 5.1 Distributed Keras

## 5.2 Handwritten Digit Classification

## 5.3 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  $\chi^2$  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 dealt with. A simple, and more common solution would be to sample a more manageable 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 negligible. 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.

### 5.3.1 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 sacrificing dimensionality of the parameterization of the model, we had to limit the data representation to a two-dimensional approach. In this two-dimensional 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.

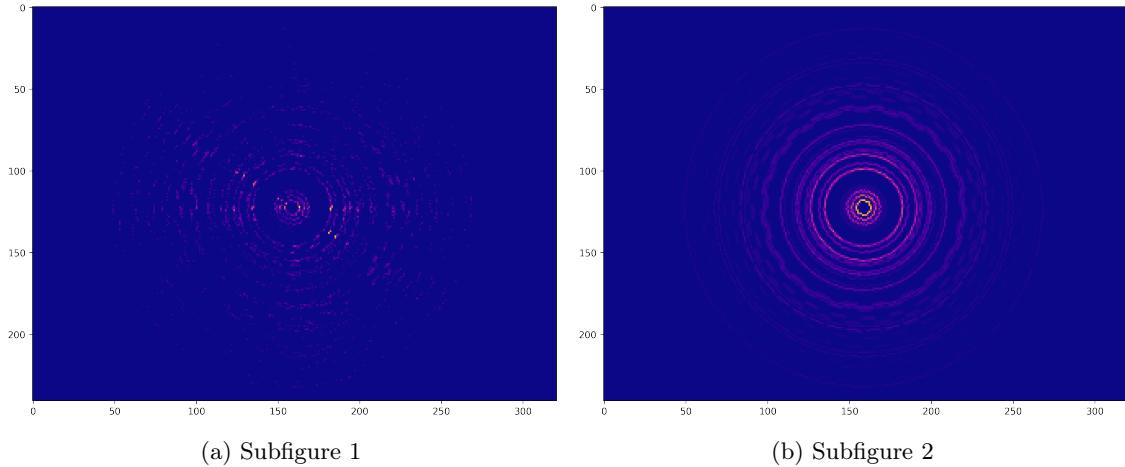


Figure 5.1: Pixel-binned data representation of the hits which occurred in two different collisions.

### 5.3.2 Model Development

# Chapter 6

## 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

