Master Thesis

On Scalable Deep Learning and Parallelizing Gradient Descent

Joeri R. Hermans

Preface

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

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

<div align="right">

Joeri R. Hermans
Geneva, Switzerland 2016 - 2017

</div>

Abstract

Abstract here.

Summary

Summary here.

Contents

Abbreviations and Notation

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

Chapter 1

Introduction

In this chapter we introduce the main concept, and problems surrounding the parallization of gradient descent. We familliarize 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 [12, 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, 11, 9] with the intention to improve detection quality, or reduce computational constraints.

However, the sheer size of these datasets severly impacts the training time of the models. In order to resolve this issue, one could sample some representative subset of the data to reduce the training time. The disadvantage of this approach is that some instances, i.e., data points, might not appear in the final training set. This is especially a problem in Deep Learning, where models usually benifit from having access to a lot of training data due to the high dimensionality of the parametrization [4]. To resolve this issue, Dean et al. [4] introduce two new paradigms to decrease the training time of a large model. The two paradigms, *Model Parallelism*, briefly discussed in Section 1.2, and *Data Parallelism*, discussed in Section 1.3, are inherently different ways of decreasing the training time of a model.

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

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

A lot of different distributed optimization schemes have been suggested in recent years [13, 4, 5]. Most of the recent contributions try to push the limits of asynchronous data parallelism, discussed in Section 3.2, by simply *annealing* the gradients with respect to some hyperparameter to improve the convergence when the number of workers increases. This suggests that there is an intrinsic limit to asynchronous data parallelism, as suggested by [10]. As a result, why don't we simply reduce the number of parallel workers if we reduce the impact of the gradient updates by means of annealing anyway? The approach of reducing the number of parallel workers in such a situation has been suggested by [5], where they perform a *grid-search* of the training hyperparamers (this includes the number of workers) in order to provide the optimal hyperparameters within a training epoch. However, the disadvantage of this technique is that after every epoch, or a specific number of iterations, a grid-search of the hyperparameters has to be performed in order to obtain the optimal configuration of the hyperparameters to ensure convergence.

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

1.2   Model Parallelism

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

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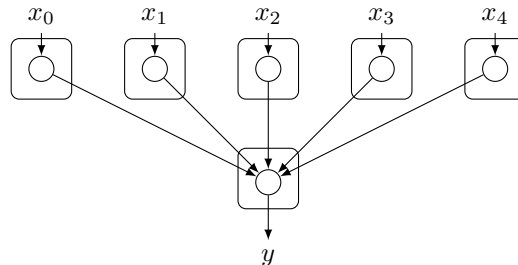

Figure 1.1: A perceptron partitioned using the *model parallelism* paradigm. In this approach every input node is responsible for accepting the input x_i from some source, and multiplying the input with the associated weight w_i. After the multiplication, the result is sent to the node which is responsible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

1.3 Data Parallelism

Data parallelism is an inherently different methodology of optimizing parameters. As stated above, it is a technique to *parallelize gradient descent*, and thereby reducing the overal training time of a model. In essence, data parallelism achieves this by having n workers optimizing a central model, and at the same time, processing n different shards (partitions) of the dataset in parallel over multiple workers[1]. The workers are coordinated in such a way that they optimize the parametrization of a central model, which we denote by $\tilde{\theta}_t$. The coordination mechanism of the workers can be implemented in many different ways. Nevertheless, a popular approach to coordinate workers in their task to optimize the central objective, is to employ a centralized *Parameter Server* (PS). The sole responsibility of the parameter server is to aggregate model updates coming from the workers (*worker commits*), and to handle parameter requests (*worker pulls*). In general, there are several approaches towards data parallelism, where some do not require a parameter server. However, all approaches can be categorized into two main groups, i.e., *Synchronous Data Parallelism*, and *Asynchronous Data Parallelism*.

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

$$\theta_{t+1} = \theta_t - \eta_t \odot \frac{\sum_i \nabla_\theta \mathcal{L}(\theta_t \; ; \; \mathbf{x}_i \; ; \; \mathbf{y}_j) + \sum_j \nabla_\theta \mathcal{L}(\theta_t \; ; \; \mathbf{x}_j \; ; \; \mathbf{y}_j)}{m}$$
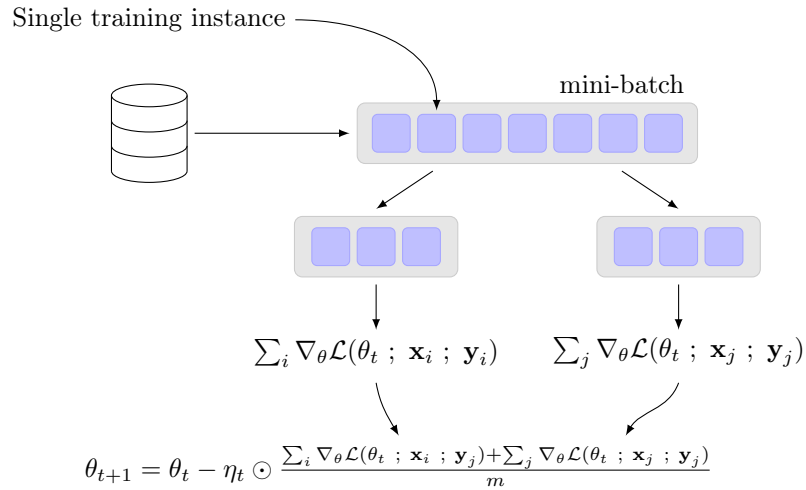
Figure 1.2: Mini-batch parallelism could be viewed as an instance of synchronous data parallelism without a centralized parameter server. Given a mini-batch size of m, we split the mini-batch into several partitions, where a specific worker is responsible for the computation of its own partition. The synchronous nature of this approach lies within the aggregation of the computed gradients, i.e., the results of all workers need to be aggregated, and afterwards averaged in order to integrate the current gradient into the model.

---

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

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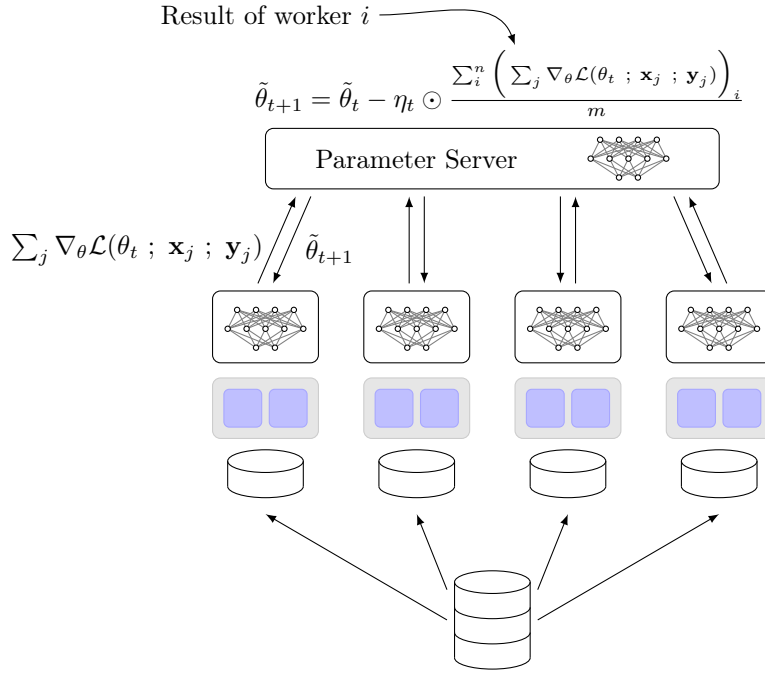

Figure 1.3: Distributed mini-batch data parallelism with n parallel workers, with a total mini-batch size of m. At the start, the data is split into n partitions, and all workers get a copy of the central model with parameterization $\tilde{\theta}_0$. When the training starts, every worker i computes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

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

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

However, experiments have shown that removing the synchronization barrier actually allows models to converge [4, 13, 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* [10]. The main reasoning behind implicit momentum, which will be discussed in detail in Section 3.2.1, is based on a very simple but powerful idea. The idea states that *memory arizes from asynchrony*. Intuitively, this implies that "memory" of previous gradients is preserved due to *stale gradient* updates. This can be observed directly from Figure 1.4, where the update of w_2 is actually updating the central variable with a gradient identical to $\nabla_\theta \mathcal{L}_1(\tilde{\theta}_t)$ if we assume that both workers computed the gradient based on the same input data. This is a clear indication that asynchronous data parallelism is *implicitly* (because of the asynchronous nature of the approach) adding *momentum* which is proportional to the number of workers, since adding more workers actually *adds* more "memory" about previous parameterizations. The authors formalize this by probabilistically estimating the expected change between $\tilde{\theta}_{t+1}$ and $\tilde{\theta}_t$. Using some additional additional assumptions, such as the expected staleness $\mathbf{E}[\tau] = (n-1)$, and geometrically distributed staleness, the authors are able to formalize the expected update in an asynchronous setting between update t and $t+1$, which is shown in Equation 1.1.

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

Using Equation 1.1, we can immediatly derrive the term which describes the implicit momentum induced by asynchrony, which is $\left(1 - \frac{1}{n}\right)$. This result actually suggests that there is a limit to asynchronous optimization: since every problem has some optimal momentum term, which implies that there is an optimal number of asynchronous workers for a specific problem. In order to push the limits of asynchronous optimization, the authors propose various techniques to reduce the abundant amount of implicit momentum. One approach is to apply a *grid-search* to find the optimal hyperparameterization for a given epoch, this also includes the number of workers. Despite that this technique finds the optimal hyperparameterization for a given epoch, the disadvantage is that after every fixed number of iterations, a grid-search of the hyperparameters has to be performed to ensure (optimal) convergence. This is actually in accordance with training in non-data parallel settings, where one starts with a smaller momentum hyperparameter because the gradients at the start will be relatively large compared to the gradients near an optimum, where usually one benifits from a larger momentum hyperparameter. From this intuition we can actually deduce that when the gradient updates are large compared to

the gradients close to an optimum, an optimizer does not benifit from high parallelism because the workers are committing a gradient which was based on a parametrization which is "far" from the current central variable, thus obtaining implicit momentum. Furthermore, one could eliminate the need for the gridsearch by constructing a distributed optimization scheme which is based on the idea described in Hypothesis 1.

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

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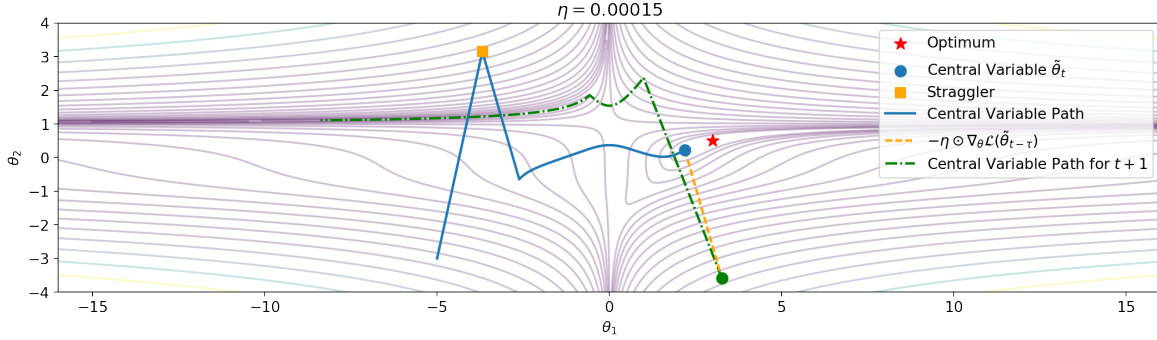

Figure 1.5: Asynchronous optimization procedure applied to Beale's function. In this experiment we introduce a straggler programatically (square) at the start of the optimization process. Despite the fact that this is a low probability event (large staleness compared to the number of workers) in real-world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

One could of course argue, why not use a smaller number of workers since we are annealing the gradients which are based on non-local variables anyway, thus wasting computational resources on those machines? This is certainly a valid argument. However, let us first consider the hyperparameter grid-search approach suggested by [10]. 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 5, this is not the case since the gradients will be annealed dynamically based on the curvature of the error space and current parametrization of the central variable, i.e., there is no need for a hyperparameter grid-search during the training process.

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

In general, all data parallel approaches share a similar training procedure, i.e., every worker computes some variable which is communicated with the parameter server to update the central model. In most cases, this variable represents some change $\Delta\theta$ which needs to be applied to the central variable $\tilde{\theta}_t$. However, some approaches such as [13], actually require that the complete worker parametrization θ_t^k is sent to the parameter server. To abstract this specific optimizer detail, we denote the variable that is sent to the parameter server by v. This procedure is described in Algorithm 1 and 2.
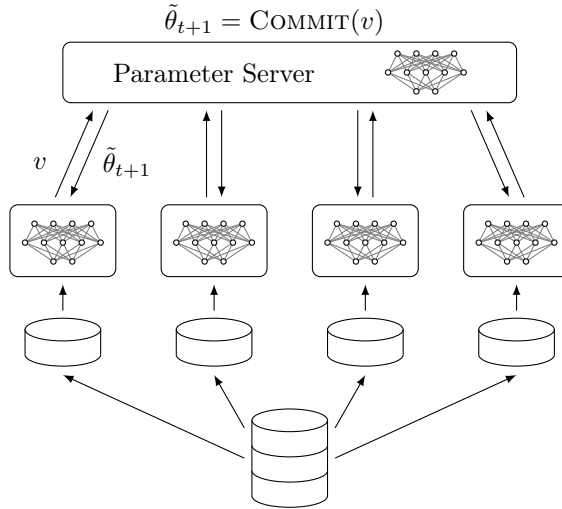

Figure 1.6: Schematic representation of a data parallel approach. In this methodology we spawn n workers (not necessarily on different machines), and assign a data shard (partition) of the dataset to every worker. Using this data shard, a worker i will iterate through all mini-batches to produce a gradient,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

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

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

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

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

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:      $\theta_0^k \leftarrow$ PULL()
3:      $t \leftarrow 0$
4:      **while not** converged **do**
5:          $\mathbf{m} \leftarrow$ 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 [8], or other optimizer.
7:          $v \leftarrow$ PREPARECOMMIT()
8:          COMMIT(v)
9:          $\theta_t^k \leftarrow$ PULL()
10:        $t \leftarrow t + 1$
11:     **end while**
12: **end procedure**

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

1: **procedure** INTIALIZEPARAMETERSERVER(θ)
2:      $\tilde{\theta}_0 \leftarrow \theta$
3:      $t \leftarrow 0$
4: **end procedure**
5:
6: **procedure** COMMIT(v)
7:      $\tilde{\theta}_{t+1} \leftarrow$ 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

TODO

1.5 Thesis Outline

TODO

Chapter 2

Optimization Algorithms

Chapter 3

Distributed Deep Learning

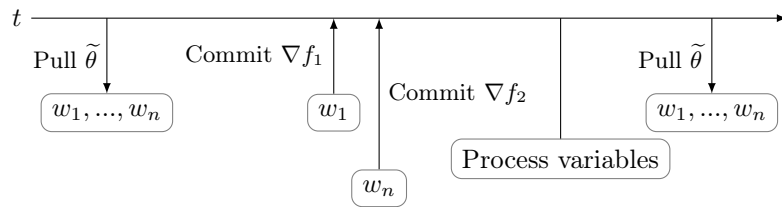## 3.1 Synchronous Data Parallelism

Figure 3.1: Caption here
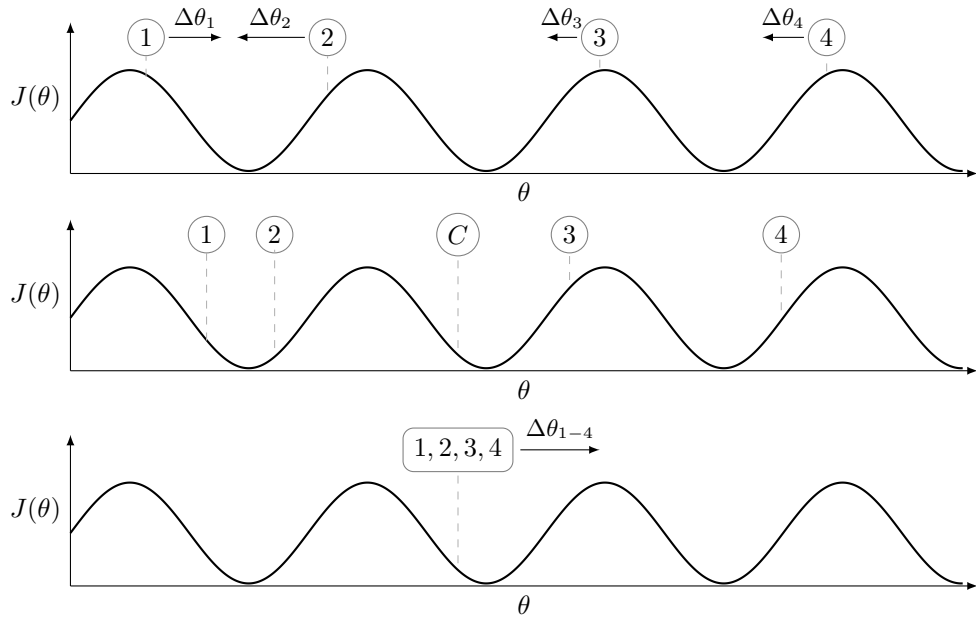
3.1.1 Model Averaging

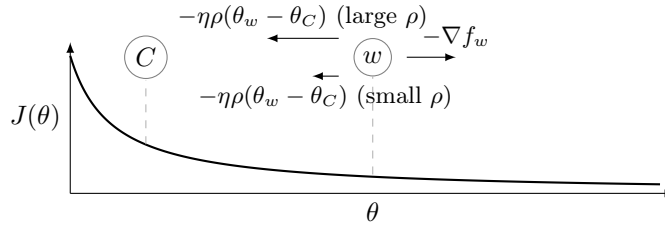Figure 3.2: Caption here

3.1.2   Elastic Averaging SGD

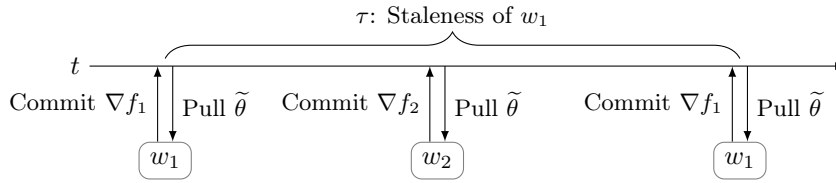Figure 3.3: EASGD Caption here

3.2   Asynchronous Data Parallelism

Figure 3.4: Caption here.

3.2.1   Asychrony Induced Momentum

3.2.2   Hogwild!

3.2.3   DOWNPOUR

3.2.4   Asynchronous Elastic Averaging SGD

Chapter 4

Accumulated Gradient Normalization

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

Chapter 5

Asynchronous Distributed Adaptive Gradients

In this chapter we introduce a novel optimizer called ADAG. ADAG, or *Asynchronous Distributed Adaptive Gradients*, is an optimization process designed with data parallel methods in mind. We build upon previous work [4, 5, 8, 13] and incorperate new insights backed up by theory and experimental evidence. We start in Section 5.1 by formalizing the problem setting. In Section 5.2, we summarize previous work on distributed (data parallel) optimization. Section 5.3 will describe our algorithm in detail, supported by intuition and theory. Finally, we experimentally show the effectiveness of our approach in Section 5.4 and give some points for future work in Section 5.5.

5.1 Problem setting

5.2 Previous work

5.3 Algorithm

5.3.1 Update rule

5.4 Experiments

5.4.1 Handwritten digit classification

5.4.2 Higgs event detection

5.4.3 Sensitivity to hyperparameters

5.4.4 Sensitivity to number of parallel workers

5.5 Future work

Chapter 6

Distributed Keras

Chapter 7

Experiments

Chapter 8

Conclusion

Bibliography

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

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

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

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

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

[11] 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).

[12] 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).

[13] 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