UNIVERSITY OF TORINO

DOCTORAL SCHOOL ON SCIENCE
AND HIGH TECHNOLOGY

COMPUTER SCIENCE DEPARTMENT

DOCTORAL THESIS

Deep Learning at Scale with Nearest Neighbours Communications

Author:
Paolo VIVIANI

Cycle XXXI

Supervisor:
Prof. Marco ALDINUCCI

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

"Training with large mini-batches is bad for your health."

– Yann LeCun, 2018

– Randall Munroe, 2017

Printed September, 2019

UNIVERSITY OF TORINO

Abstract

Computer Science Department

Doctor of Philosophy

Deep Learning at Scale with Nearest Neighbours Communications

by Paolo VIVIANI

As deep learning techniques become more and more popular, there is the need to move these applications from the data scientist's Jupyter notebook to efficient and reliable enterprise solutions. Moreover, distributed training of deep learning models will happen more and more outside the well-known borders of cloud and HPC infrastructure and will move to *edge* and mobile platforms. Current techniques for distributed deep learning have drawbacks in both these scenarios, limiting their long-term applicability.

After a critical review of the established techniques for Data Parallel training from both a distributed computing and deep learning perspective, a novel approach based on nearest-neighbour communications is presented in order to overcome some of the issues related to mainstream approaches, such as global communication patterns. Moreover, in order to validate the proposed strategy, the Flexible Asynchronous Scalable Training (FAST) framework is introduced, which allows to apply the nearest-neighbours communications approach to a deep learning framework of choice.

Finally, a relevant use-case is deployed on a medium-scale infrastructure to demonstrate both the framework and the methodology presented. Training convergence and scalability results are presented and discussed in comparison to a baseline defined by using state-of-the-art distributed training tools provided by a well-known deep learning framework.

Acknowledgements

Reaching the end of a PhD track is a collaborative effort, in the sense that the unfortunate people surrounding a PhD student participate in the effort by withstanding the fallout of all the challenges that the candidate is trying to navigate through.

This kind of support can come in different shapes, but the more people around me endured my lunacy, the more I consider myself privileged to have enjoyed such company.

I owe many thanks to who, on different sides, had to physically carry out part of my work to let me to write this thesis. In particular, I would like to thank my colleagues Marco and Riccardo, who allowed me to spend way too much time focused on writing without getting fired. Also, I owe a huge thank you to Maurizio, who made my work also his work, and without whom this thesis would never have seen the light. Thanks for the long talks despite the time zone, and thanks for always forcing me to put some order in my messy ideas. I also ought to thank who eventually contributed hands-on to produce the results shown here: my largely smarter academic successor Iacopo, and the unfortunate victim of my around-the-clock requests Daniele.

I can't help but be grateful to who initially bet on me for this experiment (how naive I was to take the bait!): Marco Aldinucci and Roberto d'Ippolito. Thanks Marco for tolerating my scattered contribution to the cause and for always being supportive despite that; thanks also for the long talks about how to name things, that always led to new and interesting ideas. Thanks Roberto for your unbreakable confidence in the success of this endeavour, and for all the invaluable knowledge you provided in order to safely navigate funded research projects. Eventually, this turned out to be a winning bet for me and I sincerely hope that it also paid out to both of you.

Many thanks are due to the reviewers, and to anyone had the patience to go through all my thesis to provide valuable comments to improve this work.

While some recipients of these acknowledgements contributed to my PhD course because they had to, some other voluntarily chosen to stay by my side: to them I owe the largest debt of gratitude. My parents always shown indestructible trust in my capabilities and endured my mood swings with the most solidarity. Mom, Dad, thank you.

To my friends, I owe apologies and thanks, since they are still treating me as a family member even after recurrently disappearing for variable time spans. I'm proud of the grown up men that you have become, and I'm glad that we can still be stupid together in spite of that.

Finally, the hardest task is to express my gratitude to my life partner for the last 10 years without being trivial. Of course you supported me, you withstood my complaints, and you kept me running when I was about to lose confidence (or too lazy), but that's not the point: I'm so grateful because you, Anna, make every day spent on your side way better than it would be without, whether I'm working hard to finalise a paper or we are relaxing on the couch of our new home. I really can't wait to keep dancing the "Walk of Life" with you.

Contents

1 Introduction **1**
- 1.1 Main contributions . . . 1
- 1.2 Collateral results . . . 4
  - 1.2.1 Funded projects . . . 4
  - 1.2.2 Industrial research . . . 5
  - 1.2.3 Other publications . . . 7
  - 1.2.4 Funding . . . 7

2 Background **9**
- 2.1 Deep Learning . . . 9
  - 2.1.1 Supervised Learning and Backpropagation . . . 9
  - 2.1.2 Gradient Descent Training . . . 11
- 2.2 Software for Deep Learning . . . 19
  - 2.2.1 Automatic differentiation and computational graphs 19
  - 2.2.2 Common features . . . 20
  - 2.2.3 Typical training process . . . 21
  - 2.2.4 Performance comparison . . . 21

3 Performance and concurrency in deep learning **23**
- 3.1 Overview . . . 24
- 3.2 Network parallelism . . . 24
  - 3.2.1 Layer computation and backpropagation . . . 24
  - 3.2.2 Model parallelism . . . 26
- 3.3 Data parallelism . . . 27
  - 3.3.1 Mini-batches and GPUs . . . 27
  - 3.3.2 Distributed training . . . 28
- 3.4 State of practice . . . 35
  - 3.4.1 Implementations . . . 35
  - 3.4.2 Performance metrics . . . 36
  - 3.4.3 Summary . . . 37

4 Nearest Neighbours Training **39**
- 4.1 Limitations of current techniques . . . 39
  - 4.1.1 Synchronous methods . . . 39
  - 4.1.2 Asynchronous approaches and model consistency . . 40
- 4.2 Proposed approach . . . 42
  - 4.2.1 Sparse topology training theory . . . 44
  - 4.2.2 Nearest-Neighbours Training . . . 45

5 Flexible Asynchronous Scalable Training Framework **49**
- 5.1 Design . . . 49
- 5.2 Software components . . . 50
  - 5.2.1 Communication Layer . . . 50
  - 5.2.2 Topology, Processors and Communicators . . . 52

5.2.3 GFastFlow . . . 53
5.2.4 Node-level parallelism . . . 58
5.3 Training node . . . 60
5.3.1 Control flow and structure . . . 60
5.3.2 Gradients transfer . . . 62
5.4 Usage . . . 62
5.4.1 User model definition and compilation . . . 62
5.4.2 Execution . . . 64

6 Evaluation **65**
6.1 Experimental set-up . . . 65
6.1.1 Hardware used . . . 66
6.1.2 Baseline . . . 66
6.1.3 Competitors . . . 68
6.1.4 Nearest-Neighbours Training . . . 68
6.2 Results . . . 69
6.2.1 Time-To-Accuracy . . . 69
6.2.2 Scalability . . . 72
6.3 Summary . . . 74

7 Conclusion **77**
7.1 Remarks and conclusion . . . 77
7.2 Future work . . . 78

Chapter 1

Introduction

Deep Learning has largely dominated the machine learning debate during the last few years: in fact it promises, and most of the time delivers, astonishing results for a large number of tasks that are considered challenging by computer scientists, leading to a significant exposure of these results even outside the academic world. This exposure, coupled with the significant effort spent by large companies to push forward both industrial and academic research on deep learning and Artificial Intelligence (AI), may trick the casual practitioner into the ill-founded belief that this methodology is well understood, established, and somehow almighty. Diving deeper into these topics, however, unveils a reality where researchers are able to accomplish tremendous tasks by means of Deep Neural Networks (DNNs), but the efficiency of this approach still involves some degree of trial-and-error tuning of the optimisation hyper-parameters. Despite many remarkable efforts, several aspects of this matter are still unclear on many different levels: from the definition of the right network architecture for a given problem, to the dynamic of the optimisation involved in the training. Given the point of view of the parallel computing researcher, this translates to a buzzword-driven approach to performance evaluation, and to an unsatisfactory understanding of the theoretical framework where all the strategies to train neural networks in parallel belong to.

This work, far from intending to fill all the gaps in the theory of deep learning, aims to shed some light upon a few theoretical and practical aspects of the parallel training of DNNs, with the hope to give more sturdy foundations to the deployment of the new breed of AI workloads that is taking over High Performance Computing (HPC) machines all over the world.

1.1 Main contributions

The task of choosing an interesting research problem in the field of parallel deep learning can be daunting, given the number of open questions faced by researchers every day. This work approaches the field from the most elementary concepts, and tries to put some order by identifying those aspects that are orthogonal to each other, and hence can be tackled independently, while proceeding step by step to narrow the focus towards a specific problem.

The problem at the core of this work lies in the theoretical and practical limitations of current *data parallel* approaches to the training of deep neural networks. While the details of this matter will be explored later on, its relevance cannot be understated given how fast these techniques are growing in popularity. In fact, there is the need to move these applications from

the data scientist's Jupyter notebook to reliable and efficient enterprise solutions, without falling into the temptation to accept bombastic statements from large commercial players proposing their own "silver bullet" to address challenging AI workloads. In this context, this work tries to lay down the path to achieve faster time-to-result, decoupling the problem of parallel training from the specific learning task, software framework, and hardware architecture.

Review of theoretical framework

Much work has been done to enable training of DNNs on parallel architectures, yet most of the previous research was made from the perspective of domain experts, as an attempt to improve the performance of a specific use-case, without the intention to provide a general approach. This led to a limited amount of theoretical research in this field, and even less attempts to summarise the literature. The first, and to date the only, comprehensive review of this topic dates back to February 2018 [1]. This thesis intends to go beyond the enumeration of different strategies and it will try to critically understand their strengths and weaknesses and, above all, will try to identify the classification criteria that can actually provide useful insight to forecast and improve real-life performance of parallel training.

Chapter 2 will review the theory of deep learning, it will discuss the mathematics behind key aspects that have an impact on the training performance, and it will provide a first theoretical contribution in the discussion about mini-batch size choice for training. Finally, it will briefly review the main software tools for deep learning.

Chapter 3 will review the techniques of concurrent training from a more practical point of view and it will delve into the detail of data parallel training, which sits at the core of this work, discussing the usual classification based on the model consistency spectrum and reviewing the major results in the field. It will explore the approaches summarised in figure 1.1 to identify their suitability to be pushed beyond the current state of the art. It will finally present the implementations provided by major software tools to perform training in a distributed environment.

Nearest neighbours communication training

This topic, originally presented in previous works [2, 3], represents the main methodological contribution of this dissertation. Chapter 4 will build on the results of the previous chapter, further discussing the theoretical limitations of mainstream approaches to parallel training, and defining some common features that can be used to identify a potential direction for improvement. Given this background, the author attempts to envision a strategy to train DNNs that is, at least in principle, not affected by such flaws. To achieve this, the focus is posed on a specific branch of figure 1.1: distributed data parallellism. This focus does not negatively affect the generality and the value of the present work, since most of the listed techniques are not mutually exclusive, hence they can be applied together given sufficient computational resources. Finally, Nearest neighbours communication training is introduced: it requires the training workers in a distributed setup, to communicate gradient updates only to their nearest neighbours in a

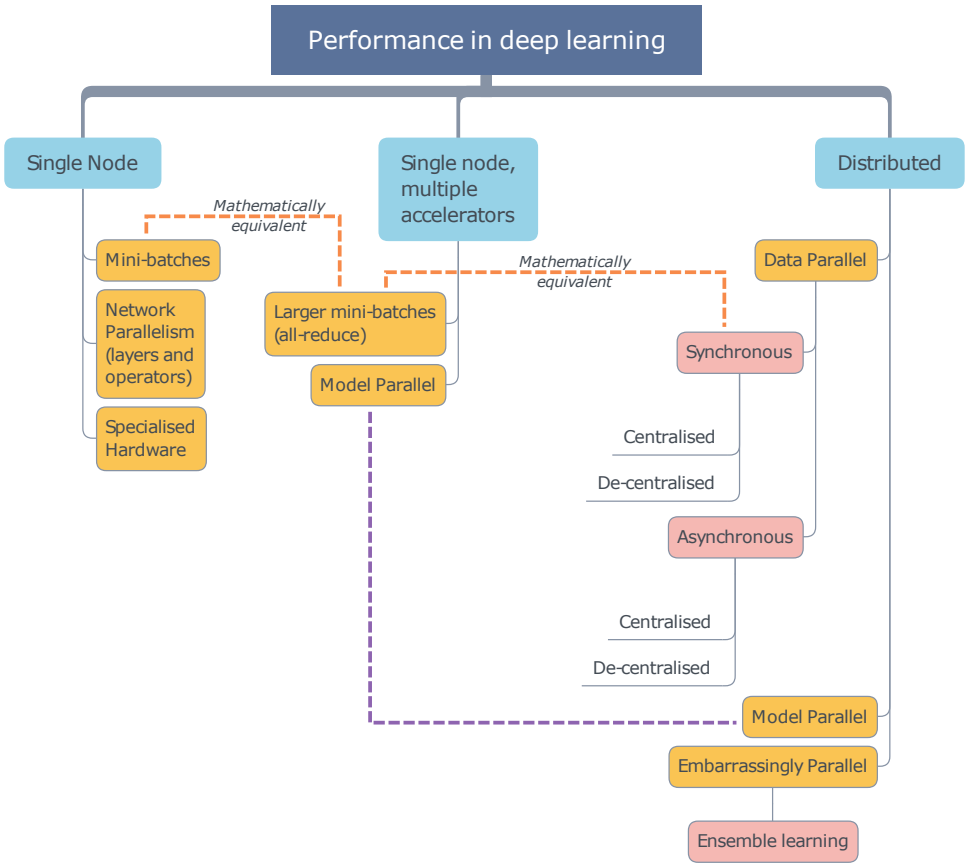


FIGURE 1.1: Brief summary of the available techniques to improve the performance of deep learning. These techniques have been reviewed in this work, and results are presented in chapter 3.

fully asynchronous manner. This is expected to be beneficial for both scalability and training accuracy.

Flexible Asynchronous Scalable Training (FAST) framework

One of the observations that will be presented in chapters 2 and 4 is the fact that theoretical results regarding the convergence of distributed learning algorithms are hardly applicable to forecast the performance of a given real use case. Unfortunately, this work could not make an exception to this rule, hence an experimental validation of the proposed approach is a paramount.

Chapter 5 presents a novel framework designed to allow the user to train deep neural networks in a distributed fashion with limited impact on the user's training code, which can be based on a deep learning framework of choice. The FAST framework is designed to encapsulate the user code written with any Deep Learning (DL) framework for which an interface is provided, capturing gradients at each iteration and exchanging them with neighbour workers; the connectivity overlay that defines the neighbouring relationships can be easily expressed in terms of channels open between workers, and communications are handled in a fully asynchronous, non-blocking manner. This framework allowed the author to eventually test the

methodology introduced above and to compare the results on real problems. Chapter 6 will present the results achieved on a representative dataset and deep learning model from literature, comparing several performance metrics with both the single-node training and the mainstream approach to distributed training.

It is worth noting that this dissertation is the result of a research effort started approximately in January 2018: given the interest surrounding this topic, a large number of publications and software tools were made available during this period, with some of them partially overlapping with the scope of this work. In particular it is worth to mention the following works, both published in March 2018, from Ben-Nun and Hoefler [1] and from Daily et al. [4]: the former presents a comprehensive review of the state of the art in parallel deep learning, which definitely intersects the scope of this dissertation, and therefore it is frequently mentioned here. The latter proposes a very similar approach, based on some common theoretical grounds: nevertheless, still, it is the opinion of the author that the present work retains its relevance regardless the amount of related literature and competing approaches published during the last year. Related works are in general reported and discussed to the best of the author's knowledge, however, the extremely fast publication cycle in this field, mainly based on pre-prints repositories like arXiv, could partially jeopardize this attempt.

1.2 Collateral results

As the bottom line of an industrial PhD program, this thesis should reflect to some degree the whole path traversed by the author to achieve its main scientific contribution. While the rest of this dissertation will present in detail this contribution, it is worth to dedicate a paragraph to resume all the relevant activities, publications, and results achieved during the PhD course, with specific focus on all the, mostly industrial, aspects that are not directly related to the principal contribution.

The author performed the most part of his research activity within the Research and Innovation department of Noesis Solutions[1], a simulation innovation partner to manufacturers in automotive, aerospace and other engineering-intense industries. Specialized in simulation process integration and numerical design optimisation (PIDO), its software provides interfaces for a large number of commercial and open source simulation tools, allowing the customer to automate and integrate it simulation workflow, while capturing knowledge with machine learning techniques that analyse data coming from the workflow itself. Noesis actively participates in many national and European R&D projects targeting the development of new technologies, in which the author has been involved.

1.2.1 Funded projects

The author activity has been related mainly to four funded research projects, in which he covered the role of main technical point of contact for Noesis Solutions, with responsibilities spanning from the development of the

[1] Noesis Solutions NV, Leuven, Belgium (www.noesissolutions.com).

core methodologies required by the project to the presentation of results at project reviews and the writing of deliverables.

MACH ITEA2 project 12002; MAssive Calculations on Hybrid systems [5]. The goal of the project is to develop a Domain-Specific Embedded Language (DSeL) and a computation framework that allows to access hybrid hardware acceleration without specific expertise. The project involved 15 partners from 4 countries, with a funding of 12.5M€.

Fortissimo 2 Horizon 2020-FoF-2015; Fortissimo2 [6] is a collaborative project that will enable European SMEs to be more competitive globally through the use of simulation services running on a High Performance Computing cloud infrastructure. The project involved 38 partners, with a funding from European Commission (EC) of 10M€.

CloudFlow Cloudflow project [7] is aimed to enable the remote use of computational services distributed on the cloud, seamlessly integrating these within established engineering design workflows and standards [7]. The project involved 44 partners, with a funding from EC of 6.6M€.

BoSS Blockchain for Online Service Security [8]. The project involved 8 partners from Belgium and funding from the Flemish government. The following publication resulted from the collaboration

- V. Reniers, P. Viviani, R. Lombardi, D. V. Landuyt, B. Lagaisse, and W. Joosen, "Analysis of architectural variants for auditable blockchain-based private data sharing", in *In proc. of the 34th ACM Symposium on Applied Computing (SAC)*, Limassol, Cyprus, Apr. 2019, pp. 1–8

1.2.2 Industrial research

The main topic investigated by the author is the application of machine learning techniques in the design engineering context, and the deployment of these applications on high-performance architectures, such as heterogeneous hardware with multi-core CPUs and GPUs. The first part of this work has been directed towards the design and implementation of a C++ linear algebra runtime system that gives the developer the capability to perform matrix factorisations either on the CPU or on the GPU, choosing the appropriate architecture at runtime, instead of having to recompile the code. This work started as the main research topic of the MACH project (cf. Sec. 1.2.1) and later was brought into the product by Noesis Solutions, that used the capabilities provided by this framework, to significantly improve the performance of a number of numerical algorithms used to interpolate discrete data. The following publications are the direct result of the author's research on this topic:

- P. Viviani, M. Aldinucci, R. d'Ippolito, J. Lemeire, and D. Vucinic, "A flexible numerical framework for engineering—a response surface modelling application", in *Improved Performance of Materials: Design and Experimental Approaches*. Cham: Springer International Publishing, 2018, pp. 93–106

- P. Viviani, M. Torquati, M. Aldinucci, and R. d'Ippolito, "Multiple back-end support for the armadillo linear algebra interface", in *In proc. of the 32nd ACM Symposium on Applied Computing (SAC)*, Marrakesh, Morocco, Apr. 2017, pp. 1566–1573

- P. Viviani, M. Aldinucci, and R. d'Ippolito, "An hybrid linear algebra framework for engineering", in *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) – Poster Abstracts*, Fiuggi, Italy, Jul. 2016

- P. Viviani, M. Aldinucci, R. d'Ippolito, J. Lemeire, and D. Vucinic, "A flexible numerical framework for engineering - a response surface modelling application", in *10th Intl. Conference on Advanced Computational Engineering and Experimenting (ACE-X)*, 2016

The performance improvement just described, made feasible a number of techniques that would have been otherwise too computationally expensive: as a first result, different strategies have been developed aimed to apply ensemble methods to regression and interpolation models in order to achieve better generalisation accuracy and robustness across different use-case scenarios with respect to the individual models themselves. One of these strategies, based on cross validation, produced very interesting results: its prediction of the "best model" among a pool of available models on a given dataset agrees with ground truth validation a surprisingly high fraction of times. Another version, also based on cross validation data, blends individual models locally across different regions of the domain and provides very robust modelling capabilities to engineers and domain experts. Both these strategies are already included in Noesis' flagship product.

While ensemble methods has been investigated to extract as much information as possible from very sparse datasets, the other end of the spectrum has been discussed too: the state of practice for discrete engineering data interpolation is given by $O(n^3)$ non-parametric regression algorithms, which are not suitable to handle large datasets. On the other hand, cheaper approaches like least squares do not provide the necessary accuracy. In this context, DNNs have been investigated to handle large-scale regression problems in the engineering domain, providing both a methodology and an implementation to let the user exploit DNNs without specific deep learning expertise (i.e. a simple speed/accuracy trade-off slider is provided). The implementation is currently integrated into Noesis software Optimus. State-of-the-art frameworks have been investigated and leveraged to both provide optimal performance on CPU and GPU and to allow C++-based training to protect the code Intellectual Property.

Details of the work on ensemble methods and deep learning regression are confidential and thus not published. On the other hand, the industrial applications of deep learning sparked the interest in the problems related to performance and scalability of such methodologies at large scale. While the approach currently included in Noesis' products does not present particular performance challenges, upcoming research efforts will definitely require large scale training of DNNs in order to leverage the large amount of data acquired by geographically distributed edge devices and sensors that will soon permeate industrial shop floors.

1.2.3 Other publications

The following publications are either directly related to the main topic of this dissertation, or the outcome of the collaboration of different members of the Author's research group[2].

- P. Viviani, M. Drocco, D. Baccega, I. Colonnelli, and M. Aldinucci, "Deep learning at scale", in *Proc. of 27th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP)*, Pavia, Italy: IEEE, 2019
- P. Viviani, M. Drocco, I. Colonnelli, M. Aldinucci, and M. Grangetto, "Accelerating spectral graph analysis through wavefronts of linear algebra operations", in *Proc. of 27th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP)*, Pavia, Italy: IEEE, 2019
- M. Aldinucci, S. Rabellino, M. Pironti, F. Spiga, P. Viviani, M. Drocco, M. Guerzoni, G. Boella, M. Mellia, P. Margara, I. Drago, R. Marturano, G. Marchetto, E. Piccolo, S. Bagnasco, S. Lusso, S. Vallero, G. Attardi, A. Barchiesi, A. Colla, and F. Galeazzi, "Hpc4ai, an ai-on-demand federated platform endeavour", in *ACM Computing Frontiers*, Ischia, Italy, May 2018
- P. Viviani, M. Drocco, and M. Aldinucci, "Pushing the boundaries of parallel deep learning - A practical approach", *CoRR*, vol. abs/1806.09528, 2018
- P. Viviani, M. Drocco, and M. Aldinucci, "Scaling dense linear algebra on multicore and beyond: A survey", in *Proc. of 26th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP)*, Cambridge, United Kingdom: IEEE, 2018
- F. Tordini, M. Aldinucci, P. Viviani, I. Merelli, and P. Liò, "Scientific workflows on clouds with heterogeneous and preemptible instances", in *Proc. of the Intl. Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, ser. Advances in Parallel Computing, IOS Press, 2018

A further publication reporting the results presented in this dissertation has been submitted for review to the special issue of *Future Generation Computer Systems* titled "On The Road to Exascale II: Advances in High Performance Computing and Simulations".

1.2.4 Funding

This work has been partially supported by the ITEA2 project 12002 MACH, the EU FP7 REPARA project (no. 609666), by the HPC4AI project funded by the Region Piedmont POR-FESR 2014-20 programme (INFRA-P) [9], and by the OptiBike experiment in the H2020 project Fortissimo2 (no. 680481). Experimentation has been possible thanks to the *Competency Center on Scientific Computing* (C3S) at University of Turin [10], to Compagnia di SanPaolo for the donation of the OCCAM heterogeneous cluster, and to resources

[2] Parallel computing group, Computer science Department, University of Turin (alpha.di.unito.it)

provided by the Pawsey Supercomputing Centre with funding from the Australian Government and the Government of Western Australia.

Chapter 2

Background

2.1 Deep Learning

Deep Learning [11] is Machine Learning (ML) technique that focuses on learning data representations in order to avoid the expensive feature engineering phase usually required by conventional ML algorithms. This allows the model to be fed with raw data and to automatically discover the features needed for detection or classification. DL models are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods [12] have significantly improved the state-of-the-art in image classification and object detection [13–15], speech recognition [16, 17], machine language translation [18, 19] and many other domains such as drug discovery and genomics [20, 21].

2.1.1 Supervised Learning and Backpropagation

In order to understand the mechanics that regulates Deep Learning, it is useful to introduce the most trivial declination of ML: *supervised learning* [22]. Supervised learning involves the collection of a large set of *labelled* data (i.e. a set of input data, each one with one or more ground truth *labels* already associated, possibly by a human) which is then shown to a model that in turn produces a set of *predictions* associated to each input. The predictions and the labels are then compared, and the model is iteratively corrected in order to provide predictions as similar as possible to the labels. Figure 2.1 represents the general setting of supervised learning. This is a very high-level description of a process that involves several subtleties, starting from the definition of input themselves, up to the metric that is used to define the distance between predictions and labels in the output space. Moreover, it is important to understand that this optimisation

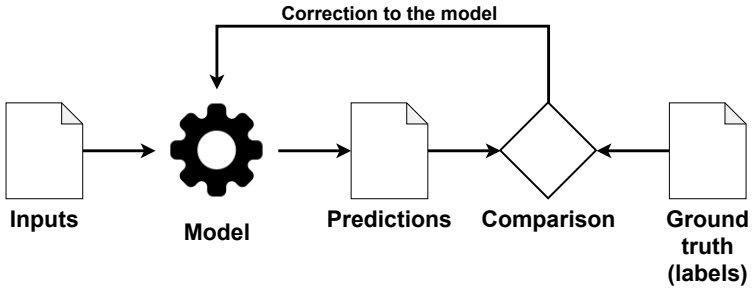
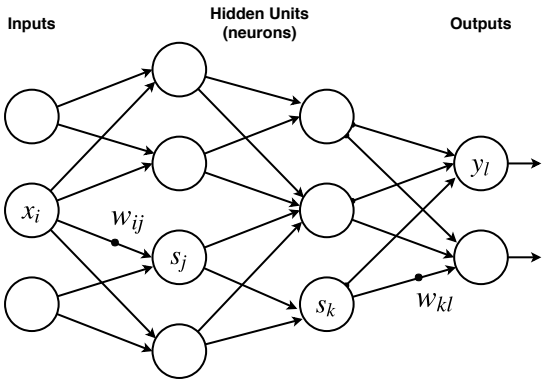
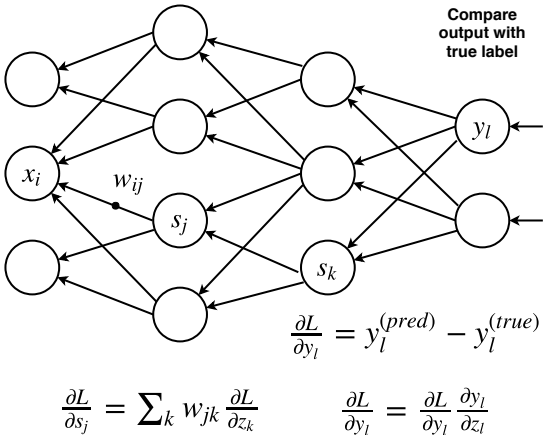


FIGURE 2.1: Typical workflow of supervised learning.

process is bound to minimise the error made by the model fed with training data, hence the distance between predicted labels and ground truth, however there is no guarantee that the same model will predict previously unseen data as well as its fitness with respect to the training data would suggest. This is the typical instance of the approximation-generalisation trade-off [23, p. 62] that is pivotal in machine learning, sometimes this issue can be referred to as the gap between *in-sample error* and *out-of-sample error* (i.e. error calculated w.r.t. training data vs. error calculated w.r.t. previously unseen data).



(A) Feedforward Artificial Neural Network. The calculation of neuron activations is shown below the network.



(B) Backpropagation process. Below the network is outlined the derivation of weight gradients from the comparison of predictions and labels. E represents the network's loss function.

FIGURE 2.2: Depiction of a feedforward, backpropagation multi-layer neural network [11]. The x_i represent input values, w_{ij} are the weights associated to the edges of the network, and y_i are the output values. z represents the neuron *activations*, which are fed to the *activation function* f to obtain the output of the neuron s. Different subscript indices are used for different layers.

Classical machine learning uses algorithms like linear regression, logistic regression and support vector machines [23] to learn correlations in a

given dataset, on the other end Deep Learning relies on multi-layer Artificial Neural Networks (ANNs). An artificial neural network is a collection of Rosenblatt's Neurons (Perceptrons) [24], which can be seen as a loose model of biological neurons, arranged in layers and connected by "synapses" (edges). These neurons usually implement some kind of non-linear *activation function* that transforms the signal (typically real numbers) received from input edges into different values to be further transformed downstream. The non-linearity of the activation function guarantees that it makes sense to stack multiple neurons in a sequence, as otherwise it would be possible to squash any sequence into a linear model. Figure 2.2a represents the transformation of input values into output values when flowing through a simple neural network made of two layers (*feedforward*). Usual implementations of neural networks associate a *weight* to each edge of the network; these weights are the parameters that are adjusted in a supervised training process in order to let the model fit the training data.

Multi-layer neural networks are in fact a chain of function compositions: hence it is possible to express the derivative of the error, defined as the distance between labels and predictions (*loss*), with respect to individual weights. This makes possible to express a gradient of the loss function in the space of weights, which can be followed in order to *train* (namely, optimise) the weights themselves to minimize the loss function. The process of calculating gradients starting from the value of the loss function is called *backpropagation* (Fig. 2.2b) [25–28].

Understanding the shape of the loss function and the path followed by the gradient-based optimisation are key points in successful training of DNNs and, even more, in distributed training. For long time it has been widely considered infeasible to train deep architectures effectively without the gradient vanishing [29] or the optimisation being trapped in some poor local minima. The former issue has been solved by a careful selection of the activation function [30] and the advent of faster hardware for training (cf. sec. 3). The latter instead appeared to not being an issue at all for large networks, as most of the time the system reaches solutions of very similar quality regardless of the initial condition, instead, saddle points where gradient is zero and the second derivative has opposite sign in different dimensions are quite common in the loss landscape [31].

Neural networks, as all machine learning models, present a set of parameters that are related to the structure of the model and to the training process, and they are not parameters of the model in the statistical sense. The number and the topology of layers, the depth of the network, as well as the parameters of the optimisation algorithm used for the training, are all identified as *hyper-parameters* of the model. Needless to say that they play a critical role in successful training of *ml* models.

2.1.2 Gradient Descent Training

The previous paragraph introduced the the concept of optimising model parameters by means of gradients computed with backpropagation. While gradient descent is a trivial optimisation algorithm, the peculiarities of deep neural networks make the task daunting: the number of weights for a state-of-the-art model can be as high as several tens of millions [1], that means a non-linear optimisation problem in a domain of the same dimensionality.

On the other hand, some surprisingly trivial optimisation methods turned out to be very effective to achieve good training and generalisation results. At this point some notation is useful before elaborating further on this topic.

For the rest of this dissertation, if not explicitly stated, the following conventions will be used: the input dataset is defined as

$$X = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$$

where each $\mathbf{x}_i \in X$ is a vector in the input space, that can be as simple as $\mathbb{R}^n$, or the array of gray levels for a the pixels composing an image. In the same way the labels can be described as

$$Y = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$$

where $\mathbf{y}_i \in Y$ are also vectors in the space of labels (i.e. for simple multi-class classification, a they can be unit vectors with ones in the position of the class represented by the label, and zeros in all the other positions). Each pair of correlated input-outputs $(\mathbf{x}_i, \mathbf{y}_i)$ is a *sample* in the dataset (X, Y) and is used to train a neural network represented by a collection of weights (typically real-valued, but they can be discrete for some applications [32, 33])

$$\mathbf{w} = \{w_1, \ldots, w_m\}.$$

Possibly, a bias term b_i is also present on each edge of the network, so that the equation for z_i of figure 2.2a becomes

$$z_j = \sum_i x_i w_{ij} + b_j \tag{2.1}$$

however, this term will be understood hereafter for simplicity of the notation. At this stage it is not yet relevant to explicit the network topology, as the following notation applies for any well known architecture. The loss function is defined as

$$L = L\left(\mathbf{y}_i^{(pred)}, \mathbf{y}_i\right), \qquad \mathbf{y}_i^{(pred)} = \text{Feedforward}(\mathbf{w}, \mathbf{x}_i)$$

where the argument $\mathbf{y}_i$ can be understood in favour of the following notation

$$L = L\left(\mathbf{w}, \mathbf{x}_i\right). \tag{2.2}$$

The loss is typically defined as $L : \mathbb{R}^m \to \mathbb{R}$ and its form is chosen depending on the specific task, such as Mean Squared Error (MSE) for regression or cross-entropy for classification [34]. The backpropagation allows to compute the gradient of the loss function with respect to the weights:

$$\nabla L(\mathbf{w}, \mathbf{x_j}) = \left(\frac{\partial L\left(\mathbf{w}, \mathbf{x_j}\right)}{\partial w_1}, \ldots, \frac{\partial L\left(\mathbf{w}, \mathbf{x_j}\right)}{\partial w_m}\right)$$

this represents the direction of steepest slope of the loss surface calculated with respect to $\mathbf{x_j}$ in the parameter's space.

It important to note that the loss is function of the weights *and* of the specific sample, but the objective of learning is usually to minimise the loss

over all the dataset, hence the target function of the optimisation is

$$L(\mathbf{w}, X) = \frac{1}{N} \sum_{i=0}^{N} L\left(\mathbf{w}, \mathbf{x}_i\right)$$

and the gradient is given by

$$\nabla L(\mathbf{w}, X) = \frac{1}{N} \sum_{i=0}^{N} \nabla L\left(\mathbf{w}, \mathbf{x}_i\right) \tag{2.3}$$

this means that, in order to compute the full gradient, it is necessary to execute the feedforward on all the samples in the dataset. The gradient descent optimisation step can be expressed as

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla L(\mathbf{w}(t), X) \tag{2.4}$$

where η represents the *learning rate*, namely the length of the step taken in the direction of steepest descent along the loss surface. Unfortunately, using the full gradient as in equation (2.3) requires a significant amount of computation, while usually resulting in poor generalisation results [35].

A significant breakthrough in deep learning has been represented by the application of Stochastic Gradient Descent (SGD) [36] to the training process, allowing faster convergence with less computational effort [28]. This process is also known as *on-line* gradient descent, as opposed to to the *batch* gradient descent described by equation (2.4) and involves the approximation of the full gradient by computing it on a single sample $x \in X$. Under the assumption that x are sampled as Independent and identically distributed random variables (i.i.d.) from X with probability density $p(x)$, it is in fact possible to state that

$$L(\mathbf{w}, X) = \mathbb{E}_{\mathbf{x} \in X}\left[L(\mathbf{w}, \mathbf{x})\right] = \int p(x)\, L(\mathbf{w}, \mathbf{x})\, dx$$

where $\mathbb{E}_x(f(x))$ is the expected value of function f for a randomly sampled variable x. From this equation it can be proven that the gradient computed on a single sample is a good approximation for the full gradient: in fact, for each component of the gradient it is possible to write

$$\begin{aligned} \frac{\partial L(\mathbf{w}, X)}{\partial w_i} &= \frac{\partial}{\partial w_i} \int p(x)\, L(\mathbf{w}, \mathbf{x})\, dx = \\ &= \int p(x)\, \frac{\partial L(\mathbf{w}, \mathbf{x})}{\partial w_i}\, dx = \mathbb{E}_{\mathbf{x} \in X}\left[\frac{\partial L(\mathbf{w}, \mathbf{x})}{\partial w_i}\right] \end{aligned}$$

and hence for all the components

$$\nabla L(\mathbf{w}, X) = \mathbb{E}_{\mathbf{x} \in X}\left[\nabla L(\mathbf{w}, \mathbf{x})\right]. \tag{2.5}$$

Replacing the dataset X with the whole input domain and defining E as the generalisation (out-of-sample) error and $\mathcal{D}$ the input domain such that $X \subset \mathcal{D}$, the same proof holds and allows to write the following equation

$$E(\mathbf{w}, \mathcal{D}) = \mathbb{E}_{\mathbf{x} \in \mathcal{D}}\left[L(\mathbf{w}, \mathbf{x})\right]$$

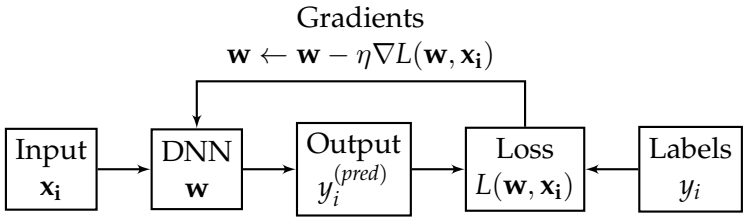


FIGURE 2.3: Representation of supervised deep learning with stochastic gradient descent.

$$\nabla E(\mathbf{w}, \mathcal{D}) = \mathbb{E}_{\mathbf{x} \in \mathcal{D}} \left[\nabla L(\mathbf{w}, \mathbf{x}) \right] \tag{2.6}$$

then is possible to conclude that *the stochastic gradient is an unbiased estimator of the generalisation error gradient*. Unfortunately, this is only partially true, as usually loss and generalisation error are correlated, but overfitting is always around the corner [23].

For on-line optimisation, subsequent steps based on individual components of the gradient are computed as following

$$\begin{aligned} \mathbf{w}(t+1) &= \mathbf{w}(t) - \eta \nabla (\mathbf{w}(t), \mathbf{x}_i) \\ \mathbf{w}(t+2) &= \mathbf{w}(t+1) - \eta \nabla (\mathbf{w}(t+1), \mathbf{x}_{i+1}) \\ &\dots \\ \mathbf{w}(t+k+1) &= \mathbf{w}(t+k) - \eta \nabla (\mathbf{w}(t+k), \mathbf{x}_{i+k}) \end{aligned} \tag{2.7}$$

where one step is taken for each sample extracted from the dataset. This approach proved to be extremely effective and, despite several attempts to develop different optimisation strategies [12], either second-order methods or not gradient-based, it is still the base of most of the state-of-the-art deep learning applications [35, 37–39]. A common approach for the optimisation is to go through all the samples without replacement multiple times before reaching a satisfactory result, in this case each run through the whole dataset is referred to as an *epoch*. Figure 2.3 reformulates the same concept introduced in figure 2.1, detailing the mathematical formulation involved in the backpropagation-SGD loop for training of neural networks. Both figure 2.3 and equation (2.7) show how the gradient value depend on the present $\mathbf{w}(t)$ configuration and how its application through back-propagation produces a new configuration $\mathbf{w}(t+1)$: the new weights represent a data dependency for the feed-forward step for sample x_{i+1}, that must come strictly after the back-propagation, otherwise the gradient would be calculated based on outdated (*stale*) weights. This makes on-line gradient descent intrinsically sequential, affecting the capability to exploit parallelism for training.

Mini-batch gradient descent

While the stochasting gradient descent is very effective for such a trivial strategy, its optimisation path is usually quite noisy. It is possible to mitigate this noisy behaviour by combining the advantages of both stochastic and batch gradient descent. *Mini-batch* gradient descent [40, 41] computes

the loss based on a subset $X_{(i,n_b)} = \{\mathbf{x}_i, \ldots, \mathbf{x}_{i+n_b-1}\}$ of size n_b (the mini-batch) of the training data as following

$$\frac{1}{n_b}\sum_{j=i}^{i+n_b-1}\nabla L(\mathbf{w}, \mathbf{x_j}) \stackrel{\text{def}}{=} \delta L(\mathbf{w}, X_{(i,i+n_b-1)})$$
$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta\delta L(\mathbf{w}(t), X_{(i,i+n_b-1)}) \tag{2.8}$$

where the gradient has been averaged by dividing the sum by n_b. Note that *mini-batch averaging*, as opposite of just summing, has a non-trivial impact on the convergence of the training [37].

Apart from the convergence properties, the main advantage of this approach is the fact that the feedforward/backpropagation steps for all the samples in the mini-batch can be executed in parallel. In particular equation (2.1) and its backpropagatin couterpart can be more effectively expressed as matrix multiplications and element-wise operations[28], and computed efficiently by means of multi-threaded General Matrix Multiplications (GEMMs) and optimised data structures on suitable architectures [42–44]. From this point of view it appears how mini-batch gradient descent is the simplest form of parallel training for deep neural networks. The convergence of the training given the mini-batch size and the relationship between mini-batch optimisation and other techniques of parallel and distributed training will be a pivotal point in this dissertation.

It is interesting to remark that mini-batch gradient descent is the most general case of gradient descent training, as both SGD and batch gradient descent are special cases with mini-batch size $n_b = 1$ and $n_b = N$ respectively. Hereafter the distinction between stochastic, mini-batch and batch gradient descent will be dropped for brevity, referring in general to SGD and specifying the mini-batch size n_b where relevant.

Equation (2.8) represents the simplest form of SGD. Several refinements have been developed to improve the convergence rate of DNNs training, a good review of them can be found in literature [34, 45]. The key points of these evolved algorithms are:

1. variable learning rate, $\eta \to \eta(t)$;
2. accounting for previous gradient steps (e.g. *momentum* [46]);
3. defining a different learning rate for each weight $\eta(t) \to \eta(t, w_k)$ (e.g. ADAM [47]).

Momentum SGD is worth to be quickly reviewed, as it is a commonly adopted modification to the plain SGD. It is defined recursively as

$$\mathbf{w}(t+1) = \mathbf{w}(t) - v_t$$

where

$$v_t \stackrel{\text{def}}{=} \mu\, v_{t-1} + \eta\, \delta L(\mathbf{w}(t), X)$$

This formulation leads to the accumulation of previous SGD steps to, loosely speaking, build up some kind of inertia in the optimisation process, which turns out to be beneficial for the convergence. The typical pictorial representation of the effect of momentum is shown in figure 2.4.

(A) Without momentum. (B) With momentum.

FIGURE 2.4: Pictorial representation of the effect of momentum to a convex optimization baed on previous literature [45].

Mini-batch size

Defining the right mini-batch size for training is by itself a complex optimisation task: several aspects must be considered, the most relevant one is the *generalisation vs. utilisation* trade-off [1]. This kind of tradeoff is also pivotal for this work, and regards the capacity of a model to achieve good out-of-sample performance even when trained with larger and larger batch sizes in order to exploit parallelism.

The typical batch size used for training is $1 < n_b \ll N$, with $n_b = 32$ as a "magic number" that is usually suggested to achieve good convergence [37]; in general the consensus in the deep learning community is that the mini-batches should neither be too large nor too small, behaviour depicted by figure 2.5. This idea comes either from experience and from some theoretical investigations, the first contribution of this work, carried out in this paragraph, is to review some theoretical points and to discuss their real-life impact to the training process.

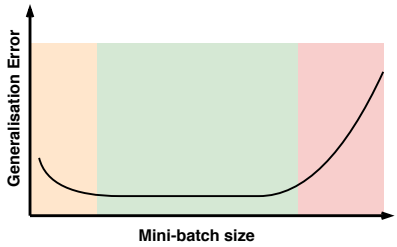


FIGURE 2.5: Illustration of common consensus about generalisation performance vs. mini-batch size. [1]

An important work that advocates the use of small batches, to the limit of on-line training, has been published by Wilson and Martinez [35]. The first observation about this work is that, despite showing how $n_b = 1$ has the best convergence rate in terms of number of epochs needed to reach a given accuracy, it does not take into account the time taken to reach that accuracy. In fact, with modern *many-core* architectures almost perfect scalability is within reach up to the saturation of the architecture (e.g. one or two hundred samples, to give an order of magnitude for a typical case). This means that, given N_{seq} epochs necessary to reach a given accuracy for the on-line version, the mini-batch version is convenient up to $N_{batch} = n_b \cdot N_{seq}$ epochs, assuming perfect mini-batch scalability. Other works advocate the optimality of on-line training in terms of convergence rate [48], however, the performance penalty of a fully sequential computation is so high that pure SGD is rarely used in practice.

The case against large mini-batches is more nuanced: while some more works definitely show an advantage of smaller n_b [49], others present effective training even with batches of several thousands of samples [50–54]. The theoretical existence of an upper bound to the batch size (the red region in figure 2.5) is stated by Ben-Nun and Hoefler [1], who state that previous works exploiting very large batches are only pushing this upper bound further, but not removing it. This work upholds a slightly different claim: in this sense it is useful to try to model how the generalisation error evolves for a training process. This derivation will follow the path of some previous literature[48, 55–57], with the hope of providing a more intuitive take on the actual optimisation behaviour. Some necessary assumptions are needed, which are mostly standard in this context:

Assumption 1 (Unbiased estimator)**.** *As in equation (2.6), the loss function computed on one sample $L(\mathbf{w}, \mathbf{x})$ is taken as an unbiased estimator for the generalisation loss $E(\mathbf{w}, \mathcal{D})$.*

$$E(\mathbf{w}) = \mathbb{E}_{\mathbf{x}}\left[L(\mathbf{w}, \mathbf{x})\right]$$

Where the domain $\mathcal{D}$ has been dropped for brevity.

Assumption 2 (Bounded variance)**.** *It is assumed that the variance of the unbiased estimator of assumption 1 is bounded by a constant σ^2.*

$$\mathbb{E}_{\mathbf{x}}\left[\left\|L(\mathbf{w}, \mathbf{x}) - E(\mathbf{w})\right\|^2\right] \leq \sigma^2, \quad \forall\, \mathbf{x}.$$

Assumption 3 (Lipschitz continuity of loss)**.**

$$\left\|\nabla E(\mathbf{w}_1) - \nabla E(\mathbf{w}_0)\right\| \leq \mathcal{L}\left\|\mathbf{w}_1 - \mathbf{w}_0\right\|, \quad \forall\, \mathbf{w}_0, \mathbf{w}_1.$$

Using assumption 3 and the *descent lemma* for Lipschitz-continuous functions [58] it is possible to write the evolution of the generalisation loss for a gradient descent step $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{n_b} \nabla L(\mathbf{w}_t, \mathbf{x}_i)$

$$\begin{aligned} E(\mathbf{w}_{t+1}) &\leq E(\mathbf{w}_t) - \eta\, \nabla E(\mathbf{w}_t) \cdot (\mathbf{w}_{t+1} - \mathbf{w}_t) + \frac{\mathcal{L}}{2}\left\|\mathbf{w}_{t+1} - \mathbf{w}_t\right\|^2 \\ &\leq E(\mathbf{w}_t) - \eta \sum_{i=1}^{n_b} \nabla E(\mathbf{w}_t) \cdot \nabla L(\mathbf{w}_t, \mathbf{x}_i) \\ &\quad + \frac{\eta^2 \mathcal{L}}{2}\left\|\sum_{i=1}^{n_b} \nabla L(\mathbf{w}_t, \mathbf{x}_i)\right\|^2 \end{aligned}$$

at this point it is convenient to define $\mathbf{d} \stackrel{\text{def}}{=} \sum_i^{n_b} \left(L(\mathbf{w}, \mathbf{x}_i) - E(\mathbf{w})\right)$, therefore the last inequality can be rewritten as

$$\begin{aligned} E(\mathbf{w}_{t+1}) &\leq E(\mathbf{w}_t) - \eta\left[\nabla E(\mathbf{w}_t) \cdot \mathbf{d} + n_b \left\|\nabla E(\mathbf{w}_t)\right\|^2\right] \\ &\quad + \frac{\eta^2 \mathcal{L}}{2}\left\|\mathbf{d} + n_b\, \nabla E(\mathbf{w}_t)\right\|^2 \\ &\leq E(\mathbf{w}_t) - \eta\left[\nabla E(\mathbf{w}_t) \cdot \mathbf{d} + n_b \left\|\nabla E(\mathbf{w}_t)\right\|^2\right] \\ &\quad + \frac{\eta^2 \mathcal{L}}{2}\left[\left\|\mathbf{d}\right\|^2 + 2 n_b \mathbf{d} \cdot \nabla E(\mathbf{w}_t) + n_b^2 \left\|\nabla E(\mathbf{w}_t)\right\|^2\right] \end{aligned}$$

In order to understand the evolution of the test error, it is possible to move to the expected values and use assumptions 1 and 2 to obtain $\mathbb{E}_x[\mathbf{d}] = \vec{0}$ and $\mathbb{E}_x\left[\|\mathbf{d}\|^2\right] = \sigma^2$

$$\mathbb{E}_x[E(\mathbf{w}_{t+1})] \leq E(\mathbf{w}_t) - \underbrace{\eta n_b \|\nabla E(\mathbf{w}_t)\|^2}_{\text{descent term}} + \underbrace{\frac{\eta^2 \mathcal{L}}{2}\left[\sigma^2 + n_b^2 \|\nabla E(\mathbf{w}_t)\|^2\right]}_{\text{variance term}} \quad (2.9)$$

Equation (2.9) shows how using a bigger n_b is beneficial as increases the descent term, but it also increases *quadratically* the variance term, which in fact hinders the convergence of generalisation error. This conclusion has also been drawn by Ben-Nun and Hoefler [1], however the author argues that, by applying the popular idea of batch gradient averaging instead of a plain summation , as in equation (2.8), the variance term becomes

$$\frac{\eta^2 \mathcal{L}}{2}\left(\frac{\sigma^2}{n_b^2} + \|\nabla E(\mathbf{w}_t)\|^2\right)$$

where n_b has also the effect of suppressing the sampling variance.

The previous result apparently contradicts the idea of an upper bound to the mini-batch size, moreover it looks like the suppressing effect on the variance term might be beneficial to the training. However, transferring this result in the real-life training of DNNs is tricky. In fact, when using large batches is hardly enough to average the gradients instead of summing: practical applications [50] reflect the need for much more careful and problem-specific fine tuning of hyper-parameters with respect to smaller batches. Moreover, the variance that is suppressed by n_b^2 is often times beneficial in order to escape some bad local minimum or saddle point at the beginning of the training, when the behaviour is heavily non-convex. It is the author's opinion that this fact is definitely related to the success of a technique used by works exploiting very large batches: they usually find beneficial to perform some kind of *warm-up* [50] phase, running earlier epochs with smaller batches. In the light of the results derived in this section, this can be traced to a transition from a non-convex regime where the stochasticity of smaller batches is beneficial, to a convex regime of optimisation, where larger batches are more effective.

As an additional remark, it can be noted that Ma et al. [48] leverage the same formalism that led to equation (2.9) in order to proof that $n_b = 1$ is the optimal size in term of total samples (namely, the number of epochs) required to achieve a certain result, however, the drawback of this analysis is the same as the aforementioned one from Wilson and Martinez [35]: computing n_b samples in a batch cost significantly less than computing n_b times one sample, hence the number of epochs required to converge may be larger, but it would be amortised by the much smaller time required to compute such epochs. At least from a theoretical point of view.

This discussion highlights how theory does not really provide solid ground to define the right approach to mini-batch size and subsequently, as this hyper-parameter has a critical impact on the DL scalability (cf. Chapter 3), there is no clear theoretical guidance to the approach to follow in that case too. On the other hand, from the general consensus emerged from

literature, where the final answer is always delegated to experimental results, the author is eventually convinced that there is no theoretical upper bound to the size of mini batches, but there are practical ones. From the additional fine-tuning needed, to poor results achieved in the initial phases of the training, stretching mini-batches has a significant impact on the training curves that requires some mitigation effort. Moreover, perfect speedup for large batches is not always straightforward due to additional issues that will be discussed in section 3.3, like batch normalisation.

This section presented the formalism and the aspects that will be relevant in the following discussion: a thorough discussion of the topic from the basics of statistical learning is out of the scope of this work. For the reader interested in deepen his understanding of the theoretical foundations of machine learning, the book by Abu-Mostafa et al. [23] is a good starting point. With more focus on deep learning, the most comprehensive work is provided by Goodfellow et al. [34].

2.2 Software for Deep Learning

The recent success experienced by deep learning methods had also the effect of animate the community of researchers and developers, that in turn provided a large number of software tools dedicated to deep learning. This paragraph will provide a brief overview of the most relevant of them and their common feature.

2.2.1 Automatic differentiation and computational graphs

Backpropagation can be seen as a special case of automatic differentiation [59, 60], which is a technique that is used to evaluate the derivative of a function defined as a computer program. This technique exploits the fact that any function can be, in principle, described as a sequence of arithmetic operations which is subject to the chain rule for the derivative of composed functions. This concept underlies all the current mainstream deep learning framework like those listed below:

- Theano [43]
- Caffe [61]
- Google's TensorFlow [62]
- Facebook's PyTorch [63]
- Apache MxNet [64]
- Microsoft's CNTK [65]

In fact, all of them implement some kind of (*dataflow-like* [66]) graph describing the control flow and data structures of the neural networks, on which automatic differentiation can be applied to calculate gradients [34, p. 201].

Among the tools mentioned above, Theano and caffe are the only ones to be mainly developed by an academic institution (Montreal Institute for

Learning Algorithms, Université de Montréal and Berkeley Artificial Intelligence Research Lab) and they were the first to gain widespread adoption. The interest around the topic of deep learning and AI drove large enterprises to invest in such tools, for internal deployment at first, and then open sourced to the community. At the time of writing, the development of Theano has stopped due to the maturity of industrial competitors and their faster development pace.

2.2.2 Common features

Most of the mentioned software tools not only share the same theoretical foundation, but they are in fact very similar in terms of structure and components that they implement: they are usually written in C/C++, with Application Programming Interfaces (APIs) provided in multiple languages, with Python that is by far the most popular one in the data science community. Their API usually allows to directly interact with the computational graphs, and while some of them provide a higher-level API out of the box, others do that by means of additional libraries, being Keras [67] the most popular one. They also provide a large catalogue of operators to define specialised neural network units and layers for image processing (e.g. convolutions [68]) and for natural language processing and speech recognition (like recurrent neural networks [25, 69]).

Graphics Processing Unit (GPU) acceleration is another common point: while the detail of these topic will be discussed in the next section, it is relevant to remark that all the major deep learning frameworks provide access to GPUs for training and inference, usually by means of proprietary tools provided by the hardware vendor like the Nvidia CUDA toolkit [70] and its specialised libraries for linear algebra and deep learning: cuBLAS [70] and cuDNN [44].

Symbolic vs. Imperative

The only significant design choice that differentiates these framework is the approach to the computational graph definition API, that can be either *symbolic* (Theano, Caffe, TensorFlow, MxNet symbolic API) or *imperative* (PyTorch, MxNet Gluon API) [71]. The relevance of this categorisation lies in the large use of such terms when presenting DL software tools, which usually make the use of one of these approaches their main selling point.

The symbolic approach defines the computational graph and executes it in different steps, in this sense most symbolic-style programs contain, either explicitly or implicitly, a compile step this converts the graph into a function that can be called. The model of the DNN is fully defined before mapping it to actual data structures allocated in memory. The imperative (or dynamic) approach allocates and executes the graph as soon as it is written in the code, allowing the developer to use native control flow within the definition of the graph. Symbolic frameworks are in principle more efficient as they are fully defined up-front and they can benefit from more optimisations, however most frameworks nowadays offer both APIs and several ways to get the best of both worlds. Finally, this categorization is more relevant for who actually develops new DNN architectures, or at least for who is concerned with node-level performance. Distributed deep learning is not

really affected by the choice made at this level, provided that the neural networks architecture is static; topologies that change during the training may affect the capability to correctly communicate gradients and weights between different workers, as their size and shape may change.

2.2.3 Typical training process

Algorithm 1 presents a pseudocode that describes the typical training process for a deep learning model, as it is structured in most of the major frameworks. The main features to be noted in this pseudocode are the initialisation phase (lines 1-5) where the training data is loaded, model is created from a symbolic graph, either created by the developer of loaded from a file, and it is compiled if needed by the API. The there are two nested loops that go over all the mini-batches in the dataset (line 8) as many times (line 7) as the give number of epochs for the training.

Algorithm 1 Pseudocode for typical training process

```
1:  net ← Symbolic computation graph
2:  data_iterator ← training dataset
3:  model ← net.compile(input size, output size)
4:  model.initialise_parameters()
5:  opt ← optimiser(learning rate)
6:
7:  for number of epochs do
8:      while data_iterator has next mini-batch do
9:          batch ← data_iterator.nextBatch()
10:         predictions ← model.forward(batch.input())
11:         model.backward()
12:         opt.update(model, predictions, batch.output())
13:         print accuracy(predictions, batch.output())
14:     end while
15:     data_iterator.restart()
16: end for
```

2.2.4 Performance comparison

Given the number of similarities it is not surprising that the performance of the major frameworks is very similar, this is particularly true when training on GPUs, where the heavy lifting is offloaded to the same computing back-end by all of them. To the best of the author's knowledge, the last comprehensive benchmark comparing the performance of those tools has been published by Shi et al. [72]. However, development is so fast that those results are possibly not relevant any more. Moreover, those tools are so complex and depending on different libraries that compiling and optimising them to achieve the best performance on a given hardware is not trivial, especially on CPUs. For instance, the author experienced a 20 times speedup for MxNet on CPU by changing a single compilation flag, that included an Intel-tuned library to accelerate deep learning operations [73].

The single node performance of these tools, given careful compilation and deployment, is so similar that the choice of one of them for a given task

should be based on different factors other than performance, for instance the suitability of the API for a certain language and task. The common opinion, though, is that MxNet and CNTK are more "production oriented", while TensorFlow and PyTorch are more "research oriented", however the latter is catching up quickly, as in version 1.0 included most of the features of the Facebook's C++ production framework Caffe2.

Chapter 3

Performance and concurrency in deep learning

This chapter will discuss what are the main factors that affect the performance of deep learning tasks, and what are the chances to exploit parallel architectures: in this context it is remarkable that the directions to achieve a performance improvement concerning a deep learning model are so many that it is not possible to explore all of them in detail in a single work, hence a brief overview will be given, reserving the right to further elaborate on the points that are directly related to this dissertation. However, the interesting upside of this variety is that most of these techniques are in fact orthogonal to each other: the verticalisation of one technique, does not affect the capability to exploit other directions for further improvement, given sufficient problem granularity and enough computational resources.

The discussion about performance and parallelism in DNNs will start from the most elementary building blocks of the neural networks, up to multi-node parallelism at large scale. However, before proceeding to investigate concurrency opportunities, it is useful to briefly discuss the two main categories of deep learning workload to better frame the scope of this work.

Training vs. Inference

While the focus of deep learning research has mostly been on the training phase, the wider adoption in production contexts highlighted a whole new set of requirements for deep learning workloads, such as the need to perform *inference* (namely, prediction) based on an pre-trained model in a highly time-constrained or power-constrained environment. This led to the distinction of training and inference as the two main deep learning workloads. This works focuses on the training workload, that will be discussed in detail in the following paragraphs, however it is worth to highlight the peculiarities of the inference workload and the main differences with training.

The typical setting for DL training is a controlled, possibly research-focused environment, where computational resources are abundant (e.g. cloud infrastructure, HPC centre, with GPUs available) and time is only a loose constraint. Data is usually available in batch mode, namely randomly and quickly accessible, and the granularity of the task is quite large: limited by the size of mini-batches.

On the other end, inference is usually based on models trained in advance that are deployed in production environments and that are required to give answers to customers or other devices in short time frames (up to

real-time). Data in this environment is usually available as a stream, where the granularity is as small a single sample at time. A typical use case for the inference workload is within autonomous vehicles or for mobile applications that leverage computer vision or speech recognition. These scenarios require very fast responses and usually set very tight constraints on the amount of computational power, let alone the power consumption itself. Usually inference workloads leverage specialised, low power, hardware architectures like Field Programmable Gate Arrays (FPGAs), Application-specific integrated circuits (ASICs) or others [74, 75], and software specific for inference [76].

In short, the performance focus for training is on throughput (e.g. number of images processed per second), while the key to inference performance is latency (e.g. response time for the prediction of a single sample). The focus of this work is strictly on the training phase.

3.1 Overview

Figure 3.1 presents all the relevant approaches that have been explored in literature and practice to improve the performance of DNN training. It also resumes the main remarks about the specific approaches that will be presented in detail in the following paragraphs.

The different techniques are aggregated based on the scale of their respective parallelism: in fact, as happens for most scientific computing applications, it is possible to achieve performance improvements by working on a single node, by means of optimised neural network operations that leverage multiple cores, vector extensions, or dedicated hardware like GPUs; by adding multiple accelerators on the same node (*scale-up*); or by distributing computation on multiple computing nodes (*scale-out*).

It should be remarked that this classification is not the only choice: in fact it is also possible to aggregate techniques based on the similarity of the theoretical approach: for instance, model parallelism is a declination of network parallelism that usually refers to a specific set of techniques used in a multi-device or distributed set-up. The discussion below does not strictly follow the order that these two possible classifications may suggest, on the other hand it tries to build a logical path that touches all the relevant points. Finally, this work is focused on distributed data parallel training, hence the detailed discussion of other techniques is in part delegated to the references provided.

3.2 Network parallelism

3.2.1 Layer computation and backpropagation

As already stated, the operations described in figure 2.2 can be expressed as a chain of matrix multiplications and element-wise operations, but this is straightforward only in case of fully-connected layers, namely layers where each unit receives inputs from all the units in the previous layer. For convolutional [68] and recurrent [25] layers it is not as trivial, however linear

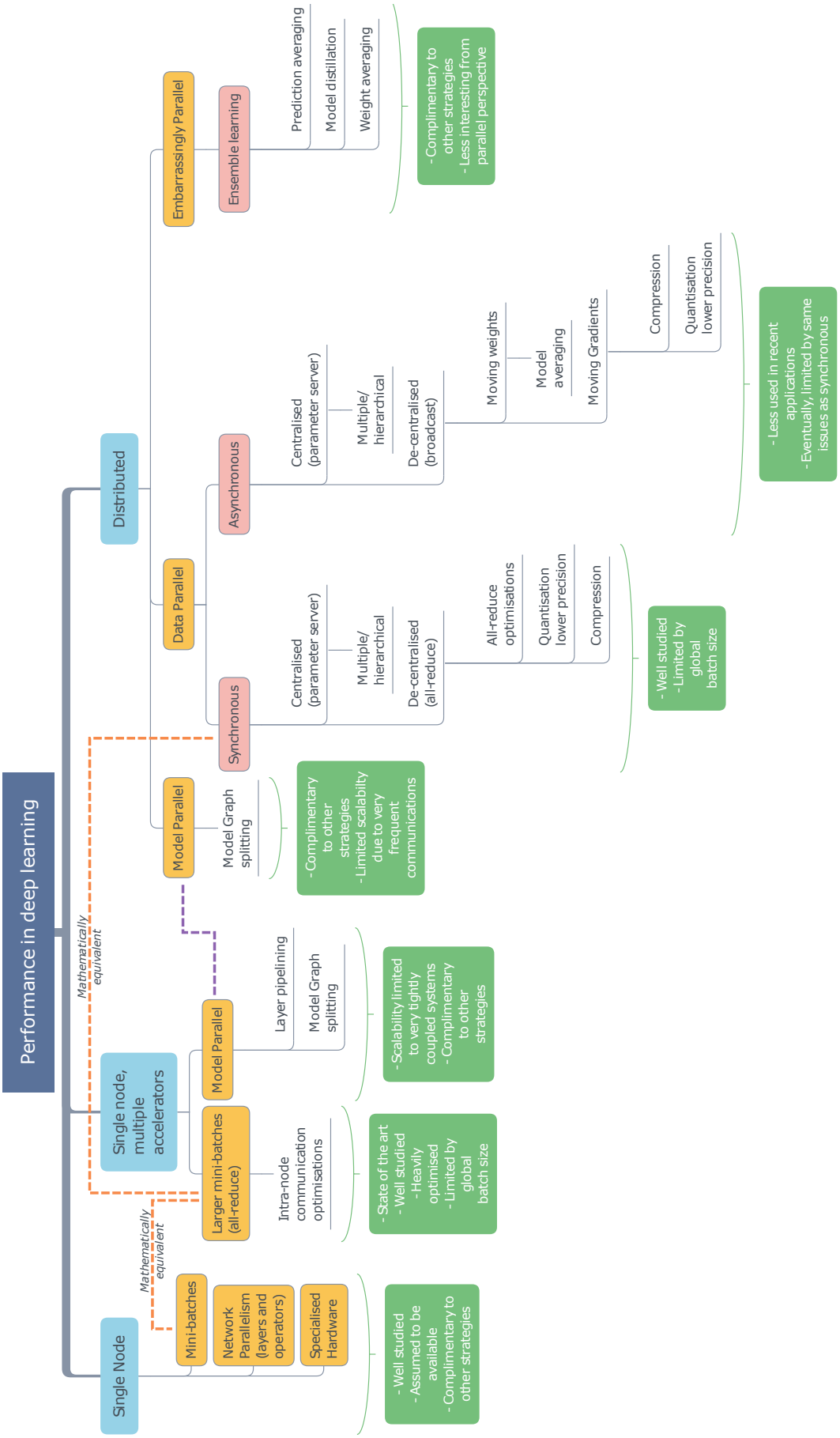


FIGURE 3.1: Map of different approaces to deep learning

algebra routines are still at the core of deep learning workloads. The typical input of a deep neural network is a *tensor* with a certain number of indices. For instance, considering the case of image classification, the indices are usually four: index of the sample in the mini-batch, vertical position of the pixel, horizontal position of the pixel, and index of the color channel. This structure can be exploited to parallelise the execution of layers of the DNN and, as already stated, this operation is trivial in the case of fully-connected layers where the major factor impacting the performance is the implementation of the GEMM routine. While the API of GEMM is a *de-facto* standard set by the Basic Linear Algebra Subprograms (BLAS) library [77], nowadays many implementations are available, the most popular being Intel MKL [78], OpenBLAS [79] and cuBLAS [70] for GPU.

Convolutional layers usually take the largest share of computation in image classification tasks, but they are not directly suitable to be expressed as matrix multiplication, hence there is usually an intermediate step that transforms the convolution kernels into a different function that is suitable to be executed as a batch of multiplication-accumulation operations on highly parallel architectures. Thereare several ways to transform convolution layers, the most relvant are: the so-called *im2col* [80], which transforms the convolution into a matrix multiplication using *Toeplitz matrices*; the *implicit GEMM* method, that saves on memory by not materialising the Teoplitz matrix [44]; the Fast Fourier Transform (FFT) method [81]; and the Winograd method [82], which is the prevalent approach nowadays. All of these different approaches have strenghts and weaknesses: software implementing deep learning primitives, like Nvidia cuDNN [44] and Intel MKL-DNN [73], usually choose among several implementations of convolutional filters based on the specific features of the workload, possibly runnig internal benchmarks to pick te best performing one. Also the order of the indices in the input tensor (that represents the memory layout of the data) plays a role in the performance of the convolution [83]. Recently, Vasilache et al. [84] introduced a technique based on polyhedral loop transformations and code generation, that should allow developers to specify custom convolutional operators and still benefit from advanced optimisations, achieving performance comparable to hand-tuned implementations provided by vendor libraries.

Finally, also recurrent units can be accelerated by carefully applying concurrency both within the layer itself, or among consecutive layers: a comprehensive review of these techniques has been published by Appleyard et al. [85].

3.2.2 Model parallelism

Model parallelism involves the partition of the model computation graph among different workers, that train different parts of the same model instance. It has been proved to be an efficient way to improve the performance of DNN training [86, 87], in particular, layer pipelining [88–90] is a very effective strategy to allow training of models that normally would not fit the memory of the training device.

As the focus of this work is on data parallel techniques, a full investigation of this topic is out of scope, nevertheless it can be argued that model parallelism capacity to scale beyond the single machine is limited

by the higher frequency of communications required with respect to data parallelism, especially if the distributed workers are loosely coupled (i.e. cloud instances without dedicated interconnection, edge devices). Moreover, model parallelism can be used transparently within a distributed data parallel set-up to improve node-level performance, hence it represents an orthogonal direction of improvement with respect to data parallelism. In fact, while this aspect is not explored in this work, it can be quickly added to the data parallel strategies discussed later as a further layer of concurrency without impacting the following discussion.

3.3 Data parallelism

Despite the widespread use of the term *data parallel* in deep learning literature, the definition that is used in this context is not the same that is well known for parallel and distributed computing [91, p. 170]: by the classical definition, in fact, also most of the techniques mentioned in the previous sections involve some kind of shared memory data parallelism. In the deep learning context, by data parallelism it is indicated any parallel algorithm that works by processing different samples or subsets of the training dataset at the same time.

3.3.1 Mini-batches and GPUs

The previous section introduced mini-batch parallelism as the most elementary way to achieve parallelism in DNN training [92]: in particular it is the simplest way to achieve what is defined in the context of deep learning as *data parallelism*. In the case of mini-batches, n_b samples are processed concurrently, possibly on a many-core device that may achieve tremendous speedup by means of optimised GEMMs, such as GPUs [42]. Figure 3.2 shows the performance improvement that can be achieved using the GPU on a typical image classification workload.

The range of batch sizes that are feasible to be used on a single device falls typically well within the green area of figure 2.5: the upper bound of batch size is usually imposed by the memory of the device itself. In this context the theoretical applicability of larger batches is less of a concern, on the other hand, Batch Normalisation (BN) [95] presents a potential performance bottleneck. Batch normalisation [95] computes statistics along the mini-batch dimension, introducing data dependencies between different samples among the same mini-batch, such that a full synchronisation is required at each invocation. Several solutions has been proposed to work around this issue [**YouScalingSGDBatcs sh2017**, 50, 96, 97], yet they all require to relax some constraint on the formulation of BN.

Parallelism at mini-batch level proved to be effective at node-level when implemented on GPUs, multi-core CPUs or other dedicated hardware (e.g. Google TPUs [98]); still, its performance portability to distributed memory architectures is subject to a suitable problem granularity and must be investigated separately.

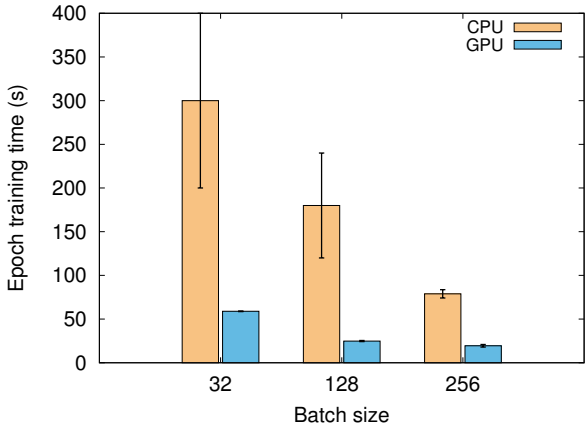


FIGURE 3.2: Time to train 1 epoch of ResNet18-v2 [93] on the CIFAR10 dataset [94] with MxNet, MKL-DNN [73] and cuDNN [44]. Large variance of CPU performance is due to the progressive worsening of compute time as the epochs go on; the nature of this effect is still unclear. CPU: 2x Xeon E5-2680 v3, 12 core 2.5Ghz. GPU: Xeon E5-2650 v2, 8 core 2.60GHz + Nvidia Tesla K40.

3.3.2 Distributed training

Distributed data parallel training entails a number of workers training (possibly identical) model replicas on different partitions of the dataset. Literature classifies the several different approaches to distributed training on two main axes: *model consistency* and *parameter distribution and communication*. The first axis reflect the property of model replicas to have equal values for the weights $\mathbf{w}$ at any given instant, the second involves the different communication approaches (e.g. topology, compression) used to transfer gradients and weights between different model replicas. Sometimes an additional axis is also considered [1], regarding a specific category of distributed training approaches that this work places on the far end of the model consistency spectrum, without the need of a further axis to discriminate.

This section will explore the spectrum along the model consistency axis, while spanning the other one when necessary.

General remarks

Section 2.1.2 stated that naïve SGD is intrinsically sequential. Figure 2.3 and equation (2.7) show how the gradient value depend on the present $\mathbf{w}(t)$ configuration and how its application through back-propagation produces a new configuration $\mathbf{w}(t+1)$: the new weights represent a data dependency for the feed-forward step for sample x_{i+1}, that must come strictly after the back-propagation, otherwise the gradient would be calculated based on outdated (*stale*) weights. Gradient *staleness* is a property of a gradient that is computed based on a given set of weights (e.g. $w(t)$), but is applied to the model at a later stage (i.e. when the model is in configuration $w(t+k)$).

In principle this data dependency between subsequent SGD steps prevents any kind of input sample-based parallelism while, in fact, this is true strictly for on-line SGD: the concept itself of batch (or mini-batch) gradient

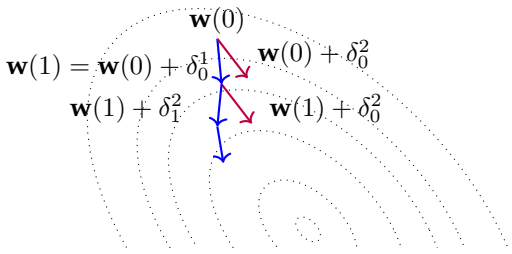


FIGURE 3.3: Gradient descent in $\mathbf{w}$ space. $\delta_t^j = -\eta \nabla L(\mathbf{w}(t), \mathbf{x_j})$ represents the gradient calculated on the weights updated up to step t, based on sample (or mini-batch) $\mathbf{x_j}$. Therefore, the red update based on δ_0^2 is outdated with respect to $\mathbf{w}(1)$, but its impact is not necessarily detrimental to the training. The target function is $L : \mathbb{R}^m \rightarrow \mathbb{R}$.

descent involves parallelism. The gradients related to all the samples in the (mini-)batch are computed based on the same value of $\mathbf{w}$ and, possibly, at the same time. It is worth noting that the data dependency depicted in figure 2.3, is introduced by on-line training algorithm and not by the problem itself, hence there is room to relax this dependency, either with mini-batches or with more sophisticated techniques that relax the dependencies *between* mini-batches. Figure 3.3 exemplifies a possible behaviour of SGD on a loss surface: it is not necessarily true that using always the most recent gradient leads to the best training accuracy, even the red update could end up to good loss minimum. In this sense is important to remember that the loss surface of DNNs is highly non-linear and difficult to describe globally [31, 99]: a certain amount of noise and randomness associated to the gradient descent can be beneficial to the training outcome in terms of generalisation. The next subsections will describe how this behaviour can be exploited to introduce some degree of parallelism into the training process.

Wheter stale gradients are in general beneficial to the convergence of the training is an open reasearch problem, no clear answer is avaialble at this time. For instance, asynchronous approaches benefit from strategies that try to keep the amount of gradient staleness under control [100], while on the other hand, the concept of staleness can be linked to the idea of *momentum* applied to SGD, which usually improves training convergence [101]. A reasonable conclusion appears to be that some degree of gradient staleness is acceptable, if not beneficial to training, while too much staleness is detrimental. Still, where to put the threshold on staleness is not clear and possibly it is problem (and approach) dependent.

As a final side note, the usual formulation of SGD involves sampling data *with replacement*. On the other hand, most of the mainstream techniques discussed below partition the data among workers *without replacement*, possibly applying reshuffling of data between epochs [50].

Synchronous

Synchronous SGD training represents the extension to the distributed domain of the mini-batch concept: assuming the local mini-batch update as an atomic work unit on the single node, the synchronous training involves

multiple model replicas running independent backpropagation steps on different mini-batches, then adding together all the obtained gradients. The workers the proceed with mini-batches in a *lock-step* way. Given K different workers, each one training its own model replica with a mini-batch size of n_b, it is trivial to prove the following

Theorem 1. *Considering the mini-batch gradient averaging set-up (cf. eq. 2.8) and given uniform learning rate, the set of weights $\mathbf{w}(t+1)$ obtained by a synchronous aggregation of K workers with batch size of n_b, is the same as the one obtained by a single node with batch size of $K \cdot n_b$ if the updates are also divided by the number of workers K.*

Proof

Given the input dataset $X = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, the proof is obtained by writing the sum of all the individual workers' updates as in equation 2.8.

$$
\begin{aligned}
\mathbf{w}(t+1) &= \mathbf{w}(t) \quad - \frac{\eta}{K} \sum_{i=0}^{K-1} \delta L(\mathbf{w}(t), X_{(i \cdot n_b, i \cdot n_b + n_b - 1)}) \\
&= \mathbf{w}(t) \quad - \frac{\eta}{K} \left[\frac{1}{n_b} \sum_{j=0}^{n_b-1} \nabla L(\mathbf{w}, \mathbf{x_j}) \right. \\
&\qquad\qquad + \frac{1}{n_b} \sum_{j=n_b}^{2n_b-1} \nabla L(\mathbf{w}, \mathbf{x_j}) + \dots \\
&\qquad\qquad \left. + \frac{1}{n_b} \sum_{j=(K-1)n_b}^{(K-1)n_b-1} \nabla L(\mathbf{w}, \mathbf{x_j}) \right] \\
&= \mathbf{w}(t) \quad - \frac{\eta}{n_b K} \sum_{i=0}^{K \cdot n_b - 1} \nabla L(\mathbf{w}, \mathbf{x_i}) \\
&= \mathbf{w}(t) \quad - \eta \, \delta L(\mathbf{w}(t), X_{(0, K \cdot n_b - 1)})
\end{aligned}
$$

□

Recent works [50–54] have demonstrated that it is possible to push the mini-batch size further than previously expected without affecting the model convergence. These works leverage distributed GPU architectures in order to allocate and efficiently compute such large mini-batches, while relying on an all-to-all communication pattern [91, p. 121] to collectively average the gradients (*all-reduce* operation, using Message Passing Interface (MPI) nomenclature [102]). Figure 3.4 depicts the aforementioned approach, which will be also referred to as *large mini-batch*. While more common in the asynchronous case, it is also possible to use a centralised Parameter Server (PS) to aggregate gradients, which are then broadcast back by the PS to all the workers to start the next batch.

Section 2.1.2 discussed widely about the opportunity to push the batch size so far and, apart from the theoretical issues of convergence and batch normalisation, it must be taken into account that the large mini-batch approach may suffer from a significant communication bottleneck due to the

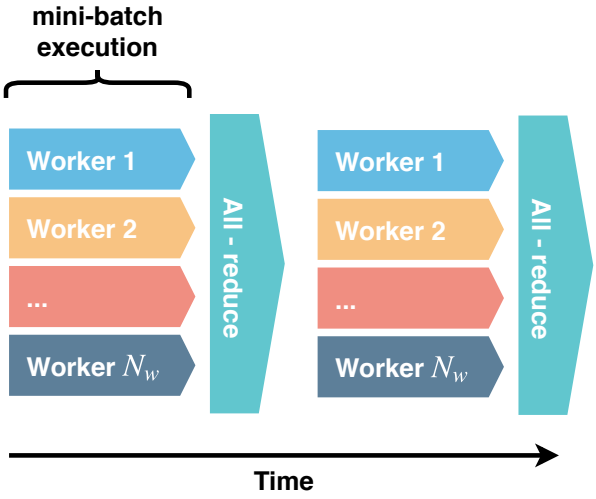


FIGURE 3.4: Synchronous SGD, a.k.a. large mini-batch approach.

all-reduce operation required at every step. In this sense several techniques to reduce the impact of communications will be presented later in this chapter.

Note that theorem 1 holds for naïve SGD holds also for momentum-based algorithms, as also stated by Goyal et al.[50] and formalised by the following theorem.

Theorem 2. *Given SGD with a momentum term $\mu < 1$ defined recursively as*

$$\mathbf{w}(t+1) = \mathbf{w}(t) - v_t$$

where

$$v_t \stackrel{\text{def}}{=} \mu\, v_{t-1} + \eta\, \delta L(\mathbf{w}(t), X)$$

The weight update for a single worker with batch size $K \cdot n_b$ is equivalent to the synchronous weight update of K workers with batch size n_b, where the gradients of each worker are divided by the number of workers.
This entails that each worker should communicate the update as

$$\frac{\eta}{K}\, \delta L(\mathbf{w}(t), X)$$

in order to preserve the correctness with respect to the sequential version.

This Follows from theorem 1: as everyone's v_{t-1} term is the consistent, due to the synchronous behaviour, the only critical requirement is to take into account the number of workers when averaging gradients, as happens in the plain SGD case.

Asynchronous

The success of momentum [46] as a method to accelerate the training convergence, show that the information of previous gradients is definitely relevant even at the current iteration. Although the idea of trading gradient staleness for computational efficiency has been at first exploited for what

is defined *asynchronous parallel training*. As the name suggests, this strategy involves multiple workers performing their own gradient descent for a certain amount of iterations, while their findings (i.e. new weights, accumulated gradients) are shared with other workers without a global synchronisation at the mini-batch level. Figure 3.5 depicts the typical set-up with a centralised parameter server and multiple workers that asynchronously send gradients to the PS and receive back updated weights.

This kind of approaches sits midway through the model consistency axis, and it is possible to discriminate between *centralized* and *de-centralized* implementations along the parameter distribution axis. In practice, the latter categorisation regards the usage of a centralized *parameter server* to store a "master copy" of the model weights or, otherwise, to coordinate the exchange of gradients without a central authority. Early notable implementations of asynchronous parallel gradient descent are HOGWILD! [103] and its deep learning-focused derivatives like Downpour SGD [86, 104]; followed by some other significant works [100, 105–110]. Apart from the *DistBelief* [86] and *Project Adam* [104] papers, that presented results previously not achievable and moved deep learning resolutely into the HPC domain, most of other works, while reporting solid scalability and timing results, were not able to provide a significant legacy. In fact, the dominating entries from DAWNBench [111, 112], at the time of writing, are still relatively small-scale, synchronous implementations.

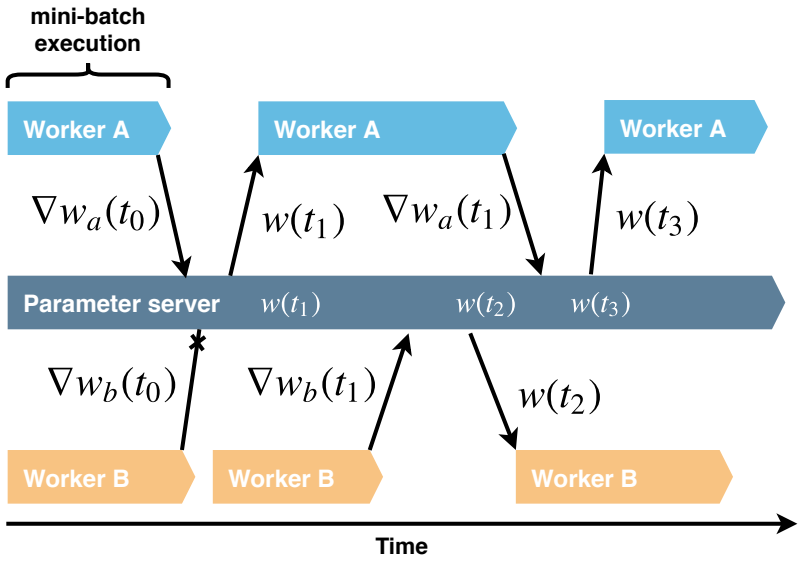


FIGURE 3.5: Asynchronous SGD with parameter server. The second batch of worker B is based on *stale* gradients, as its first update failed due to the busy PS.

While this review is far from being conclusive, it suggests some limitations that arguably prevented widespread adoption of asynchronous techniques. For instance, the added complexity of a parameter server or a sophisticated decentralized protocol might be perceived as not necessary since synchronous, all-reduce-based, parallelism has mostly satisfied the quest for deep learning scalability up to this point. Moreover, most of these works present asynchronous implementations of naïve SGD, while the state of the art is moving to more sophisticated algorithms like ADAM [47].

Some effort in this directions exists [113], as well as a prominent theoretical work [101] that links gradient staleness to momentum; still, the literature is lacking a comprehensive analysis of the asynchronous behaviour of algorithm beyond SGD. Finally, results are usually reported as a collection of experiments on specific use cases, lacking a generalisation effort that might help to understand the validity of the methodology. In this sense a relevant analysis has been performed by Lian et. al [114]: the theoretical discussion of the convergence rate for an asynchronous, decentralized algorithm represent a good starting point for a performance analysis. However, it can be argued that the real life behaviour is affected by a large number of variables (e.g. weight update protocol, communication latencies, etc.) that prevent this model to fully describe the performance of a given implementation. These limitations, along with the lack of details on the code and framework used for experiments, call for a research effort that aims to fill the gap between sparse experimentation and mathematical modelling of convergence rates.

Overall, the choice between a synchronous and an asynchronous approach is not trivial. The practitioner would probably choose the former given the large number of production-level implementations readily available, but the researcher should not stop there. It is a hard task to characterise the two approaches in terms of their relative performance, an intuitive take would be to assume that synchronous strategies provide more accuracy, while asynchronous ones are directed towards efficiency, but there is no clear consensus. Given the literature reviewed, it is reasonable to conclude that synchronous methods are more efficient both in terms of raw speed and convergence than asynchronous ones when considering their preferred area of application (i.e. small-to-medium scale systems, tightly coupled, with hyper-parametrization to accommodate large batches), while some other scenarios are more favourable to asynchronous methods (i.e. very large scale, less tightly coupled systems with need for fault tolerance [86, 104]), despite the heavier fine-tuning needed for hyper parameters. Chapter 6 will mostly deal with the former scenario.

Other approaches

At the most inconsistent end of the model consistency spectrum, *Model averaging* [115, 116], like Elastic Averaging SGD (EASGD) [117], allow concurrent model replicas to perform training independently up to a certain point (i.e. from several mini-batches to multiple epochs), then the weights are averaged among the different replicas. *Ensemble learning* [118, 119] is an *embarrassingly parallel* approach that performs the whole training on different model instances defined by different hyper-parameters, then averages the predictions among them. As said before with respect to model parallelism, ensemble learning represents an orthogonal direction of improvement with respect to parallel gradient descent, hence it will not be discussed hereafter. On the other hand, while it is out of scope to formally draw the connection, model averaging is strictly related to the techniques presented in the previous paragraph, where instead of exchanging gradients, there is an exchange of weights.

Figure 3.6 resumes the methods introduced in this section along the axis of consistency and communications.

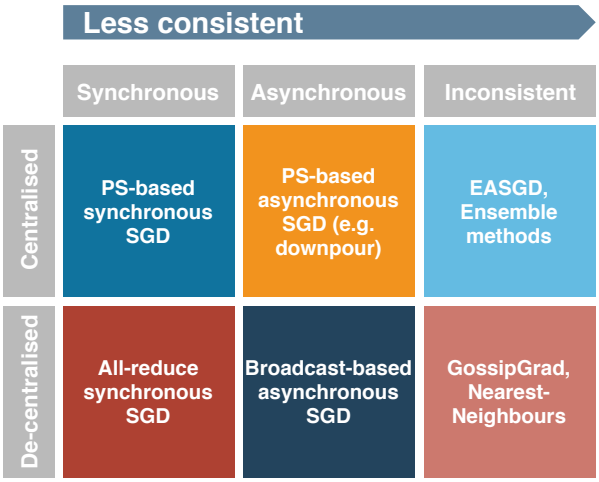


FIGURE 3.6: Taxonomy of different approaches to distributed deep learning.

Further parallelism issues

As already said in this chapter, mini-batch parallelism tends to be performed within a single node, either in shared memory or distributed among multiple GPU. The computing horsepower provided by GPUs or other dedicated hardware is usually enough for most applications, still, there is the need to push the capability to train DNNs effectively beyond the single node. While large mini-batches and asynchronous techniques can be applied also within a single machine when the problem is small enough, representing an interesting research domain itself, they are born to be distributed; this raises a number of issues related to the communication of gradient updates.

The size of the gradient set ($\Delta L(\mathbf{w}, X_{(i,i+n_b-1)})$) for a state of the art DNN easily reaches a few hundred MB [120]. This represents a serious bottleneck for distributed implementations and two main techniques are used to reduce the size of the gradient set to be transmitted: *quantization* and *sparsification*. The former intends to reduce the precision of the gradient representation in order to reduce its overall size and it is demonstrated that this technique works up to 1-bit representation [106, 121]; the latter exploits the sparsity that naturally occurs in DNN gradients, where most of the components are zero or almost zero. In this way the array gradient component can be represented as sparse and compressed with well-known techniques [106]. A more recent work [120] also includes momentum in the discussion and presents interesting results. Also in this case, apart from the 1-bit quantization provided by Microsoft CNTK [65], the frameworks used are not mentioned nor the code is made available.

More methodologies can be exploited to enhance the performance of distributed training, like the optimisation of the all-reduce pattern required by the large mini-batch training or the overlapping of computation and communication during training. Even if these techniques fall more in the domain of the implementation details than in the field of parallel training

algorithms, they play a non-negligible role in the overall training performance.

3.4 State of practice

The previous section stated how DAWNBench [111, 112] entries are mostly synchronous implementations with a few nodes and a high density of GPUs per node. This is mostly true also for the recently announced MLPerf benchmark [122]: an industry wide effort to set standard metrics for machine learning tasks performance.

A brilliant example of the trend of performance-oriented deep learning machines is the DGX/HGX series of machines by Nvidia [123], or the Big Basin architecture from Facebook [124]: single nodes with an extreme density of GPUs and high-performance interconnection between graphic processors that allow to perform communication collectives among them without moving data to the host memory. This kind of deployments are usually enough to handle most industrial training workloads, on the onther hand, in the research environment it has been observed a clear convergence between "conventional" HPC and deep learning-oriented infrastructure. For instance, the world's most powerful HPC machine at the time of writing [125] is clearly designed with DL workloads in mind [126], given the 6 Nvidia tesla V100 GPU per node. This kind of extreme scale machines with thousands of nodes, definitely require a novel take on the distributed training strategies, as naïve synchronous SGD might not be applicable at such scale.

Finally, it is very hard to find any recent trace of asynchronous implementations from either software tools documentation, benchmarks, industrial deployments, and research papers. This is even more true for model averaging techniques, that seem to be completely left behind by the community.

3.4.1 Implementations

On the software side, all the major frameworks among those listed in section 2.2 (Tensorflow, PyTorch, MxNet and CNTK) implement a synchronous version of the SGD either based on all-reduce operationsor on a parameter server. Some also implement an asynchronous training mechanism, like MxNet, however it seems more like an afterthought solution, and the author was not able to get any of the examples provided in the documentation to converge, even with only 2 workers. External tools that allow the frameworks to run in a distributed fashion are also available, like Horovod [127], which is it compatible with Tensorflow and MxNet. While interesting, as compatible with multiple frameworks, Horovod is only able to run synchronous SGD, thus is limited by the suitability of large mini-batches.Below there is a review of what is implemented by the major frameworks.

TensorFlow

Originally based on and external Spark runtime for distributed training, TensorFlow now implements its own built-in distributed training runtime based on gRPC [128] for communication. It provides both synchronous and

parameter-server based strategies. At a lower level, it also allows to partition the computational graph among different executors to achieve model parallelism. TensorFlow provides proper strategies to leverage its cloud-based TPU clusters.

PyTorch

Before version 1.0, PyTorch used to provide a minimal approach to distributed training, based on point-to-point and collective communications modelled on those offered by MPI. At the time of writing it is also offered a distributed data parallel training module, that automatically performs synchronous SGD on multiple nodes or GPUs. Communication is either based on plain TCP sockets, MPI or Gloo [129], Facebook's collective communication library based on Nvidia GPUDirect [130] for optimised GPU-to-GPU communications.

MxNet

MxNet built-in distributed runtime is somehow mature and provides dedicated launcher scripts for many cluster configurations and scheduling systems (e.g. ssh, MPI, Slurm and kubernetes-based clusters) and it is fully based on a PS approach in the shape of a distributed *key-value* store. Despite being based on a parameter server, it implements both synchronous and asynchronous strategies and the communications are based on ZeroMQ [131]. There is also support for gradient compression using quantisation. MxNet also advertises built-in support for the Horovod library.

Horovod

The main feature of Horovod is that it provides interfaces for all the previous frameworks, allowing the developers to leverage a high-performance, distributed implementation regardless of their choice of DL framework. It is based on MPI and Nvidia Collective Communications Library (NCCL) [132] for high-performance all-reduce operations, however it only implements synchronous SGD.

3.4.2 Performance metrics

Early efforts of parallel and distributed training, starting from naïve single node, multi-GPU implementations, used to state the number of images processed per second as the main performance metric. This may have been sufficient as the global mini-batch size was still largely in the feasible range so that accuracy was not affected, however, the scenario is quite different now that the batch sizes allowed by a single device are much larger, let alone the global batch size reachable by a few nodes with synchronous SGD. Moreover, industrial players are used to release thunderous claims about their machines' capability to perform incredible amounts of Operations per Second (OPS), when considering lower than 64-bit precisions. There is the need to define performance metrics that actually reflect the capability to train a model faster by going distributed.

Hoefler presented an interesting point of view [133] about the possible pitfalls of performance evaluation of parallel deep learning; his remarks

provide some useful inputs to this investigation. In particular, the main metric that should be considered in any deep learning performance discussion is the *Time-To-Accuracy (TTA)*, namely the wall-clock time needed to reach a certain, pre-defined, *test* accuracy. The interesting outcome of learning is indeed a trained model able to make useful predictions, hence the relevant time to be considered is the one needed to achieve such model. This kind of measure is also commonly referred to as end-to-end training time. This is relevant as often times it is only reported the time to run one epoch of training, which is hardly meaningful if the training does not converge quickly. In chapter 5 the time per epoch will be also presented, but it will be carefully put into the right context, without using it as a replacement for TTA.

The discussion about performance measurement of deep learning models should also try to discern between *strong* and *weak scaling*: in fact there are two different points of view that can be adopted, the first considers end-to-end training as a single task and uses TTA as its only metric to evaluate performance. In this case any parallel algorithm that can train a network in less time than what required by the state of the art approach scales strongly. Under this assumption, synchronous, very large mini-batch approaches that can train on the ImageNet dataset [134] in 1 hour and less [50] with respect to single node training are scaling strongly. However, at a closer look, these methods are not accelerating the inner mechanisms of training, but they are only exploiting a grater scalable parallelism in the form of large input data (larger mini-batch), which matches exactly the definition of weak scaling. It is the author's opinion that those two views are complimentary: at the end of the day, data scientist are interested in faster training, whatever the methodology to achieve it, hence the former view is fine to evaluate the performance of training. Nevertheless, improvements in this field can not only come from large scale distributed solutions, hence there is also the need of a concurrent effort to improve the performance at lower level (i.e. operators, layers and networks) that is both on software side and on (specialised) hardware side.

The most interesting point of Hoefler's discussion, however, is the fact that most of the time, despite presenting a good TTA, it is foregone the time needed to tune hyper-parameters in order to achieve reasonable convergence for the training. Hyper-parameter tuning is in fact a very time consuming process: any technique that can achieve scalability without requiring very careful fine-tuning can bring a great contribution to the field. In fact this will be the main advantage of the strategy proposed in the next chapter.

3.4.3 Summary

In this work it has been stated multiple times that most of the approaches presented in previous works are in fact orthogonal, hence can be applied together to achieve better performance, if the amount and the architecture of computational resources allows. This allowed the author to focus on a specific class of parallel approaches to deep learning, without sacrificing the potential advantages carried by other classes. This class is identified in figure 3.1 by the distributed, data-parallel branch. The next chapter will

discuss the limitations of current approaches and proposes a new take on this class of algorithms.

Chapter 4

Nearest Neighbours Training

This chapter will leverage on the knowledge summarised in the previous ones to review the limitations of current approaches and to propose a different take on distributed deep learning, highlighting the expected improvements over mainstream solutions.

4.1 Limitations of current techniques

4.1.1 Synchronous methods

Section 2.1.2 discussed in depth the impact of mini-batch size on the expected convergence of training. Based on that discussion and, above all, on equation 2.9, the following subtleties are required when dealing with large mini-batches:

- careful and problem-specific optimisation of the learning rate;
- smaller-batch warmup phase

and yet the convergence of larger batches is typically slower that SGD with smaller ones. Nevertheless, large batches allow, up to a reasonable amount of nodes, to scale easily and almost linearly, hence the success that brought this methodology into all the major frameworks.

The heavy fine tuning required, though, can be a major bottleneck when developing new models, or possibly training on datasets that are not the well-curated ones used in literature: in fact large infrastructures might not be readily available, hence there is first an expensive search for optimal hyper parameters during the research phase, then an additional optimisation is required for large infrastructures without guarantees of success. Also, while batch size can be pushed quite far, this tuning becomes harder and harder as n_b grows, causing a granularity problem: large mini-batch parallelism *requires* the global mini-batch size to grow in order to (weakly) scale, otherwise is not effective. A practical, even if not theoretical, limit on the batch size, also limits the applicability of this approach.

Moreover, synchronous methods require either frequent all-to-all communications or a parameter server which is expected to be flooded with updates at the end of every mini-batch. This fact can possibly hinder the scalability of this approach when considering large deployments, where the amount of communication can become overwhelming also for high-end HPC machines. Given the aforementioned issues, it is reasonable to state that for all practical purposes, the mini-batch size has an upper bound in the range of a few thousands samples.

The communication scheme of synchronous SGD is also detrimental to the portability of such strategies outside HPC infrastructures or cloud environments. In particular, the envisioned perspective of *edge computing* entails a large number of agents acquiring and processing data locally, while possibly creating machine learning models on the without moving the data to a centralised warehouse. Whether the need of not moving the data is dictated by technical or privacy-related reasons, the low-bandwidth, loosely coupled environment where the devices operate is not suitable for them to train cooperatively in a synchronous fashion.

4.1.2 Asynchronous approaches and model consistency

Assuming that using very large mini-batches is not generally suitable for any application, it would be natural to investigate asynchronous methods, that in theory sport good convergence properties thanks to gradient staleness mimicking the behaviour of momentum [101], that is itself quite beneficial to the accuracy. Moreover, they require less frequent communications [135], so they are definitely a good candidate to achieve scalability at scale without affecting the convergence of training.

On the other hand, there are a two major issues with this category of parallel SGD: a pratical one and a theoretical one. The former regards the amount of *ad-hoc* solutions needed to achieve convergence at scale, which is exemplified by the DistBelief work [86]: achieving convergence with asynchronous methods without prior and deep knowledge of the problem is even more difficult than with large mini-batches. Then, it is possible to show that asynchronous methods are, at least in principle, only an asymptotic approximation of synchronous SGD, also sharing the same weaknesses.

To introduce this idea, it can be shown that, despite usually being treated as different approaches, all the synchronous and asynchronous techniques are in fact all *eventually consistent*. To proceed it is useful to introduce the following

Property 1 (Communication completeness)**.** *Let*

$$\mathbf{p} = \{p_1, \dots, p_r\}$$

be the set of processors participating in a communication. Such communication is said to be complete if every message sent by a processor p_i is received, with a bounded or unbounded delay, by all other processors $\{p_j : j \neq i\}$.

A statement can be formulated from here that, while being quite naïve, is still important to understand the behaviour of model replicas

Proposition 1 (Potential consistency)**.** *Assuming a mini-batch SGD without momentum in a distributed setting, if property 1 holds for gradient communication all model replicas can potentially be consistent.*

from which derives

Proposition 2 (Eventual consistency)**.** *Assuming proposition 1, if all the gradient updates generated by all the processors are actually applied to all the recipient model replicas, regardless of the delay, these will be eventually consistent.*

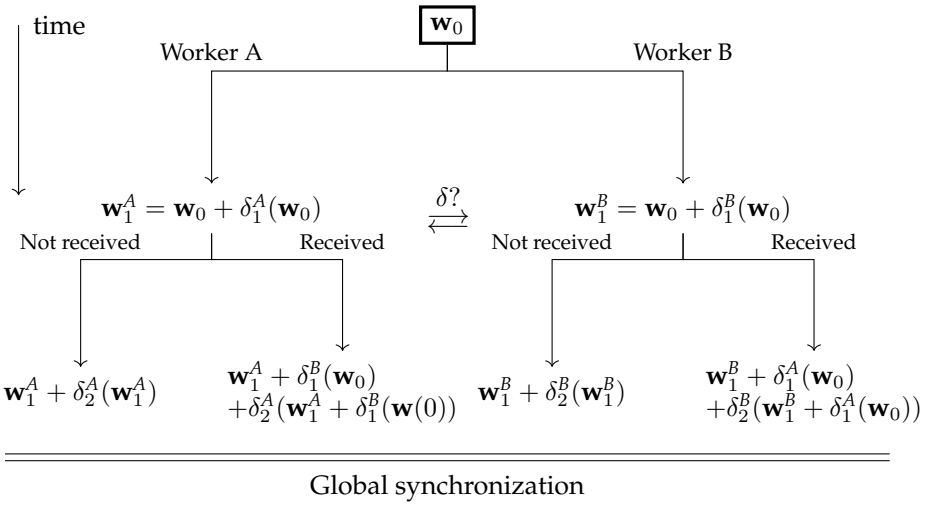


FIGURE 4.1: Diagram of weights updates between two workers. $\mathbf{w}_0$ is the common starting configuration. Assuming that all the updates that are not immediatly applied are queued in a per-worker buffer, *commutativity and associativity of vector sum guarantee that A and B will always be consistent once the buffers are emptied.*

Figure 4.1 presents the diagram of subsequent gradient updates for 2 workers: using commutativity and associativity of the vector sum that represent the gradient update, it is trivial to prove that, if an event triggers the application of all the pending updates (e.g. a global synchronisation), whatever is the state of both workers before the event, their state will be consistent afterwards. Of course proposition 2 does not hold if, for instance, updates not yet received are simply dropped, instead of accumulated. Moreover, it must be highlighted that having consistent model replicas does not mean that the result is the same as the sequential implementation, but only that all the model replica will agree on the value of $\mathbf{w}$ at a certain time in the future. In this sense workers are *eventually consistent*, as most of the strategies proposed either accumulate all the updates in a parameter server or require a synchronisation at each epoch [106] (or at fixed time frames) or both. This leads us to the following

Proposition 3. *Given property 1, there is no need to distinguish between centralized and de-centralized set-ups, as the eventual consistency is granted.*

In this view the centralized parameter server is only a way to simplify the implementation of asynchronous SGD as well as inducing artificially some staleness (along with the obvious communication bottleneck), that can be beneficial to the training, but the same approach can be replicated with a decentralised approach with broadcasts [106].

The relevance of this discussion becomes clear as the goal of this work is to exploit more parallelism without resorting to large mini-batchs, but *if workers in figure 4.1 always go through the received branch, the outcome is exactly equal to the large mini-batch strategy*. Less trivially, it is possible to figure that this is exactly what happens in an homogeneous, de-centralized set-up, where the load is perfectly balanced and updates are broadcast by each worker to all the others [106], making an asynchronous solution not different from a synchronous one. Of course it can be argued that not enforcing

explicit synchronisation can benefit scalability on very large-scale deployment, however, it would still represent, at most, an approximation of very large mini-batches.

This paragraph presented a slightly different take on the weaknesses of the asynchronous approaches with respect to other works [136, 137], which highlight the drawbacks of the communication bottleneck represented by the PS architecture, or the complexity required by a hierarchical PS, both of which can jeopardise the constant communication complexity granted by the centralised approach. Moreover, the tuning effort needed to achieve convergence is significant, and requires sequential warm-up as happens for some large mini-batch approaches. Essentially, the bottom line of this discussion can be resumed saying that asynchronous SGD shares the same issues of synchronous SGD, without significant improvements in the communication efficiency, while being even harder to tune for convergence in real-life problems.

4.2 Proposed approach

Previous considerations lead to identify a common property of all the presented approaches: a *complete communication graph* as defined in property 1. In fact this is the assumption that guarantees the eventual consistence of all the workers (cf. proposition 2), and that also supports the asymptotic equivalence between synchronous and asynchronous methods. This equivalence is the key to identify potential new strategies: this research effort originally started by investigating how to make asynchronous methods more efficient and staleness-bounded, but soon *the author realised that any effort in that sense would only have led again to large mini-batches*, hence the need to find another, different, approach.

The potential strategies left to be explored are the following:

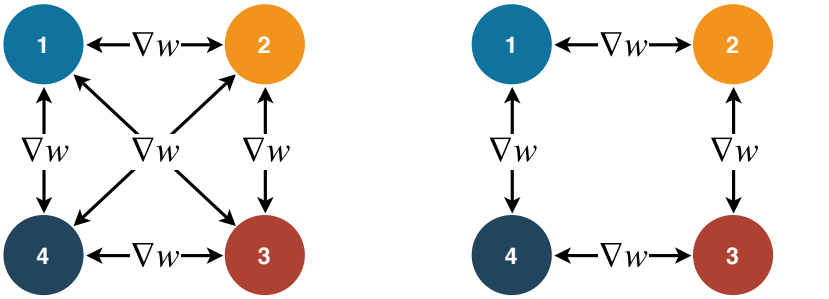
1. at node level
    - by implementing tensor operations in back-propagation even more efficiently;
    - by developing new dedicated hardware that is better suited to handle small mini-batches;
2. at distributed level
    - by improving parallel gradient descent without falling back-on large mini-batches;
    - by developing a different optimisation strategy that exploits parallelism better than gradient descent.

Point 1 is being researched actively [84, 138] and it is clearly out of the scope of this paper. Also the development of algorithms that departs completely from gradient descent is an interesting topic, still this work is focused on improving on parallel gradient descent. The solutions that is being explored in this work instead keeps all the properties of parallel SGD, but drops the main assumption discussed above: *communication completeness*.

A brief revision of the worker consistency spectrum presented before [1] is useful before delving into the details of the envisioned strategy:

1. synchronous communication (large mini-batches)
2. complete communication with bound delay (stale-synchronous [100, 135])
3. complete communication with unbound delay (Downpour SGD [86])
4. partial communication ([4, 139, 140])

where it has been introduced a new class of algorithms that leverage incomplete communication topologies. This class involves a number of worker nodes, each one hosting a model replica which performs training on local data and exchange gradients with a *proper subset* (i.e. neighbours on a given topology) of the workers set in either synchronous or asynchronous way. They key feature of this point is that not all the gradients reach all the workers, not even after an arbitrary delay.



(A) Complete communication graph. (B) Incomplete communication graph.

FIGURE 4.2: Comparison of different communication strategies for training. Figure (a) is typical of synchronous, all-reduce, strategies and broadcast-based asynchronous strategies. Parameter server-based approaches achieve the same result by distributing weights instead of gradients, although with a different topology. In this set-up all the gradients produced in the workers' graph always reach all the other nodes, while in figure (b) the gradients are always relayed by other nodes, hence their effect is mediated by a certain amount of non-linearity the effect of which is not trivially understood.

The previous section stated that the first three points in the spectrum are asymptotically equivalent in terms of convergence, although chapter 6 will show that practice does not always behave as expected. It is true that a centralized set-up with a parameter server forces a degree of asynchrony since gradient updates are queued, still this is more a limitation of the centralized implementation that a property of this strategy, moreover the centralized approach introduces an obvious communication bottleneck. Point 4 would be, instead, a significant departure from large mini-batches, potentially providing beneficial effects to the convergence, while its scalability can be expected to be almost linear in terms of samples processed per unit of time, as it is for most of the asynchronous implementations. Moreover, this approach would significantly benefit in loosely-coupled heterogeneous environments (e.g. *edge*), where the communication is costly and unreliable.

4.2.1 Sparse topology training theory

The previous sections of this chapter came to the conclusion that any effort to improve the efficiency and bound the delay that is typical of asynchronous methods is futile, as it only gets closer to the optimisation path of a large mini-batch, which is already quite efficient at small scale, but it is flawed when trying to move to the HPC domain or to loosely coupled, edge settings. Moreover, large-scale asynchronous deployments require extreme and problem-specific careful to achieve convergence. Dropping the completeness of the communication graph, and concurrently dropping the requirement of model consistency, would represent a completely different optimisation dynamics with respect to any approximation of large mini-batches; this section will try to sketch some theoretical foundations and to discuss some previous work. However, as it happens for the large mini-batch approach, the assumptions needed to provide meaningful theoretical results are so tight that the results are hardly applicable to real-life training problems, shifting the focus on experimentation to validate the idea.

It is clear that allowing partial communication definitely gives up on model consistency, even in the long run, and the effect of this concept on the convergence must be better understood. This last issue is also strictly related to the possibility to terminate some workers at any given time without impacting the overall convergence: this matter has been already discussed [86], but only from the point of view of fault tolerance of the training system, not in terms of training accuracy.

Previous efforts

Very few previous works explored this topic: Jin et al. [141], Blot et al. [142] and, during the progress of the present work also Daily et al. [4], all proposed some kind of training based on *gossip* communication [139, 143], where the weights are exchanged only between selected nodes every time. These works takes their origin from model averaging techniques like EASGD [117], hence they communicate weights instead of gradients, as proposed here.

Jin et al. [141] provide a comprehensive convergence analysis based on the same premises used in section 2.1.2, which is definitely theoretically interesting, but hardly applies in this case because it is based on weights exchange instead of gradient exchange. Moreover, it can still be argued that the subtleties of implementation can impact the results too much to make this results directly applicable to predict training performance. Another issue of this work, shared with the one from Blot et al., albeit this last one is more limited in its scope, is that the choice of gossip partners is random, causing potential communication imbalance and irregular diffusion of results. These pitfalls prevented the two approaches to achieve performance and convergence at large scale.

The work from Daily et al. [4], on the other hand, presents an idea which is closer to what envisioned in the next section, with a more structured communication topology. This work presents an interesting proof of convergence for this approach, yet it is based on the concept of exchanging weights instead of gradients, hence it is not useful to describe the current scenario. Moreover, the proof postulates the convergence of all the workers

to the same minimum, but this is not verified by experiments[1]: this observation does not necessarily void the correctness of the proof, which is sound, but only remarks once again how the assumptions made in order to build a theoretical framework are hardly applicable in real problems. Finally, a critical point in this work is given by the mini-batch exchange strategy applied between workers: while effective in the specific set-up, this idea potentially limits the applicability of this approach to scenarios where workers draw from the same data pool (i.e. in case of a shared file system), or at least are allowed to exchange real data; different settings can be envisioned where data are produced locally and may not be exchanged, either because of technical limitations (e.g. limited bandwidth, that should be dedicated to gradients) or because of privacy issues. On the experimental side, while presenting interesting results compared to asynchronous approaches, there is no clear convergence comparison with respect to a sequential implementation, which will be show to be the most challenging comparison.

The present effort hopes to present a more comprehensive theoretical background, as well as a more detailed discussion on the implementation of the experimental software framework. In the author's opinion, clearer experimental results with respect to previous works are also proposed based on what discussed in section 3.4.2.

Incomplete topologies has been also explored theoretically in generic optimisation context [139, 144–146]; a promising direction for a rigorous formalisation, at the time of writing, is to draw a connection between the work done in general-purpose distributed optimisation, and deep learning, similarly to what is done by Tatarenko et al. [146] when highlighting how their formulation corresponds exactly to SGD as described by Robbins and Monro [36].

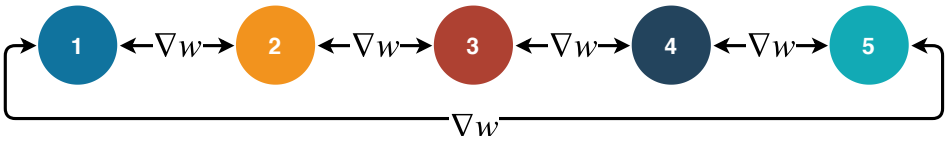
A full fledged formalisation for sparse communication SGD is even more difficult to realise than for synchronous training and it is not expected to be very effective in describing the real-life training performance. In this sense, this paragraph presented no claims about the theoretical performance of this approach, the validation of which will be delegated to experimentation.

4.2.2 Nearest-Neighbours Training

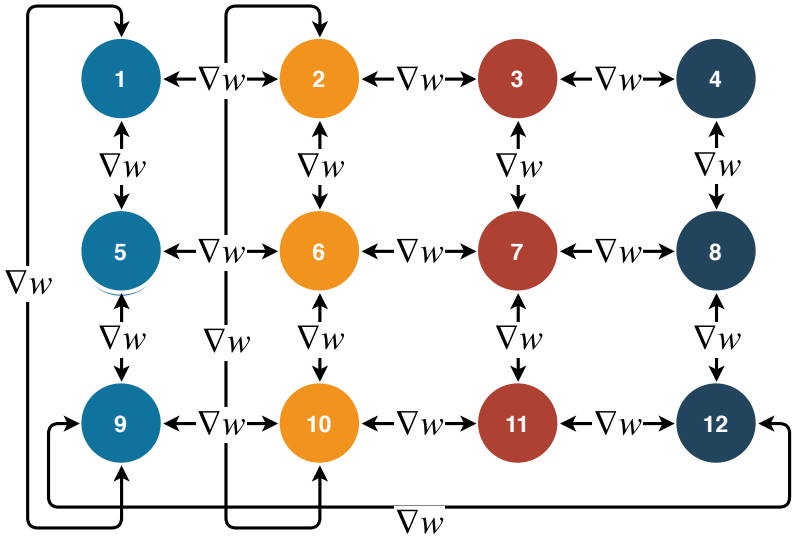
Finally, this section will introduce the core contribution of this research: Nearest-Neighbours Training (NNT) [2, 3]. As anticipated, this idea extends the concepts of synchronous and asynchronous SGD towards the inconsistent side of the consistency spectrum. This background explains the choice of exchanging gradients instead of averaging weights, which represents the main novelty with respect to the techniques discussed in the previous section, that instead appear to come from a model averaging background.

Loosely speaking, exchanging gradients between individual workers is expected to provide comparable scalability and some improvement in convergence with respect to large mini-batches, since it retains some of the "noisiness" of small batches, that is very beneficial in early training

[1] This point has been personally confirmed by the authors.



(A) Ring topology.



(B) 2D-torus topology.

FIGURE 4.3: Topologies for nearest neighbours training.

stages. On the other hand, given a proper topology, this approach is also expected to reduce the average staleness of gradients with respect to parameter server approaches, without the heavy communication cost of broadcast-based asynchronous SGD. This is eventually expected to be less reliant on problem specific tuning such as warm-up strategies and learning rate adjustments: with the final goal of allowing a straightforward transition from sequential training used for research and tuning of the model itself, to distributed, large-scale production deployment.

NNT represents the practical declination of sparse communications SGD: the idea is to arrange workers in a logical k-dimensional grid where gradients are exchanged between geometrically adjacent workers, as the name suggests. Figure 4.3 represents this concept, with boundary effects avoided by adopting a torus-like communication topology where the grid (or the line, in 1D), is closed on itself by connecting the boundaries together. The chosen topology in this case is not a negligible detail: in fact it impacts both the amount of required communication and the distance between workers: namely, the number of "hops" required for the information about gradients to propagate between two arbitrary chosen worker nodes (given by the L^1 distance on a k-dimensional discrete torus). For instance, this distance differ significantly if the same number of workers is arranged on a 1D ring, or on a 2D grid, at the cost of double the communications. As a rule of thumb, a grid topology can be a good trade-off between the amount of communication and the average distance between nodes.

In this set-up, workers perform a local update on their local data, then they broadcast the obtained gradients to their nearest neighbours . Algorithm 2 presents the simplest version of NNT training as seen from a single

Algorithm 2 Pseudocode for an epoch of a single worker in NNT. Gradient formulation is simplified for brevity.

```
1: while local_data_iterator has next mini-batch do
2:     Compute ∇L(w) with local data
3:     w ← w − η∇L(w)_local
4:     while input_channel ≠ empty do        ▷ non-blocking poll on channel
5:         w ← w − η∇L(w)_neighbour
6:     end while
7:     output_channel.send(∇L(w)_local)
8: end while
```

node. It must be taken into account that gradients are not simply propagated, but they are applied locally, and another gradient is propagated that somehow includes the information carried by all the received gradients, in this sense, for instance, the information produced by node 1 in figure 4.3b reaches (with a delay) node 6 via node 2, 5, and also via different other paths starting from nodes 4 and 9 thanks to the torus topology. Of course in this case the impact of the original contribution of node 1 will be diluted by the number of additional contributions received along the path.

With respect to the training dataset, for the purpose of this work it is *randomized and partitioned among workers at the beginning of each epoch*, while in the future will be considered cases where data is created locally and should not be moved from the node where it resides (e.g. for privacy reasons). The global vision of the training process is depicted by figure 4.4, where a

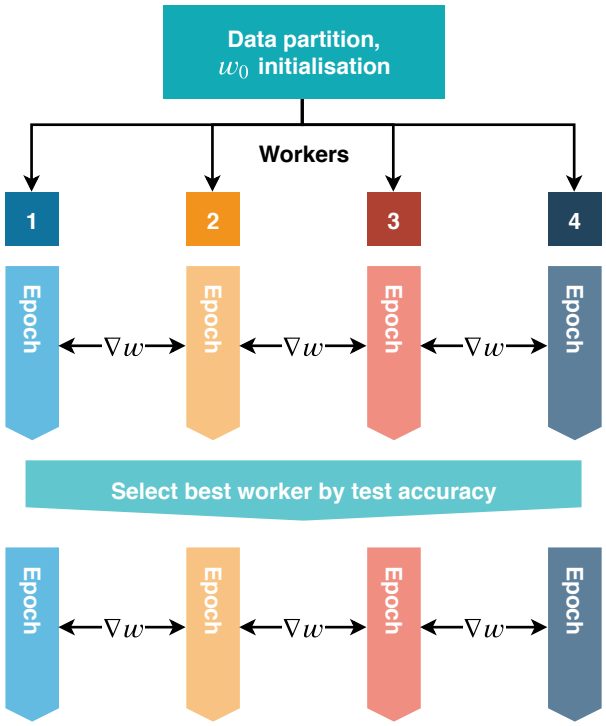


FIGURE 4.4: Typical workflow for NNT. Synchronisations are performed at each epoch to keep bounded the variance between the worker's accuracy. The best worker is the one with the highest test accuracy at the end of the given epoch.

synchronisation step at the end of each epoch is performed in order to reduce the variance between the accuracy of different workers. This kind of synchronisation has limited impact on the performance, as its granularity is very large, and can be made larger by enforcing it every 2 or more epochs, at the cost of larger variance.

The next chapter will detail the implementation of a software framework designed to apply this approach on top of existing deep learning tools, with performance and productivity in mind. Convergence and scalability results will also be presented.

Chapter 5

Flexible Asynchronous Scalable Training Framework

This chapter will illustrate the second last contribution of this research effort: the FAST framework for data parallel training. The choice of developing a software tool from scratch, instead of relying on some existing implementation should be well motivated, as the risk of producing "yet another deep learning framework" with negligible relevance compared to those introduced in chapter 2 is high.

The intention of this work is not to replace or compete with current DL frameworks, but rather to complement them providing a tool tat can ingest existing training code with minimal modification, and run it in a distributed fashion. This approach is similar to the one followed by Horovod [127], which integrates with exitsting frameworks to provide optimised synchronous SGD capabilities. Still, Horovod is constrained to all-reduce, large batch training and does not allow to easily experiment with different strategies. An alternative would have been to leverage existing frameworks' facilities for distributed training, however that would have limited the applicability of this work to code using that specific software, moreover there is no guarantee that the available APIs would have been suitable for the task, although the *send-receive* primitives provided by PyTorch could have been suitable.

The presented approach is designed from scratch to allow the maximum flexibility in terms of both compatible frameworks (limited only by the provided interfaces, not by any design choice) and in terms of allowed training strategies. On the other hand, it is still a research product, and the number of available strategies is, at the time of writing, limited to NNT.

5.1 Design

FAST is a header-only C++ template library with a minimal design: it provides facilities related training, offloading lower level tasks to specific components. Figure 5.1 introduces the main layers that make up its stack: *communication*, *topology* and *training*. Those components will be analysed individually in the next sections, but it is helpful to set the context that led to some key design choices.

This work adopted a framework design instead of a library approach: this choice is reflected in how the training code is handled by FAST, as it is wrapped in a "model logic" object without major modifications and without exposing API calls directly to the developer. In fact, a library approach

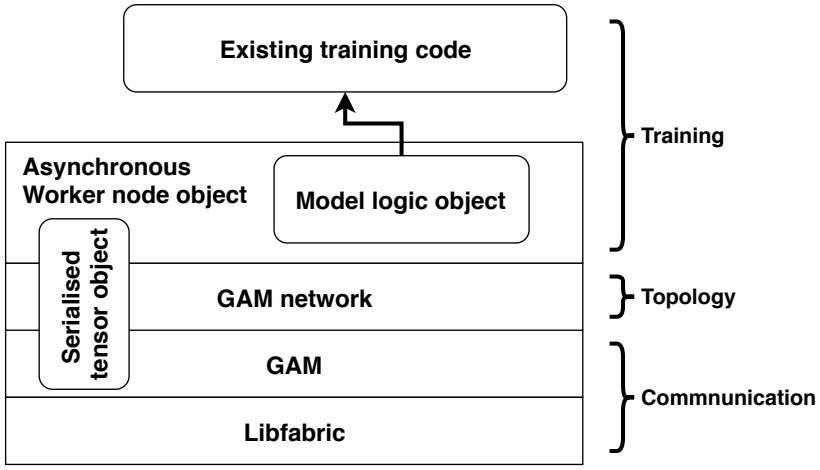


FIGURE 5.1: Simplified architecture of FAST. Communication is based on Libfabric [147] and GAM [148], the SPMD machinery and the NNT topology are instead based on GAM networks [148]. FAST itself implements the code running on each worker node of the process network and that have access to the original training code, wrapped in a proper interface. FAST also provides a dedicated serialised tensor object optimised to be transferred using GAM pointers, and facilities to convert framework-specific tensors to and from this data type.

would have led to a modification of algorithm 1, adding some topology-aware send/receive API calls in the training loop. This approach, while equally good from thedevelper's point of view, would have led to some significant issues in the internal machinery of FAST, which led to the framework approach.

The choice of C++ as the main development language and for the training code has been driven both by the need for a more production-oriented tool, as well as by the availability of better low-level multithreading facilities with respect to Python, that is the dominant language in the deep learning community. Nevertheless, the design of FAST allows enough flexibility to implement specific interfaces for Python-based training.

5.2 Software components

5.2.1 Communication Layer

The communication layer is in charge of moving gradients between worker nodes and it is required to be able to perform collective communications among groups of nodes (neighbours) in a fully asynchronous and non-blocking manner. This goal could have been achieved by using the newly introduced MPI non-blocking collectives [102] and groups, however that approach would have required to implement a significant amount of Single Program Multiple Data (SPMD) machinery in the form of `if` clauses specifying the rank of the process. This machinery would have been possibly visible to the user, affecting its capability to quickly define a topology, and consequently impacting productivity.

Instead, this work leveraged a solution developed within the University of Turin Parallel Programming Group: the Global Asynchronous Memory (GAM) programming model and its C++ implementation, the GAM runtime [148], based on a memory space shared among a set of executors (i.e., a Global Address Space (GAS)). This framework is intended a the lowest level of a stack that aims to overcome the shared-memory vs. message-passing dichotomy by sharing the data on a distributed infrastructure, and solving data races by means of asynchronous message-passing.

GAM

In the GAM model, a memory location is either *public* or *private*. Public memory is accessed in a *single-assignment* fashion, whereas private memory is accessed *exclusively* by the respective *owner*. Therefore, GAM programs are Data Race Free (DRF) by construction. GAM provides message-passing communication *along with* shared-memory primitives, by which executors exchange *capabilities* over memory locations, thus overcoming the traditional dichotomy between shared-memory and message-passing paradigms.

The GAM runtime is a C++ template library that provides smart pointers to private and public memory locations, respectively `private_ptr<T>` and `public_ptr<T>`, which reflect the same syntax and semantics of C++11 Smart Pointers (`unique_ptr<T>` and `shared_ptr<T>` respectively) [149]. At this level GAM runtime still requires an SPMD programming model, however, this is not meant to be exposed to the user. Instead, the right approach to leverage the capabilities of GAM is by means of GAM *networks*, that will be discussed in section 5.2.2.

The key feature of GAM pointers is the fact that they can be moved around between workers in a fully asynchronous, non-blocking manner: passing a GAM pointer does not directly require either workers involved in the communication to copy the referenced memory area, which can be transferred later upon request of the recipient, without affecting the sender.

The back-end used by GAM to perform communications over the given network interconnecting nodes is Libfabric, introduced below.

Libfabric

OFI (OpenFabrics Interfaces) is a framework focused on exporting fabric[1] communication services to applications. Libfabric [150] is a core component of OFI, that defines the user API, enabling a tight semantic link between applications and underlying fabric services. More specifically, libfabric software interfaces have been co-designed with hardware providers (the bottom layer of the stack in figure 5.2) with the goal of giving access to different hardware for HPC users and applications.

A distinguishing feature of libfabric is that it is agnostic with respect to the underlying hardware provider, thus allowing programmers to write applications that can exploit any supported hardware. In this setting, considering figure 5.2, the GAM runtime sits among the "libfabric-enabled middlewares", at the same level as MPI or UPC.

[1] *Fabric* is an industry term to denote a network of interconnected devices in a tightly coupled environment.

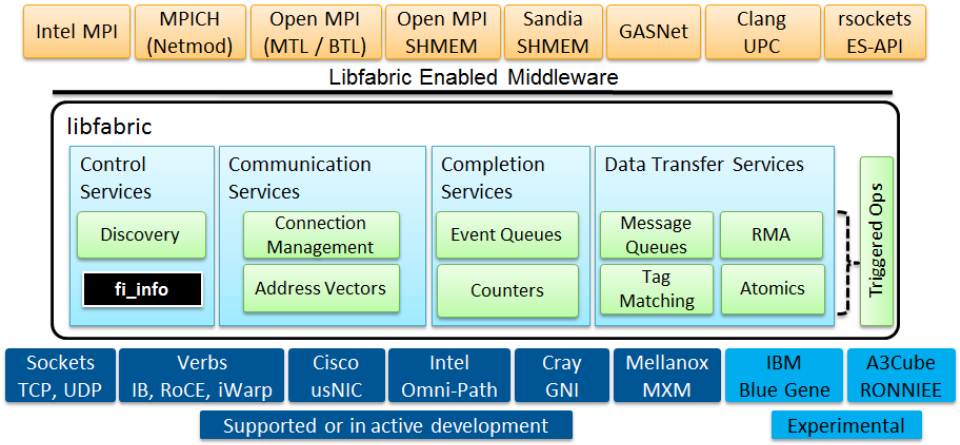


FIGURE 5.2: OFI interfaces overview (from [150]).

Libfabric provides two different APIs for transferring data among network nodes (the "Data Transfer Services" block on the right of figure 5.2): Message Queues and Remote Memory Access (RMA). According to the former API, usually referred as *two-sided* communication, nodes communicate via intermediate queues by means of send and receive primitives, as in any message-passing environment. With the latter API, usually referred as *one-sided* communication, nodes exchange data by accessing memory locations from some shared space. For both APIs, libfabric enforces *asynchronism* by means of user-level notifications (the "Completion Services" block in the middle of figure 5.2), through which the user can query the runtime about the completion of issued data transfers, for instance, to safely reuse memory involved in transfers.

In addition to asynchronous operations, libfabric focuses its support on HPC environments through a number of design choices, described in detail in the "High Performance Network Programming with OFI" guide [150]. GAM topologies are based on connection-less communication ("Address Vectors" within the "Communication Services" block in figure 5.2), that targets large-scale environments by reducing the amount of memory required to maintain large address look-up tables, thus eliminating expensive address resolution.

The adoption of the GAM-Libfabric architecture allows FAST to operate on top of several network infrastructures in a transparent way, without significant performance penalties with respect to a more conventional MPI implementation. Moreover, it allows a natively asynchronous, non-blocking behaviour.

5.2.2 Topology, Processors and Communicators

This layer contains the key components used to materialise the training topology required by NNT without exposing any SPMD machinery to the user and, in this sense, with very limited effort required from the FAST developer.

The formal definition of GAM networks can be found in Drocco's PhD dissertation [148], in this section will be reported a simplified vision of the

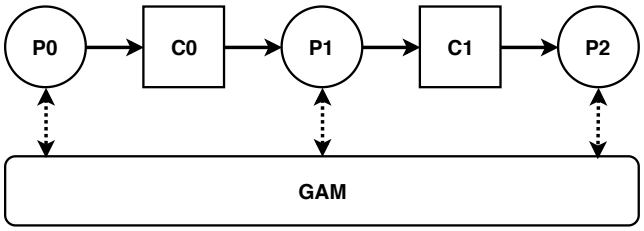


FIGURE 5.3: GAM network with 3 processors and 2 communicators arranged as a pipeline skeleton. The runtime allows the developer to define policies within the communicator, that can have multiple inputs and outputs (e.g. broadcast, round-robin for output channels, costant, round-robin, and other for input channels).

topic that is sufficient for the scope of this work. A GAM network is a bipartite directed graph composed by 2 types of nodes *processors* and *communicators*, with edges connecting processors to communicators or vice-versa. Figure 5.3 represents a basic GAM network arranged as a pipeline skeleton. The communicator is aware of the topology of the network: in fact it knows the identity of input nodes, the identity of output nodes, and the scheduling policy to be applied to inbound and outbound GAM pointers.

The GAM network runtime is designed around generic template programming, in fulfilment of modern C++ principles, in order to avoid any overhead due to runtime polymorphism. GAM nets are based on the same principles as the FastFlow parallel programmin skeleton library [151] (cf. section 5.2.4), for this reason this runtime is referred to as GFastFlow (for GAM-FastFlow).

GFastFlow builds an higher-level programming model on top of GAM, that is centred around stream parallelism. However, literature demonstrates how stream parallelism can be fruitfully exploited to implement other models. For instance, several frameworks for high-level parallel programming, such as OpenMP [152], FastFlow [153], and Flink [154], exploit stream parallelism for implementing data-parallel operations.

5.2.3 GFastFlow

GFastFlow provides three types of processor nodes: *sources*, *filters* and *sinks*, that are defined by their role in terms of communications. Sources can be attached only to an output communicator, on the contrary, sinks only receive data from an input communicator. Filters, instead, are both able to receive and send data. The typical usage of GAM networks is to create dataflow-like Direct Acyclic Graphs (DAGs) where pointers traverse a specific path from a source to a sink. On the other hand, the topology required by NNT is a flat grid of peers, where there is no privileged direction for data, and loops are common. Fortunately, GFastFlow is flexible enough to implement this vision: for this purpose, neither sources nor sinks are needed, but only filters. The C++ API used to define filter nodes is the following:

```
template<typename InComm, typename OutComm,
         typename in_t, typename out_t,
         typename ProcessorLogic>
class Filter;
```

where `InComm`, `OutComm` represent the communicator nodes in the GAM network graph; `in_t` and `out_t` are the GAM pointers of a specific type to be passed to input and output communicators; `ProcessorLogic` in an object with a specific structure of member functions that represents the business logic of the application.

The class representing the business logic must show at least three public members: `svc_init`, `svc` and `svc_finalize`, with proper signatures. The first one is called upon process creation, the second one is called each time a pointer is received by the inbound communicator, while the latter is called upon process termination, which is handled by means of a proper termination token that is propagated along the network with a specific protocol.

Network construction

In order to build a network like the one presented in figure 5.3, the user has to define its `ProcessorLogic` class of each processor in its network (note that some processors can share the same class), then he has to create a proper template specialisation of the filter/source/sink nodes as in listing 5.1. Finally, he may build the network as in listing 5.2.

LISTING 5.1: Specialised objects needed to build a GFast-Flow network behaving as a pipeline (cf. figure 5.3).

```
class SourceLogic {
public:
  gff::token_t svc(gff::OneToOne &c) {
    // Business logic
  }

  void svc_init() {
  }
   void svc_end(gff::OneToOne &c) {
  }
};

typedef gff::Source<gff::OneToOne, //
        gam::private_ptr<int>, //
        SourceLogic> PipeSource;

class FilterLogic {
public:
  gff::token_t svc(gam::private_ptr<int> &in, gff::OneToOne &c) {
    // Business logic
  }

  void svc_init(gff::OneToOne &c) {
  }

  void svc_end(gff::OneToOne &c) {
  }
};

typedef gff::Filter<gff::OneToOne, gff::OneToOne, //
        gam::private_ptr<int>, gam::private_ptr<float>, //
        FilterLogic> PipeFilter;

class SinkLogic {
public:
  void svc(gam::private_ptr<float> &in) {
```

```
37      // Business logic
38    }
39
40    void svc_init() {
41    }
42
43    void svc_end() {
44    }
45  };
46
47  typedef gff::Sink<gff::OneToOne, //
48          gam::private_ptr<float>, //
49          SinkLogic> PipeSink;
```

LISTING 5.2: GFastFlow network creation and execution, based on objects in listing 5.1.

```
1  int main(int argc, char * argv[])
2  {
3    gff::OneToOne comm1, comm2;
4
5    gff::add(PipeSource(comm1));
6    gff::add(PipeFilter(comm1, comm2));
7    gff::add(PipeSink(comm2));
8
9    gff::run();
10
11   return 0;
12 }
```

Listing 5.2 highlights how straightforward is to build a multiprocessing application[2] with GFastFlow: in fact there is no trace of SPMD machinery exposed to the user. In order to run multiple processes it is still needed a proper launcher, as happens with `mpirun` for MPI applications, however that will be discussed in section 5.4, as FAST provides its own specialised launcher.

Communicator bundles

As anticipated, the NNT topology requires a slightly different take on GAM networks: in fact it is not possible to design a similar topology with the communicators provided by GFastFlow out of the box. The reason can be understood by considering a ring topology: from listing 5.2 it is clear that, in order for two nodes to be connected, the output communicator of the first node must be identical to the input communicator of second node; this implies that the output communicator of node 1 ($OC1$) in figure 5.4 must be also the input communicator of node 0 and 2 ($IC0$ and $IC2$).

$$OC1 = IC0 = IC2$$

However, also the output communicator of node 3 ($OC3$) must be equal to $IC2$, and concurrently equal to $IC4$, but $IC4$ must not be the same as $IC2$, since the set of their neighbours, from which they should receive gradients, is different.

$$OC3 = IC4 \neq IC2$$

[2] It can be distributed or not whether multiple processes are launched on physically different machines.

A feasible solution would be to use "routing nodes" that are not executing business logic, but only redirecting pointers to the right output communicator; still, this would increase significantly the complexity of the network, possibly impacting the user productivity. It should be noted that it would be possible to connect all the workers with plain communicators in a chain, but that would entail a privileged direction in the network, so the gradients will flow along that direction instead of being spread in the network. This idea is not wrong *per se*, but it is not the goal of this work.

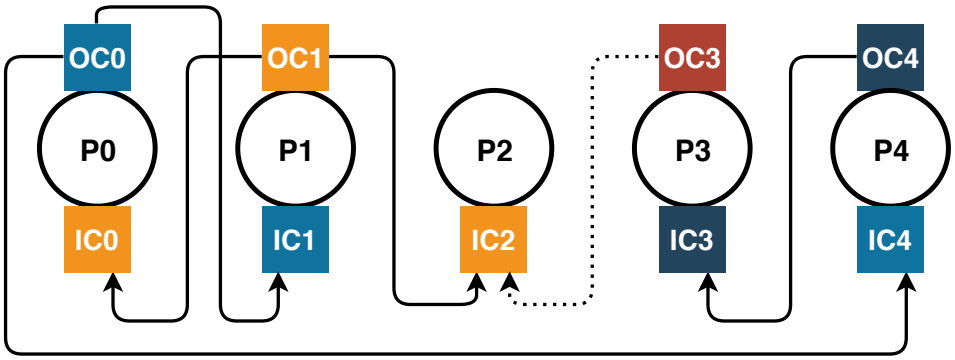


FIGURE 5.4: Unfeasible attempt to impement ring topology with plain communicators. Input communicator below, output communicator above. It is not possible to trivially implement such topology without additional nodes that route the data between neighbours.

For this reason, GFastFlow has been extended in the context of this work so it is now able to support what has been referred to as *communicator bundles*. These structures present exactly the same API as normal communicators, therefore they can be specified as template arguments for GFastFlow processors, but they contain an array of communicators. This allows a processor to be actually connected to more than one communicator, while keeping the complexity of the network under control.

Figure 5.5 represents a ring topology implemented with communicator bundles, above the processors there is the output communicator bundle, below there is the input communicator. Each bundle contains a number of communicators equal to the number of neighbours, which is two in case of a ring topology and 4 in case of a grid topology.

Listing 5.3 shows how to create a 2D-torus topology as shown in figure 4.3b. In FAST design this kind of topology is *already provided* (along with the ring topology) to the user, so he is not required to implement the graph of processors and communicators. Still, using wise indexing it is easy to extend this approach to higher dimensions. Also in this case, no SPMD mechanisms are needed to describe the topology, not even within the nodes.

LISTING 5.3: Definition of a 2D-torus grid topology for NNT.

```
size_t grid_h = // user defined (>= 3)
size_t grid_w = // user defined (>= 3)
size_t workers = grid_h * grid_w;
// Define incoming communicators and outgoing comm bundles for
  each node. Row major ordering.
```

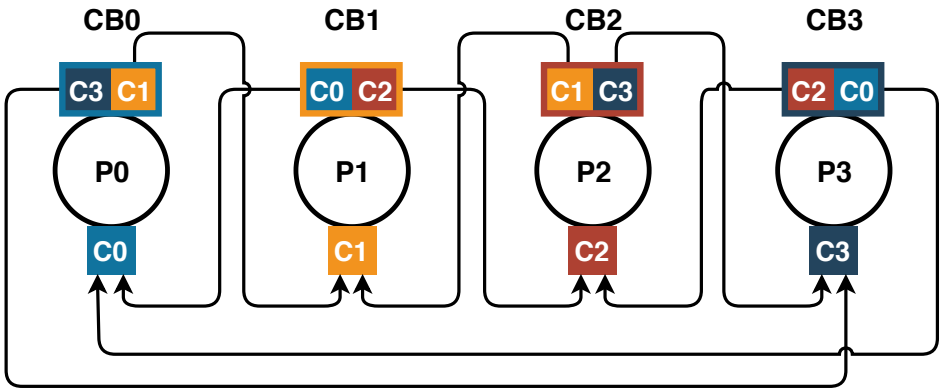


FIGURE 5.5: Ring topology correctly implemented with communicator bundles. Input communicator below, output communicator bundle above. The bundles *CBx* contain a number of communicators that is equal to the number of neighbours.

```
5  std::vector<std::vector<gff::NondeterminateMerge>>
     incoming_channels(grid_h);
6  std::vector<std::vector<gff::OutBundleBroadcast<gff::
     NondeterminateMerge>>> outgoing_channels(grid_h);
7
8  // Create inbound communicators
9  for (int i = 0; i < grid_h; i++)
10   for (int j = 0; j < grid_w; j++)
11     incoming_channels.at(i).emplace_back();
12
13 // Create outbound communicators
14 for (unsigned int i = 0; i < grid_h; i++)
15   for (unsigned int j = 0; j < grid_w; j++)
16     outgoing_channels.at(i).emplace_back();
17
18     // Add neighboring channels (i+1,j),(i-1,j),(i,j+1),(i,j-1)
     in torus topology.
19     unsigned int up, right, down, left;
20     right = j + 1;
21     left = j - 1;
22     up = i + 1;
23     down = i - 1;
24     // Wrapping boundaries to achieve torus behaviour
25     if (up == grid_h)   up = 0;
26     if (down == -1) down = grid_h - 1;
27     if (right == grid_w)   right = 0;
28     if (left == -1) left = grid_w - 1;
29     outgoing_channels.at(i).at(j).add_comm(
30         incoming_channels.at(up).at(j));
31     outgoing_channels.at(i).at(j).add_comm(
32         incoming_channels.at(down).at(j));
33     outgoing_channels.at(i).at(j).add_comm(
34         incoming_channels.at(i).at(right));
35     outgoing_channels.at(i).at(j).add_comm(
36         incoming_channels.at(i).at(left));
37   }
38 }
39
40 // Create processor nodes
41 for (unsigned int i = 0; i < grid_h; i++)
42   for (unsigned int j = 0; j < grid_w; j++)
```

```
43      gff::add(WorkerNode(
44          incoming_channels.at(i).at(j), outgoing_channels.at(i).
    at(j)));
45
46  // execute the network
47  gff::run();
```

The template type of the communicator bundles is a communicator itself, and represents the policy of the communicators contained by the bundle. At the bundle level, in this case it is used the policy `OutBundleBroadcast`, which forwards the pointer received by the processor to all the communicators included in the bundle which, in turn, will apply their own policy.

5.2.4 Node-level parallelism

Up to this point it has been discussed the topology of the multiple worker nodes that make up the NNT training network, still, there is the need to understand how to implement the inner mechanism of the individual worker nodes.

Before tackling more training-related aspects, it is worth to introduce the need for a sound multithreading support at node level. Despite the non-blocking nature of GAM communications, a fair amount of latency hiding is still required to achieve satisfying global performance figures: this of course requires node-level concurrency. In practice, this is achieved by exploiting parallel patterns provided by the the FastFlow library [151], introduced below, and in particular the *pipeline* pattern.

FastFlow

FastFlow [151] is an open source programming framework for structured parallel programming, targeting shared-memory multi-core and supporting the exploitation of GPU accelerators. Its efficiency stems from the optimized implementation of the base communication mechanisms and from its layered design (cf. figure 5.6), based on C++ templates. FastFlow provides a set of algorithmic skeletons addressing both stream parallelism (e.g., *farm* and *pipeline*) and data parallelism (e.g. map, stencil, reduce), along with their arbitrary nesting and composition [155]. Map, reduce, and stencil patterns can be run on multi-cores or can be offloaded onto GPUs. In the latter case, the user code can include GPU-specific code (i.e., CUDA or OpenCL kernels). For instance, leveraging the farm skeleton, FastFlow exposes a *ParallelFor* pattern [156], where chunks of a loop iterations are streamed to be executed by the farm workers. Just like Intel TBB [157], FastFlow's `parallel_for` pattern uses C++11 *lambda* expression as a concise way to create function objects: lambdas can "capture" the state of non-local variables, by value or by reference, and allow functions to be syntactically defined where and when needed.

From the performance viewpoint, one distinguishing feature at the core of FastFlow is that it supports lock-free (fence-free) Multiple Producer Multiple Consumer (MPMC) queues [158], thus providing low overhead high bandwidth multi-party communications on multi-core architectures for any

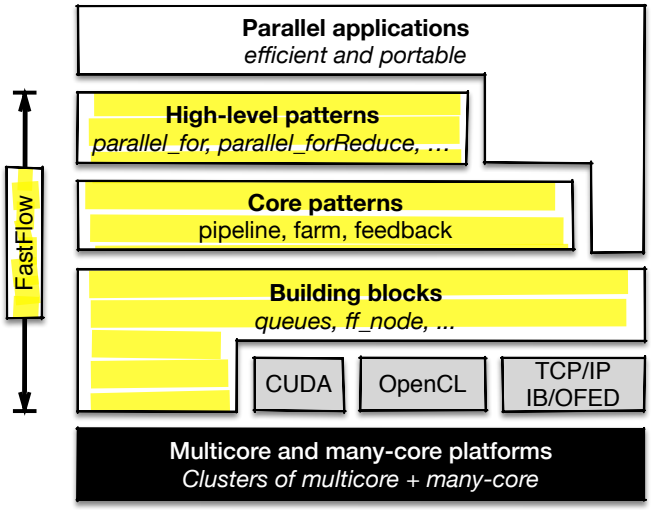


FIGURE 5.6: Layered FastFlow design.

streaming network, including cyclic graphs of threads. The key intuition underlying FastFlow is to provide the programmer with fast lock-free Multiple Producer Single Consumer (MPSC) queues and Single Producer Multiple Consumer (SPMC) queues—that can be used in pipeline to build MPMC queues—to support fast streaming networks.

Traditionally, MPMC queues are built as passive entities: threads concurrently synchronize (according to some protocol) to access data; these synchronisations are usually supported by one or more atomic operations (e.g., Compare-And-Swap) that behave as memory fences. FastFlow design follows a different approach: to avoid any memory fence, the synchronisations among queue readers or writers are arbitrated by an active entity (e.g., a thread). We call these entities *Emitter* (E) or *Collector* (C) according to their role; they actually read an item from one or more lock-free Single Producer Single Consumer (SPSC), queues and write onto one or more lock-free SPSC queues. This requires a memory (pointer) copy but no atomic operations.

The advantage of this solution, in terms of performance, comes from the higher speed of the copy operation compared with the memory fence; this advantage is further increased by avoiding cache invalidation triggered by fences. This behaviour also depends on the size and the memory layout of copied data. The former point is addressed using data pointers instead of data, ensuring that the data is not concurrently written: in many cases this can be derived by the semantics of the skeleton that has been implemented using MPMC queues—for example, this is guaranteed in a stateless farm as well as many other cases.

The *pipeline* skeleton is composed by a number of workers connected sequentially by lock-free SPSC queues, that model computations expressed in stages. Formally, the pipeline behaves as the composition of the functions performed by the stages on the input of the first stage. The parallel semantics of the pipeline skeleton ensures that all the stages will be execute in parallel, thus, neglecting transients, the pipeline is able to process several input elements in the same time that would be required by a single element

to go through all the stages once. the exact speedup that can be achieved by a pipeline depends on several factors, including the load balancing between its stages.

5.3 Training node

The training node is in charge of performing the local training of the model replica in the data parallel NNT set-up. In this case it is identified with a processor in a GFastFlow network based on the filter node type, as discussed in section 5.2.3. The training processor object can be defined as following

```
typedef gff::Filter<gff::NondeterminateMerge,
      gff::OutBundleBroadcast<gff::NondeterminateMerge>,
      gam::public_ptr<FAST::gam_vector<float>>,
      gam::public_ptr<FAST::gam_vector<float>>,
      FAST::MXNetWorkerLogic<ModelLogic, float>>
    MxNetWorker;
```

where the input and output data types are GAM public pointers to `FAST::gam_vector<T>`: it is a class that inherits from `std::vector<T>` and that is enriched with facilities to be transparently transferred over a GAM network. This data type is meant to contain gradients transferred between worker nodes, its template type, that is specified as `float` in the `MxNetWorker` node definition could be used to extend the presented approach to quantised (i.e. with smaller precision, such as 8-bit integers) or otherwise compressed gradients.

It is useful at this stage to discuss the framework used for the current implementation of FAST. In fact, while the structure of the worker node is common, the code to implement it is specific for the underlying framework. In this case the choice fallen on MxNet [64]: at the beginning of the development it was, in fact, the DL framework with the most expressive C++ API for training among the major ones. FAST provides a worker class

```
FAST::MXNetWorkerLogic<ModelLogic, float>
```

that is specific for a model logic class using MxNet for training that will be discussed briefly in section 5.4 (this class is provided as template argument, on the same line of what is done by the Filter class for the business logic). Beside the worker node, the only framework-specific code required is necessary to convert MxNet tensor objects, containing the gradients, into a proper `gam_vector`, and the other way around.

5.3.1 Control flow and structure

The main task of the worker node is to preform the steps described in algorithm 2. However, as already anticipated, several steps involve data transfer from remote hosts, or data transformation and local copy (possibly between CPU and GPU), forcing the adoption of proper techniques to hide these latencies, described in detail in figure 5.7. For this reason the previous section discussed the implementation of a FastFlow *pipeline*: in fact this pattern is used in the worker node to hide those network IO or memory latencies with actual computation. On top of this, it should be taken into account that each worker receives gradients from all the neighbours defined

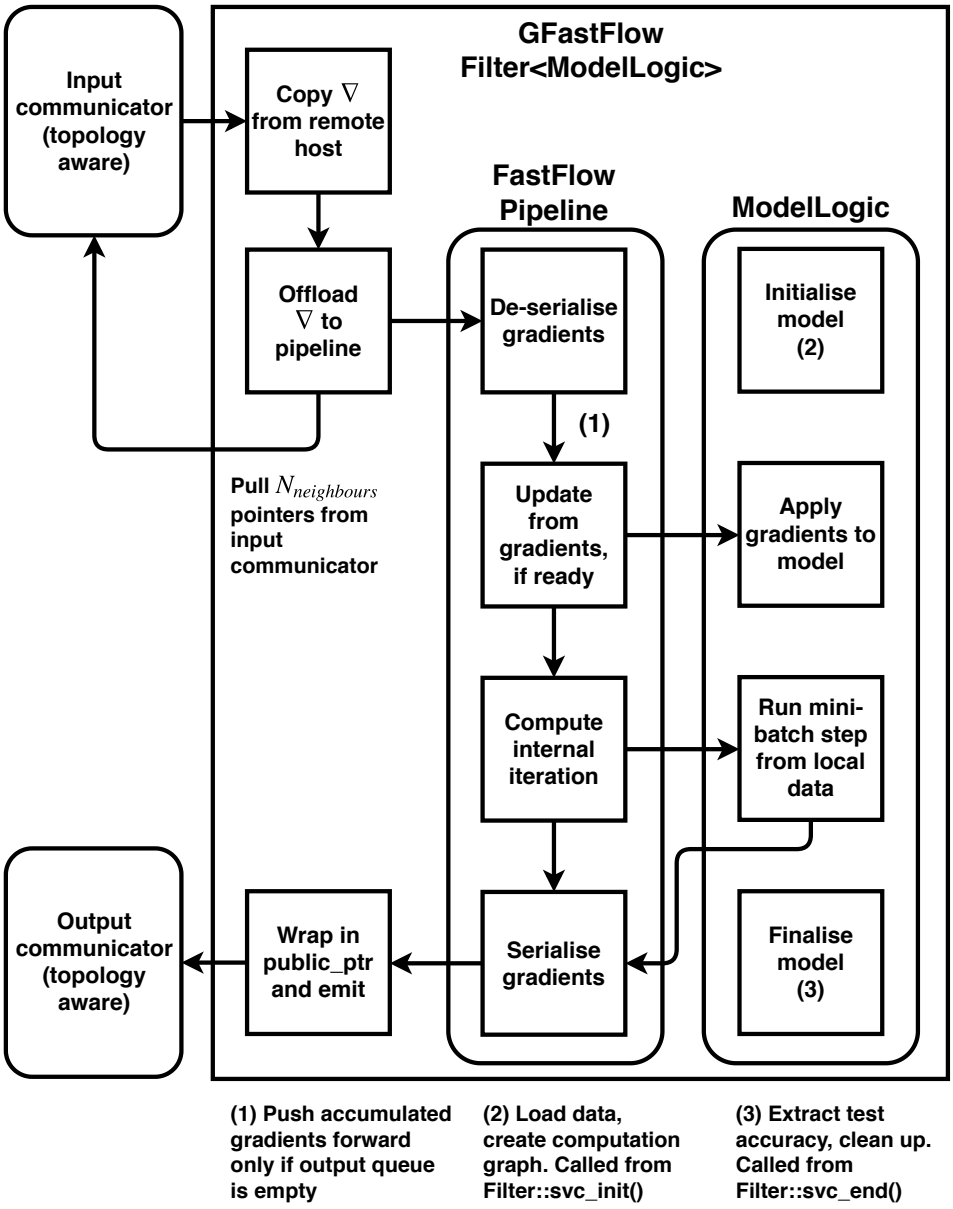


FIGURE 5.7: Detailed architecture of the FAST worker node.

by its topology (e.g. 4 in the 2D-torus case), this means that meanwhile one worker computes its own gradients, it will receive 4 gradients from its neighbours, causing a potential computation-communication imbalance that might fill up the message queues between workers if not handled. To solve this issue, the gradients received are not directly pushed forward in the pipeline, but they are accumulated instead (cf. figure 5.7, reference (1)) until the queue between the accumulation node and the node performing the update from remote gradient is empty.

This structure achieves a number of goals:

- Decoupling of model logic from distributed training logic
- Decoupling of node logic from topology

- Significant performance results thanks to strongly asynchronous execution

Beside these points, it is noteworthy that the MxNet worker logic is completely unaware of the inner mechanisms of MxNet, but it is still capable of achieving good performance results. This is relevant as Mxnet is, by itself, strongly asynchronous in the execution of the computation graph: this could be leveraged by a distributed training framework integrated from within (e.g. the built-in PS) in order to achieve even better scalability. However, this work aimed to be interoperable with different DL frameworks, therefore it has been adopted this solution that trades some scalability in favour of interoperability.

5.3.2 Gradients transfer

Figure 5.7 shows a serialisation and a de-serialisation block that are applied asynchronously to outbound and inbound gradients respectively. This step is needed as gradients are stored by MxNet in GAM a dictionary-like object, while GAM requires data to be contiguous in memory in order to be transferred[3]. This issue could be solved by either copying the gradients in a new, contiguous, memory location, or by defining a protocol that could exchange the elements of the MxNet gradient dictionary one by one. The asynchronous nature of communications in FAST is not really suitable to implement this kind of protocol, as managing the reconstruction of gradients coming from different sources in an interleaved fashion would cause excessive overhead. On the other hand, the former solution, at the cost of an additional copy, bears a number of advantages. First of all, when the model is trained on the GPU, the serialisation is performed as a single operation with the copy from device to host memory, therefore it does not add overhead at all. Moreover, different use-case scenarios may require operations to be applied to the outbound gradients, like quantisation and compression, or even proper serialisation (e.g. string serialisation), when dealing with heterogeneous distributed infrastructures. These operations can be fused with the serialisation of gradients, effectively cancelling its impact.

5.4 Usage

The FAST framework is currently available as a research prototype, although quite mature, in a public repository[4]. This section will discuss the way a user is expected to interact with the tool and the design principles that have driven the development. Specific details about the code are mostly avoided, since the development is still ongoing and some aspects may change in the near future.

5.4.1 User model definition and compilation

The previous section anticipated that the business logic should be provided by the user in the form of a `ModelLogic` class. The structure of this class is

[3] in this context serialisation is used loosely to identify contiguous memory allocation
[4] FAST git repository: https://github.com/paoloviviani/FAST

provided in the documentation of the framework, but broadly follows the structure presented in algorith 1. The required components are:

- An initialisation member function

  It is in charge of loading the data local to the worker and to create or load the computation graph. Moreover, the weights are initialised, possibly based on a common initialisation file.

- A finalisation member function

  It can perform user-defined actions, such as calculating the test accuracy.

- An SGD step member function

  It performs a local optimisation step using a mini-batch extracted from the local data partition.

- A weight update member function

  It gets a pointer to an MxNet object containing the gradients and updates the local model replica according to them, and to the local status of the optimiser.

- A pointer to the gradients

  Either a public data field or member function that returns the pointer to the MxNet object containing the gradients.

- A pointer to the weights

  Either a public data field or member function that returns the pointer to the MxNet object containing the weights.

- A flag that reports termination

  Either a public data field or member function that returns true if the training is over (i.e. because the maximum number of epochs has been reached.)

Other than these required components, the class can contain any function or data field that is necessary to the user to perform training. it should be noted that, most of the times, including the examples provided in the code, the class can be created by cutting and pasting segments of the sequential training code.

In order to compile an executable that can run distributed training, the framework already provides source files with pre-built ring and 2D-torus topologies. The user is only requested to pass its own header file containing the ModelLogic class as a gcc `-include` include flag within a Makefile already available in FAST. A configuration file should be adjusted to the specific paths of the dependencies: libfabric and, in this case, MxNet.

The resulting executable is ready to be launched in a distributed environment thanks to the launcher described in the next paragraph.

5.4.2 Execution

Any distributed application typically requires a launcher that runs the right commands on each node of the distributed architecture. A common example of this idea is the `mpirun` command provided by all the MPI implementations, that exploits Secure Shell (ssh) to execute commands on several compute nodes. GAM itself already provides a launcher that is used to also to set a number of environment variables required to run GAM applications. The peculiarities of FAST lead the development towards a customised, python-based launcher, that is tailored to the needs of distributed training workloads.

The launcher acts as a front-end, while the command specified by means of several arguments is executed on the nodes by means of different back-ends: MPI, Slurm[5], direct ssh, and local multiprocessing. The launcher is named `fast-submit` and accepts a number of parameters: a typical way to launch a FAST executable is the following:

```
$ fast-submit -l [mpi,local,ssh,slurm] -n <number of workers> \
    -H <hostfile> <executable>
```

Where the option `-l` is used to define the back-end launcher, `-n` indicates the total number of worker nodes required, `-H` points to the path of the file containing a list of available hosts where to to run a command and, finally, it takes the name of the executable to be launched. Note that the number of workers specified must agree with what expected by the topology, for instance, assuming an executable `resnetGrid` where a user model is compiled with the FAST-provided 2D-torus topology, the command to run it on an MPI cluster is the following:

```
$ fast-submit -l mpi -n 16 -H /path/to/hostfile resnetGrid 4 4
```

Where the numbers following the executable name are arguments to the executable itself defining the size of the grid of workers, as per listing 5.3. In this case, a 4×4 grid will need the 16 workers specified in the launcher.

[5] https://slurm.schedmd.com

Chapter 6

Evaluation

6.1 Experimental set-up

This chapter will finally assess how the discussion carried out up to this point stands the test of the facts. The problem used to validate NNT and FAST has been selected to be challenging enough to present reasonable granularity, while not so large to require infeasible development-debugging-validation cycles due to the time constraint of this dissertation. As it will be clear later on, this choice possibly unfavourably impacted the comparison between state-of-the-art distributed techniques and the presented approach, which will be investigated and improved also beyond the scope of this work. The test case used for benchmarking is a common image classification task from literature, based on the CIFAR-10 [94] and the ResNet model [93, 159]. The CIFAR-10 dataset consists of 60000 32x32 RGB images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. Classes and example images are shown in figure 6.1. The CIFAR-10 dataset also appears in industry benchmarks [112, 122].

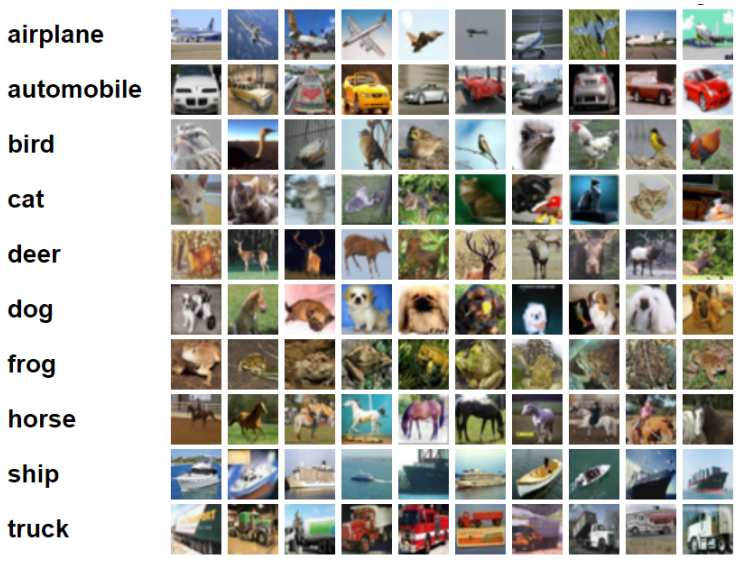


FIGURE 6.1: Sample of the CIFAR-10 dataset, from [160].

Residual Networks (ResNet), on the other hand, are a well-known model appeared in literature and among the top ranks in benchmarks. The specific version used in this case is ResNet18-v2 [93], which provides good accuracy results without the burden of models using more layers like ResNet50, or ResNet152, which are also well-known in literature.

6.1.1 Hardware used

While deep learning models are commonly trained and deployed using GPUs, it is not trivial to get access to a reasonably large multi-node GPU machine. On the other hand, it has been possible to access a 16-nodes machine for long enough to perform development and validation.

The results reported in section 6.2 are obtained on an HPC infrastructure[1] provided by the *Competency Center on Scientific Computing* (C3S) at University of Turin [10]. To run experiments were used 16 CPU nodes equipped with 2 Intel Xeon E5 with 12 physical cores each and 128Gb of memory. Infiniband interconnection is available on the nodes and has been used, although only as IP over InfiniBand (IPoIb), that showed no appreciable improvement with respect to the also available Ethernet.

Further experiments are being performed on a larger infrastructure[2] provided by the Pawsey Supercomputing Centre in Western Australia, however their scope goes beyond this dissertation, and results are not mature yet to be presented here.

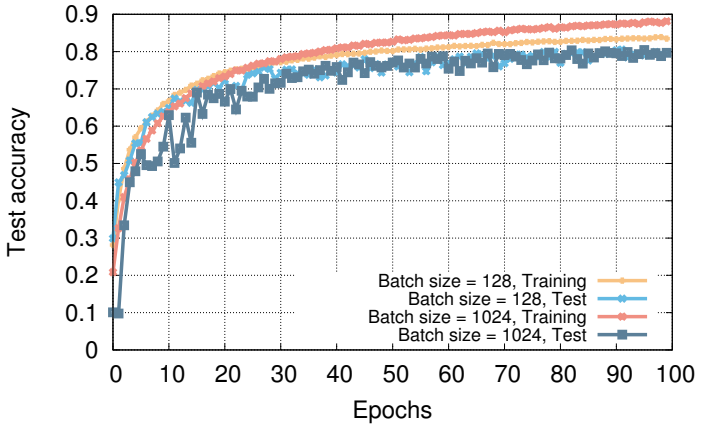
6.1.2 Baseline

In order to provide a baseline benchmark for this implementation it is used a C++ single node training code based on MxNet. The code is based on examples provided by MxNet developer on the GitHub repository of the framework[3]. Exactly the same code has been used to train ResNet with NNT thanks to FAST and its capability to wrap existing code with minimal modifications. Mxnet ha been compiled against Intel MKL-DNN [73] to provide reasonable performance on CPU. Figure 6.2 shows the training and test accuracy trend for two different batch sizes for the single-node training.Sub-figure 6.2b highlights how larger batches are definitely more efficient to compute, as 100 epochs take roughly half of the time to be processed than in the case of $n_b = 128$. On the other hand, larger batches present a larger gap between training and test accuracy that let it be understood that generalisation performance is worse. Despite this fact, the final result after 100 epochs is on par with the smaller batch size and it is achieved much faster.

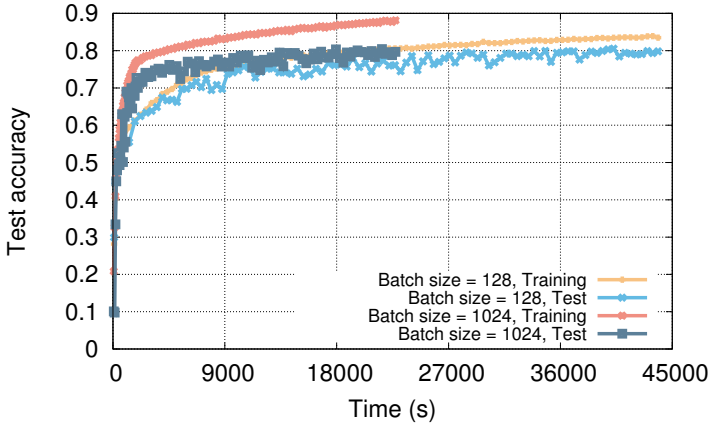
Configuration

MxNet C++ and Python API eventually point to the same underlying engine, hence instruction can be roughly mapped 1-to-1 between the two versions. Here are briefly reported the main settings used for the training using the Python API. The specific hyper-parameters are chosen based on MxNet examples and literature, however *the absolute test accuracy at the end of the training is not as interesting in this scope as the relative performance of the different approaches*, hence there is no claim of superior absolute performance with respect to published results. All the code, including sequential C++, is available in the FAST repository[4].

[1] Detailed technical specifications:
https://c3s.unito.it/index.php/super-computer/super-computer
[2] Specifically the Magnus machine: https://pawsey.org.au/systems/magnus/
[3] Mxnet repository: https://github.com/apache/incubator-mxnet
[4] FAST git repository: https://github.com/paoloviviani/FAST



(A) Test accuracy vs. index of the epoch.



(B) Test accuracy vs. wall-clock time for all the 100 epochs. Larger batches are, as expected, significantly more efficient to compute, as 100 epochs are processed in roughly half the time.

FIGURE 6.2: Training curves for ResNet18-v2 [93] on the CIFAR-10 dataset [94] with MxNet and MKL-DNN [73], capped at 100 epochs. Node specifications: 2x Xeon E5-2680 v3, 12 core 2.5Ghz, 128Gb of memory.

Data loading from CIFAR-10 files is performed as following, with minimal data augmentation (i.e. data shuffling, random cropping, and mirroring):

```
train_data = mx.io.ImageRecordIter(
  path_imgrec = "./cifar/cifar10_train.rec",
  resize      = False,
  data_shape  = (3, 32, 32),
  batch_size  = batch_size,
  rand_crop   = True,
  rand_mirror = True,
  shuffle     = True,
  pad         = 4,
  num_parts   = store.num_workers,
  part_index  = store.rank)
```

Weights initialisation and the optimiser are configured with the following instructions:

```
# Initialize the parameters with Xavier initializer
net.collect_params().initialize(mx.init.Xavier(magnitude=2), ctx=
    ctx)
# Use Adam optimizer. Ask trainer to use the distributer kv store.
trainer = gluon.Trainer(net.collect_params(), 'adam', {'
    learning_rate': 0.01, 'rescale_grad': 1.0/(batch_size*store.
    num_workers), 'clip_gradient': 10}, kvstore=store)
```

Note that the normalisation mentioned in theorem 1 is used here. This normalisation is not used in NNT, where the gradient is only divided by the local batch size. The computation graph of the ResNet model is directly provided by the MxNet Python API.

6.1.3 Competitors

To provide this work a fair comparison, the choice fell on the distributed implementation of MxNet, that in principle guarantees the same single-node performance. MxNet provides synchronous and asynchronous SGD out of the box, implemented in a centralised way and based on the ZeroMQ [131] library for messaging. Despite the centralised implementation, the result is highly efficient due to the knowledge of the inner working of the framework and the direct access to relevant data structures that are managed in a strongly synchronous way. This fact, coupled with the reasonable problem granularity that leaves room for a synchronous approach to perform well, results in a very efficient implementation that turned out to be almost impossible to beat with a general tool that does not make assumptions on the underlying framework.

The FAST-based Nearest-Neighbours Training approach has been compared against the sequential version, the synchronous SGD, and the asynchronous SGD using the same hyper-parameter configuration. Whenever different hyper-parameters are used, either multiple configurations are reported for comparison, or only the best configuration is considered as the most relevant.

6.1.4 Nearest-Neighbours Training

A few remarks can be made also on the peculiarities of the NNT training used in this benchmark. First of all, the training strategy is based on figure 4.4, therefore a synchronisation is imposed every epoch to identify the model with the best test accuracy and re-start all the replicas from its weights. In fact both the test and the training accuracy of model replicas have been observed to slowly drift away from each other if no synchronisation is ever imposed during the training, this represents the cost of full model inconsistency. On the other hand, enforcing a barrier at every epoch is definitely acceptable, as the granularity is very large (e.g. several seconds).

Another point to be considered is that, in order to correctly implement a specific topology, there is a minimum number of nodes. For instance, the ring topology requires at least 3 workers, otherwise the right neighbour and the left neighbour will be the same. That holds for the grid topology, that requires at least a 3-by-3 grid with 9 worker nodes: any configuration with

less nodes will imply that a processor has two communicators pointing to the same worker, that will receive the same updates twice.

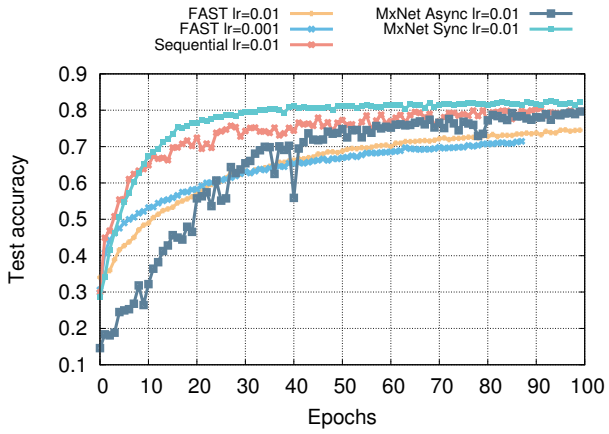
6.2 Results

6.2.1 Time-To-Accuracy

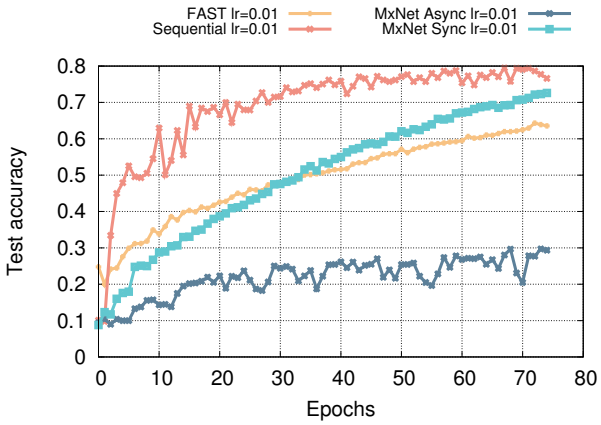
This section will report in detail the convergence results of NNT with respect to other approaches. The metric that is considered when discussing convergence is the top-1 accuracy measured on the test dataset (namely, the times the predicted output with highest probability is equal to the label of the test sample).

Figure 6.3 shows the training curves for single-node SGD, state-of-the-art MxNet implementations and NNT based on FAST for a per-node batch size of 128 and 1024. The considered metric is the top-1 accuracy measured on the test dataset (namely, the times the predicted output with highest probability is equal to the label of the test sample). The training is arbitrarily capped at 100 epochs (75 for batch size 1024), since it is sufficient to achieve reasonable accuracy and to show relevant behaviours. For a per-node batch size of 128, the obvious result from these figures is that the synchronous approach does not suffer at all from the generalisation issues showed by previous work, at least for this set-up. Indeed, synchronous SGD is both the fastest to converge when counting epochs and in terms of wall clock time. This is probably due to the fact that, despite the centralised implementation of MxNet, the knowledge of the inner working of the framework, the direct access to relevant data structures that are managed in a strongly synchronous way and the reasonable problem granularity, which leaves room for a synchronous approach to perform well, result in a very efficient implementation that turns out to be almost impossible to beat with a general tool that does not make assumptions on the underlying framework. The plot in figure 6.3a, which shows the training curves with respect to the number of epochs for a per-node batch size of 128, presents a smooth convergence for synchronous and sequential SGD, while NNT is slightly slower to reach higher accuracies, struggling to go beyond 70% top-1 accuracy by the 100 epochs threshold. The asynchronous approach is very slow to converge at the beginning, but eventually catches up with the rest of the group. Figure 6.3c considers wall-clock time instead, for the same per-node batch size. It only reports details related to the first epochs, as the sequential training is so slow that it is impossible to have a clear overview when representing it entirely. Here again the advantage of synchronous SGD is clear: no competitor can even come close to it. Nevertheless, NNT with FAST is performing definitely better than the sequential version and, somehow not surprisingly, it is comparable with the asynchronous method. Finally, the asynchronous method overtakes FAST reaching higher final accuracy.

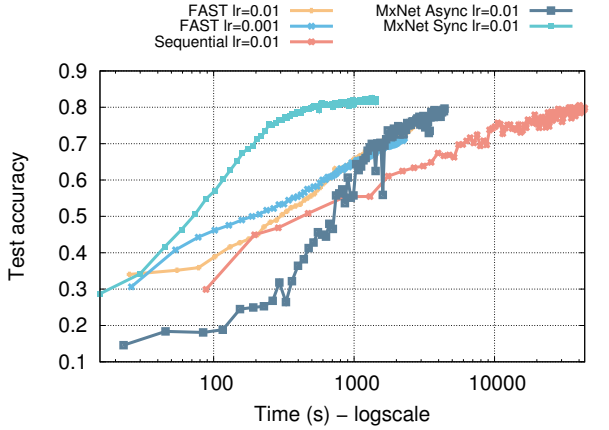
On the other hand, a per-node batch size of 1024, as shown by figure 6.3b and 6.3d, is large enough to present the first signs of the effect predicted by figure 2.5. In fact the synchronous version is much slower to converge with respect to the smaller batch size, and the the asynchronous one performs even worse. Conversely, NNT seems much less affected by the larger batch size: for instance, while MxNet synchronous is 5 times slower



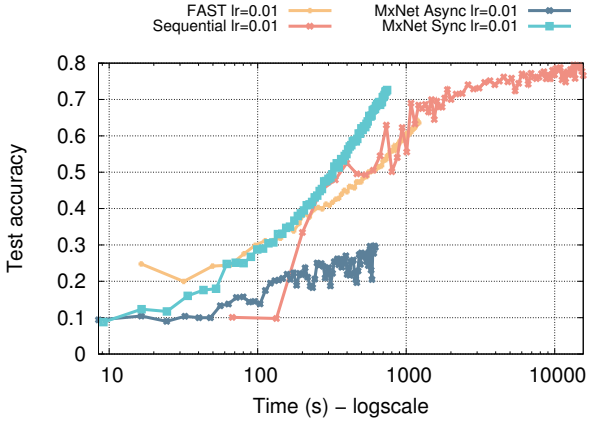
(A) Test accuracy vs. index of the epoch. Per-node batch size is 128.



(B) Test accuracy vs. index of the epoch. Per-node batch size is 1024.



(C) Test accuracy vs. wall-clock time, time in log scale. Per-node batch size is 128.



(D) Test accuracy vs. wall-clock time, time in log scale. Per-node batch size is 1024.

FIGURE 6.3: Training curves for ResNet18-v2 [93] on the CIFAR-10 dataset [94] with MxNet and MKL-DNN [73], capped at 100 epochs. *lr* denotes the learning rate used. Sequential version vs. 16 nodes with FAST (grid topology) vs. 16 nodes synchronous SGD. Node specifications: 2x Xeon E5-2680 v3, 12 core 2.5Ghz, 128Gb of memory.

to reach 60% accuracy with $n_b = 1024$ compared to $n_b = 128$, NNT is only two times slower.

Discussion

The behaviour just presented should be analysed carefully. It cannot be concealed that FAST is still no match for synchronous MxNet for this specific problem and set-up, but there is some bright side that can be inferred by these results. First of all, the proposed approach is indeed successful in achieving a better performance-accuracy trade-off than the sequential implementation, as it reaches reasonably close accuracy results in much shorter time (i.e. with a much steeper accuracy vs. time learning curve). Moreover, it also sports a better trade-off than the asynchronous method for small batches, and it is definitely better performing for large batches. Finally, the gap between state-of-the-art synchronous SGD and NNT gets smaller as the batch size gets larger: while still not conclusive, this represents a circumstantial evidence that *the proposed approach is less sensitive to larger per-node batch size than the synchronous implementation*. As a consequence, it is reasonable to expect that, given the same global batch size used here ($16 \cdot 1024 = 16384$) arranged on more nodes (e.g. 128 nodes with local batches of 128 samples), the synchronous implementation should behave exactly in the same way as it does in figures 6.3b, while NNT with FAST should behave more like in figure 6.3c, therefore it is expected to outperform the synchronous SGD. Moreover, *the smaller granularity and the larger number of nodes is expected to undermine the scalability of the synchronous version, while FAST should be much less affected*.

While a formalisation for NNT convergence is lacking for now, it is possible to make some educated guesses on the reasons behind the observed results. It is sensible to expect that the inconsistency between workers adds a certain amount of noise to the optimisation process, given that multiple model replicas have the capability to explore the space of $\mathbf{w}$ much more than a single sequential worker. This noise can be beneficial in the initial part of the training, where NNT shows good performance. On the contrary, the asynchronous strategies possibly have too much noise in the first stages, leading to worse results in terms of convergence. Conversely, in the long term an excessive amount of variance can hinder the performance of NNT, since it struggles to go far beyond 70% of accuracy. These observations may suggest that a way to get the best of both worlds can be, for instance, using NNT as a warm-up phase for large batches, whenever those are struggling to converge early in the training process. This would allow for a much faster warm-up than with sequential small batches.

The fact that the synchronous strategy is only marginally affected by the larger mini-batches might be related to the specific problem selected for validation, as this effect is confirmed by previous literature [50] for the ImageNet dataset [134]. The good convergence achieved in this case somehow hides the main issue related to synchronous SGD with large batches, while the tight integration with MxNet, the suitable problem granularity, and the size of the test cluster prevented the author to highlight the expected scalabilty issues of this approach. In this sense it is not possible to claim strong performance advantages related to the presented work, but its relevance is beyond doubt, as it achieves good convergence despite dropping most

of the usual requirements of model consistency and completeness of communication. Its low sensitivity to the per-node batch size cannot be overstated, as it supports, albeit with circumstantial evidence, the initial guess that drove the development of NNT. Also relevant is that NNT appear to be consistent, possibly better than what is reported for CIFAR-10 in the GossipGrad paper [4], although full results for comparison are not available for that case.

6.2.2 Scalability

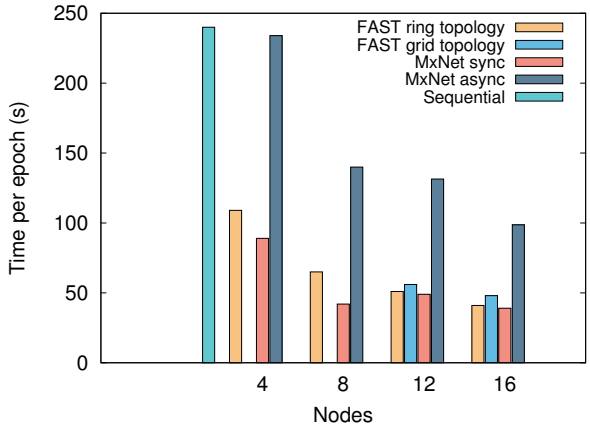
This section will deliberately ignore convergence performance, focusing on the efficiency of the FAST runtime system. This is achieved by measuring the time needed to run a single epoch of training, regardless of the accuracy. An epoch is completed when the model has executed backpropagation on the whole dataset: in the distributed case the whole network of nodes is considered, hence an epoch is completed when each worker completes its own fraction of the training dataset in a data parallel fashion. Figure 6.4 reports the results achieved, both in terms of time and in term of speedup. Note that the grid topology can't be applied for less than 9 workers in a 3x3 toroidal grid. The first thing that should be observed, is the super-linear speedup achieved for one of the presented configurations, this result should be taken with a grain of salt: in fact it has been observed a degradation of the performance in the sequential implementation as the training proceeds, actually increasing the average time per epoch for that deployment. This effect did not manifest itself in any of the distributed versions, those have shown much more stable performance. For instance, this degradation has not been observed in sequential training on GPUs which is not reported here, this means that in other set-ups results may vary. In this sense the focus should be more on the comparison among distributed approaches than on the absolute speedup figures.

These tests show how the MxNet implementation of asynchronous SGD is not performing well compared to the synchronous one: in fact *it clearly suffers the small problem granularity of batch size 32*, where in principle it should have had an edge over the synchronous approach that proceeds in a lock-step fashion on small tasks. On the other hand, the latter one is the most efficient of the group in all the cases observed.

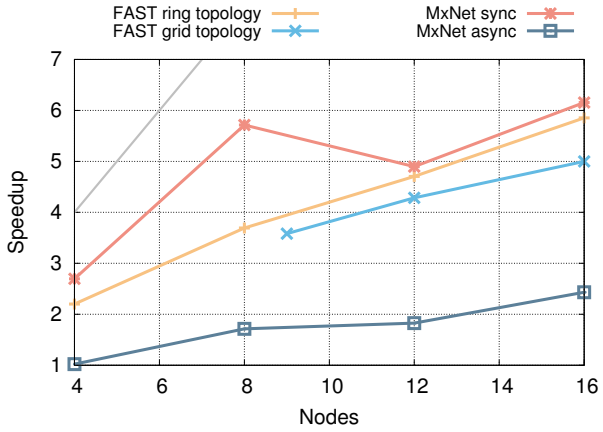
The performance of FAST is good in all the circumstances, even if for large granularities (e.g. $n_b = 1024$) MxNet is faster. The most satisfying plot is, however, figure 6.4b: in fact it shows how FAST *catches up with MxNet performance in case of large deployments with small batches*. This is not unexpected, as the lock-step processing of mini-batches, and the bottleneck represented by the parameter server, are detrimental to performance. It is also possible to not that, also unsurprisingly, the ring topology scales slightly better than the grid topology, due to the smaller communication complexity.

Discussion

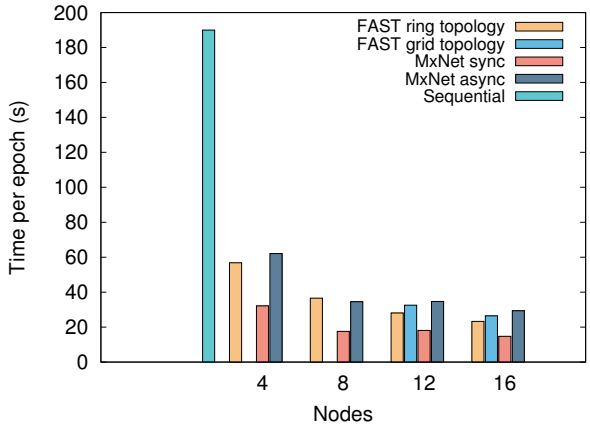
FAST has been designed in order to be agnostic with respect to the underlying deep learning framework, this required a number of trade-offs that could potentially impact its performance linked to the missing knowledge



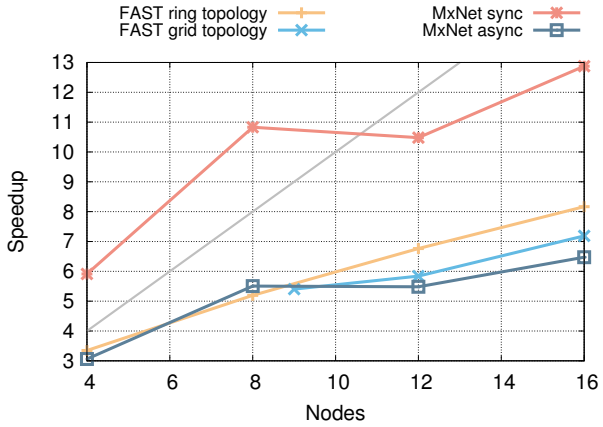
(A) Average time per epoch. Mini-batch size = 32.



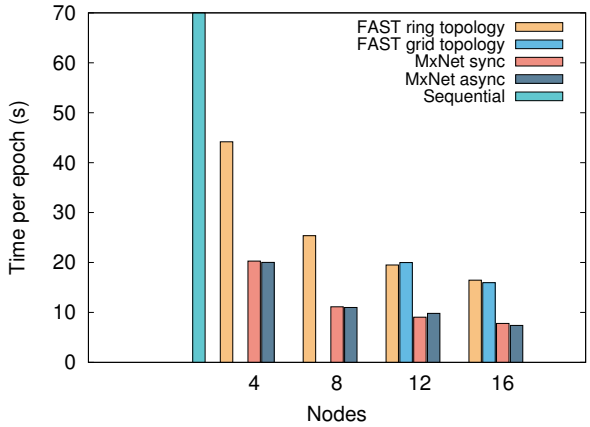
(B) Speedup with respect to sequential time per epoch. Mini-batch size = 32. Gray line represents the ideal trend.



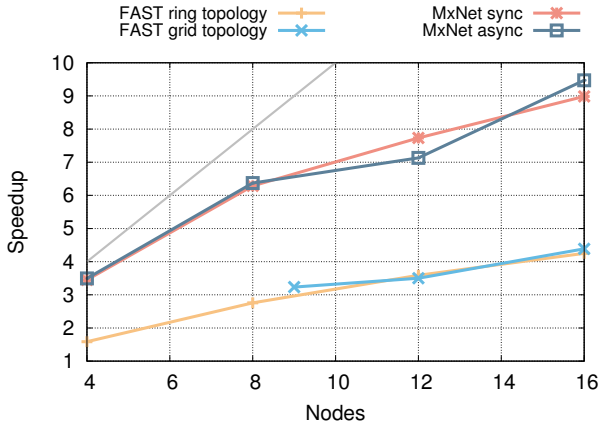
(C) Average time per epoch. Mini-batch size = 128.



(D) Speedup with respect to sequential time per epoch. Mini-batch size = 128. Gray line represents the ideal trend.



(E) Average time per epoch. Mini-batch size = 1024.



(F) Speedup with respect to sequential time per epoch. Mini-batch size = 1024. Gray line represents the ideal trend.

FIGURE 6.4: Scalability of the approaches under evaluation. Super-linear speedup for figures 6.4d is due to the *degradation of the sequential performance* observed with the progress of training, that raises the average time-per-epoch. That effect is not observed in distributed deployments, despite almost identical code.

of the inner mechanisms of the framework. In fact, Mxnet uses a highly asynchronous runtime system that manages the execution of the computational graph that would potentially lead to unpredictable results if an external application tries to read MxNet tensors during the training. In particular, the reported content of the tensor might be outdated, or inconsistent. The distributed runtime of MxNet is aware of these mechanisms, and possibly exploits them to achieve top-notch performance, at least for the synchronous case. On the other hand, FAST is forced to synchronously wait for gradients and weighs to be written, before accessing them. This has an impact on the efficiency of the distributed runtime: for instance, gradients have to be serialised before exchange (in the sense that they must be placed in a contiguous memory location, as opposite to string serialisation), and de-serialised upon reception, operation that is not necessary in the case of MxNet. However, this step that is detrimental to the performance of CPU training, should not be so significant in case of GPU training, where the device-to-host copy is fused with the serialisation in a single operation.

Despite these trade-offs, synchronous communications and parameter servers unavoidably become a burden when dealing with smaller and more frequent tasks, moving the balance from computation to communication, as shown in figure 6.4b. This is the case where FAST, thanks to it NNT topology, excels. Fully asynchronous, non-blocking message passing is able to catch up with the limitations imposed by other design choices and eventually match the performance of MxNet.

To conclude, while speedup plots may suggest a large gap between state-of-the-art MxNet and FAST, the left side of figure 6.4 shows that time results are quite close to each other, and definitely better than sequential times: real world training would definitely benefit from either FAST or distributed MxNet, as they both provide satisfactory performance.

6.3 Summary

It is clear from the results presented above that the outcome of this work, at least at this stage, is twofold. This approach still lags behind optimised synchronous deployments, that have the edge both on the convergence and the scalability side in most use-cases. On the other hand, NNT implemented with FAST is indeed able to converge, and it does it, at least initially, at a faster rate than the sequential SGD in terms of time, despite the number of theoretical requirements that have been given up. Moreover, it is on par with state-of-the-art asynchronous SGD, with a definitely smoother convergence profile.

The most interesting point is, however, highlighted by the corner cases explored in this chapter: very large batches for convergence, and very small batches for scalability. The deterioration of the convergence rate shown by synchronous SGD for a global batch size of 16384 points towards a potential advantage of the NNT approach in case of very large scale deployments, where synchronous methods would behave as a very large mini-batch even with small local batch sizes, while NNT should retain the better convergence properties of small batches. This also means that NNT is expected to be less sensitive to hyper-parametrisation than synchronous SGD, as it

does not behaving like a single huge mini-batch even in case of large deployments. Of course this claim is not directly backed by data, but the previous discussion points in this direction, and this strategy deserves more investigation.

Considering runtime efficiency, better scalability demonstrated on smaller tasks is always a good sign: this also means that, given a proper GPU cluster available, the shorter time required by GPUs to compute mini-batches should be better tolerated by FAST than by MxNet. This coupled with the device-host copy required to MxNet, thet is already in place for FAST due to serialisation, sets high expectations for the application of this approach on GPU-equipped HPC machines.

To summarise, the present work requires further investigation to better identify its strengths when compared to mainstream algorithms, but it is definitely promising. Moreover, distributed deep learning strategies are not mutually exclusive: while model parallelism can be easily added on top of data parallel training, also different distributed optimisation strategies can be coupled to achieve globally better results faster. As already anticipated, the good properties of NNT in the initial phases of the training could make it a good candidate for the warm-up phase of synchronous methods, that can kick in when a convex region is reached, in order to reach a higher final training accuracy, that might be difficult to reach with NNT alone due to the persistent variance brought by neighbouring workers.

Chapter 7

Conclusion

7.1 Remarks and conclusion

At this stage, the quest for training performance at scale has been met mostly by synchronous, large mini-batch, parallelism. Unfortunately, literature defines this strategy as heavily reliant on hyper-parameters fine-tuning and not suitable for other platforms than conventional HPC clusters and tightly coupled cloud instances. To some extent also this work observed that pushing the global batch size too far is detrimental to the convergence of the training. This dissertation advocates a departure from both synchronous and conventional asynchronous training, as the latter has been shown to be even more sensitive to hyper-parameter tuning than the former, without providing a real advantage over synchronous strategies, with witch it shares most of the drawbacks due to the complete communication graph.

In this regard, a training methodology with an incomplete topology for gradient communication, based on a nearest neighbours scheme, has been proposed. Dropping the completeness of the communication graph represents a completely different approach with respect to any approximation of large mini-batches. Furthermore, exchanging gradients between individual workers can provide comparable scalability and some improvement in convergence with respect to large mini-batches, especially in early training stages. Moreover, there are practical advantages due to the limited communication complexity, which can affectively applied outside the typical cloud/HPC scenario (e.g. for training on mobile devices).

In order to evaluate the proposed strategy, the Flexible Asynchronous Scalable Training (FAST) framework has been developed, which allows to apply this novel communications approach to a deep learning framework of choice, with minor modification to an already-available training code. While still a research prototype, it is the author's opinion that the framework approach advocated by FAST, and followed also by other implementations like Horovod [127] represents the way to go. In fact, it allows the researcher to completely decouple the phase of development of the model from the phase of the deployment on a distributed infrastructure. Hopefully, a deeper theoretical investigation will also provide a strategy for distributed training that preserves the effect of the model hyper parametrization, allowing a deep learning model to be trained in parallel without the need to, for instance, adjusting the learning rate.

By the time of presenting this work, most of the main deep learning

tools are able to provide mostly seamless distributed training tools, however, all of them still provide only synchronous strategies (or at most PS-based asynchronous ones), while implementing other approaches within the boundaries of the tools themselves is a daunting task. Moreover, it can be argued that a unified, external approach for distribution of training can support the portability of performance among different infrastructures and deployments.

Thanks to its implementation with FAST, the nearest neighbours training strategy implemented with FAST has been evaluated on a well known dataset from literature, showing promising results. NNT sports steeper (accuracy vs. time) learning curves compared to sequential training, which is an interesting result alone as all the the usual properties of model consistency and completeness of communication are simply dropped. Compared to state-of-the-art implementations, however, this approach still leaves room for improvement, but results for larger batches are encouraging and definitely supporting our initial guess. In particular it is appears to be less sensitive to hyper-parametrisation than mainstream approaches.

The topic of distributed deep learning has demonstrated to be particularly hard because of the profound entanglement that characterises performance aspects and domain specific aspects. No matter how good the attempts are, it is very hard to completely decouple the two aspects at the level of distributed SGD. The presented contributions move a resolute step in this direction by providing a methodology to improve the Time-To-Accuracy that is also less sensitive to hyper-parametrisation and fine tuning. While the use case discussed in this dissertation did not show fully conclusive evidences, the deployment of the presented strategy on large infrastructures with fairly small per-node batch sizes is expected to provide significant improvements over the state of the art both in terms of raw scalability and TTA.

7.2 Future work

This dissertation tried to delve as deep as possible into a broad field, trying to achieve both relevant methodological and experimental results, but research on this topic is far from over.

The first gap to be filled is theoretical: a clear formulation of the convergence of NNT is lacking. Previous literature can be of help, but deep-learning oriented works discussing incomplete communication strategies only refer to exchanging weights, instead of gradients. A formal relationship between the two approaches may be hiding in plain sight, but it has not been investigated by this work. Some interesting contribution may come from general-purpose distributed optimisation, where a large corpus of literature considered the case of incomplete communication graphs and unreliable links. This aspect is definitely worth of deeper study beyond the scope of this dissertation.

With respect to the NNT strategy itself, work is in progress to investigate its subtleties (e.g. the degree of asynchrony in communications between neighbouring workers), so that convergence performance is not affected by implementation issues. Also, a deeper understanding of its response to hyper-parametrisation is due, that should be coupled with the

theoretical investigation mentioned above.
For a more fair evaluation with respect to synchronous approaches, the first step will be to apply it to a larger problem, like ImagNet: this would allow for larger batches on many workers, without depleting an epoch after only a couple of gradient updates, as happens with 16 workers and, for instance, a batch of 2048 (i.e. in this case only two updates per worker can be carried out before ending an epoch, which counts 50,000 samples). The deployment on larger, and possibly GPU-equipped infrastructures, should also highlight the strengths of NNT in a comparison to synchronous methods that suffer large scales and small granularities.

The FAST framework also deserves some further effort to bring it more resolutely outside the "research prototype" realm. In particular the way the training class should be written by the end-user may be improved, creating a proper interface, instead of just a specification. This would allow, in principle, to link python objects from it, allowing full python training with FAST under the hood. Performance-wise, there is the need of a review of the thread-safety of some GAM inner mechanisms, that would allow for a more asynchronous behaviour of the worker node. Finally, providing interfaces for TensorFlow and PyTorch should be the next step towards adoption of this tool outside the author's research group.

Bibliography

[1] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis", *CoRR*, vol. abs/1802.09941, 2018.

[2] P. Viviani, M. Drocco, and M. Aldinucci, "Pushing the boundaries of parallel deep learning - A practical approach", *CoRR*, vol. abs/1806.09528, 2018.

[3] P. Viviani, M. Drocco, D. Baccega, I. Colonnelli, and M. Aldinucci, "Deep learning at scale", in *Proc. of 27th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP)*, Pavia, Italy: IEEE, 2019.

[4] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amatya, "GossipGraD: Scalable Deep Learning using Gossip Communication based Asynchronous Gradient Descent", *CoRR*, vol. abs/1803.05880, 2018.

[5] MACH project consortium, *MAssive Calculations on Hybrid systems*, 2013. [Online]. Available: https://itea3.org/project/mach.html (visited on 02/17/2019).

[6] Fortissimo project consortium, *Fortissimo2*, 2015. [Online]. Available: https://www.fortissimo-project.eu/about/fortissimo-2 (visited on 02/17/2019).

[7] CloudFlow project consortium, *CloudFlow*, 2013. [Online]. Available: https://eu-cloudflow.eu/ (visited on 02/17/2019).

[8] KU Leuven, *Blockchain for misc Service Security*, 2017. [Online]. Available: https://distrinet.cs.kuleuven.be/research/projects/BoSS (visited on 02/17/2019).

[9] M. Aldinucci, S. Rabellino, M. Pironti, *et al.*, "Hpc4ai, an ai-on-demand federated platform endeavour", in *ACM Computing Frontiers*, Ischia, Italy, May 2018.

[10] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino, "Occam: A flexible, multi-purpose and extendable hpc cluster", in *Journal of Physics: Conf. Series (CHEP 2016)*, vol. 898, San Francisco, USA, 2017, p. 082 039.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", En, *Nature*, vol. 521, no. 7553, p. 436, May 2015. (visited on 01/24/2018).

[12] J. Schmidhuber, "Deep learning in neural networks: An overview", *Neural Networks*, vol. 61, no. Supplement C, pp. 85–117, Jan. 2015. (visited on 12/14/2017).

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[14] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning Hierarchical Features for Scene Labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915–1929, Aug. 2013.

[15] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler, "Joint training of a convolutional network and a graphical model for human pose estimation", in *Advances in Neural Information Processing Systems*, 2014, pp. 1799–1807.

[16] G. Hinton, L. Deng, D. Yu, *et al.*, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[17] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for LVCSR", in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 2013, pp. 8614–8618.

[18] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks", in *Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.

[19] G. Lample, M. Ott, A. Conneau, L. Denoyer, and M. Ranzato, "Phrase-Based & Neural Unsupervised Machine Translation", *CoRR*, vol. abs/1804.07755, 2018.

[20] M. K. K. Leung, H. Y. Xiong, L. J. Lee, and B. J. Frey, "Deep learning of the tissue-regulated splicing code", eng, *Bioinformatics (Oxford, England)*, vol. 30, no. 12, pp. i121–129, Jun. 2014.

[21] H. Y. Xiong, B. Alipanahi, L. J. Lee, *et al.*, "The human splicing code reveals new insights into the genetic determinants of disease", en, *Science*, vol. 347, no. 6218, p. 1 254 806, Jan. 2015. (visited on 03/16/2019).

[22] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[23] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning From Data*. AMLBook, 2012.

[24] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain", *Psychological Review*, pp. 65–386, 1958.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[26] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis", en, in *System Modeling and Optimization*, ser. Lecture Notes in Control and Information Sciences, Springer, Berlin, Heidelberg, 1982, pp. 762–770. (visited on 04/19/2018).

[27] Y. LeCun, "A theoretical framework for back-propagation", English (US), in *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, D. Touretzky, G. Hinton, and T. Sejnowski, Eds., Morgan Kaufmann, 1988, pp. 21–28.

[28] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp", en, in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1998, pp. 9–50. (visited on 04/19/2018).

[29] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies", 2001.

[30] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks", in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, PMLR, Apr. 2011, pp. 315–323.

[31] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The Loss Surfaces of Multilayer Networks", in *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, G. Lebanon and S. V. N. Vishwanathan, Eds., ser. Proceedings of Machine Learning Research, vol. 38, San Diego, California, USA: PMLR, May 2015, pp. 192–204.

[32] M. Marchesi, G. Orlandi, F. Piazza, L. Pollonara, and A. Uncini, "Multi-layer perceptrons with discrete weights", in *1990 IJCNN International Joint Conference on Neural Networks*, Jun. 1990, 623–630 vol.2.

[33] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks", in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Springer International Publishing, 2016, pp. 525–542.

[34] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[35] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning", *Neural Networks*, vol. 16, no. 10, pp. 1429–1451, Dec. 2003. (visited on 12/14/2017).

[36] H. Robbins and S. Monro, "A Stochastic Approximation Method", EN, *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, Sep. 1951. (visited on 06/27/2018).

[37] Y. Bengio, "Practical Recommendations for Gradient-Based Training of Deep Architectures", en, in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2012, pp. 437–478. (visited on 04/06/2018).

[38] L. Bottou and O. Bousquet, "The Tradeoffs of Large Scale Learning", in *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., vol. 20, NIPS Foundation, 2008, pp. 161–168.

[39] L. Bottou and Y. LeCun, "Large Scale Online Learning", in *Advances in Neural Information Processing Systems 16*, S. Thrun, L. K. Saul, and B. Schölkopf, Eds., MIT Press, 2004, pp. 217–224. (visited on 04/20/2018).

[40] M. Moller, "Supervised learning on large redundant training sets", in *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, Aug. 1992, pp. 79–89.

[41] G. B. Orr, "Removing Noise in On-Line Search using Adaptive Batch Sizes", in *Advances in Neural Information Processing Systems 9*, M. C. Mozer, M. I. Jordan, and T. Petsche, Eds., MIT Press, 1997, pp. 232–238. (visited on 04/20/2018).

[42] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale Deep Unsupervised Learning Using Graphics Processors", in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09, Montreal, Quebec, Canada: ACM, 2009, pp. 873–880.

[43] J. Bergstra, F. Bastien, O. Breuleux, *et al.*, "Theano: Deep Learning on GPUs with Python", in *Big Learn Workshop, NIPS'11*, 2011.

[44] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning", *CoRR*, vol. abs/1410.0759, 2014.

[45] S. Ruder, "An overview of gradient descent optimization algorithms", *CoRR*, vol. abs/1609.04747, 2016.

[46] N. Qian, "On the momentum term in gradient descent learning algorithms", *Neural Networks*, vol. 12, no. 1, pp. 145–151, Jan. 1999. (visited on 04/20/2018).

[47] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *CoRR*, vol. abs/1412.6980, 2014.

[48] S. Ma, R. Bassily, and M. Belkin, "The Power of Interpolation: Understanding the Effectiveness of SGD in Modern Over-parametrized Learning", *CoRR*, vol. abs/1712.06559, 2017.

[49] D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks", *ArXiv e-prints*, 2018.

[50] P. Goyal, P. Dollár, R. B. Girshick, *et al.*, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", *CoRR*, vol. abs/1706.02677, 2017.

[51] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD Batch Size to 32K for ImageNet Training", *CoRR*, vol. abs/1708.03888, 2017.

[52] M. Cho, U. Finkler, S. Kumar, D. S. Kung, V. Saxena, and D. Sreedhar, "PowerAI DDL", *CoRR*, vol. abs/1708.02188, 2017.

[53] E. Hoffer, T. Ben-Nun, I. Hubara, N. Giladi, T. Hoefler, and D. Soudry, "Augment your batch: Better training with larger batches", *CoRR*, vol. abs/1901.09335, 2019.

[54] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes", *CoRR*, vol. abs/1711.04325, 2017.

[55] S. Ghadimi and G. Lan, "Stochastic First- and Zeroth-order Methods for Nonconvex Stochastic Programming", *CoRR*, vol. abs/1309.5549, 2013.

[56] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization", in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15, MIT Press, 2015, pp. 2737–2745.

[57] X. Zhang, J. Liu, and Z. Zhu, "Taming Convergence for Asynchronous Stochastic Gradient Descent with Unbounded Delay in Non-Convex Learning", *CoRR*, vol. abs/1805.09470, 2018.

[58] D. Bertsekas, *Nonlinear Programming*, ser. Athena Scientific Optimization and Computation Series. Athena Scientific, 2016.

[59] R. E. Wengert, "A Simple Automatic Derivative Evaluation Program", *Commun. ACM*, vol. 7, no. 8, pp. 463–464, Aug. 1964.

[60] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic Differentiation in Machine Learning: A Survey", *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018.

[61] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding", *arXiv preprint arXiv:1408.5093*, 2014.

[62] M. Abadi, P. Barham, J. Chen, *et al.*, "TensorFlow: A system for large-scale machine learning", *CoRR*, vol. abs/1605.08695, 2016.

[63] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in PyTorch", in *NIPS-W*, 2017.

[64] T. Chen, M. Li, Y. Li, *et al.*, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems", *CoRR*, vol. abs/1512.01274, 2015.

[65] D. Yu, A. Eversole, M. Seltzer, *et al.*, "An Introduction to Computational Networks and the Computational Network Toolkit", *Microsoft Research*, Aug. 2014. (visited on 12/15/2017).

[66] G. Kahn, "The Semantics of Simple Language for Parallel Programming", in *IFIP Congress*, 1974, pp. 471–475.

[67] F. Chollet *et al.*, "Keras", 2015.

[68] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989.

[69] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[70] NVIDIA Corporation, *CUDA Toolkit Documentation*. [Online]. Available: http://docs.nvidia.com/cuda/eula/index.html (visited on 02/17/2019).

[71] Apache MxNet, *Deep Learning Programming Style*. [Online]. Available: https://mxnet.incubator.apache.org/versions/master/architecture/program_model.html (visited on 03/20/2019).

[72] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools", *CoRR*, vol. abs/1608.07249, 2016.

[73] Intel Corporation, *Intel Math Kernel Library for Deep Neural Networks*. [Online]. Available: https://github.com/intel/mkl-dnn (visited on 03/20/2019).

[74] Google, *Google edge tpu*. [Online]. Available: https://cloud.google.com/edge-tpu/ (visited on 03/20/2019).

[75] Intel Corporation, *Intel Neural Compute Stick*. [Online]. Available: https://software.intel.com/en-us/movidius-ncs (visited on 03/20/2019).

[76] Google, *Google TensorFlow Lite*. [Online]. Available: https://www.tensorflow.org/lite (visited on 03/20/2019).

[77] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.

[78] *Intel Math Kernel Library: Reference Manual*, Intel Corporation, 2009.

[79] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, ACM, 2013, 25:1–25:12.

[80] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing", in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Suvisoft, Oct. 2006.

[81] M. Mathieu, M. Henaff, and Y. LeCun, "Fast Training of Convolutional Networks through FFTs", *CoRR*, vol. abs/1312.5851, 2013.

[82] A. Lavin, "Fast Algorithms for Convolutional Neural Networks", *CoRR*, vol. abs/1509.09308, 2015.

[83] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, IEEE Press, 2016, 54:1–54:12.

[84] N. Vasilache, O. Zinenko, T. Theodoridis, *et al.*, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions", *CoRR*, vol. abs/1802.04730, 2018.

[85] J. Appleyard, T. Kociský, and P. Blunsom, "Optimizing Performance of Recurrent Neural Networks on GPUs", *CoRR*, vol. abs/1604.01946, 2016.

[86] J. Dean, G. S. Corrado, R. Monga, *et al.*, "Large Scale Distributed Deep Networks", in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, USA: Curran Associates Inc., 2012, pp. 1223–1231.

[87] J. Ngiam, Z. Chen, D. Chia, P. W. Koh, Q. V. Le, and A. Y. Ng, "Tiled convolutional neural networks", in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds., Curran Associates, Inc., 2010, pp. 1279–1287. (visited on 04/19/2018).

[88] M. Abadi, A. Agarwal, P. Barham, *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems", *CoRR*, vol. abs/1603.04467, 2016.

[89] L. Deng, D. Yu, and J. Platt, "Scalable stacking and learning for building deep architectures", in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2012, pp. 2133–2136.

[90] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide, "Pipelined Back-Propagation for Context-Dependent Deep Neural Networks", en-US, *Microsoft Research*, Sep. 2012. (visited on 02/09/2018).

[91] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

[92] X. Zhang, M. Mckenna, J. P. Mesirov, and D. L. Waltz, "Advances in Neural Information Processing Systems 2", in, D. S. Touretzky, Ed., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. An Efficient Implementation of the Back-propagation Algorithm on the Connection Machine CM-2, pp. 801–809.

[93] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks", *CoRR*, vol. abs/1603.05027, 2016.

[94] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", Tech. Rep., 2009.

[95] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", *CoRR*, vol. abs/1502.03167, 2015.

[96] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: Closing the generalization gap in large batch training of neural networks", *CoRR*, vol. abs/1705.08741, 2017.

[97] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks", in *Advances in Neural Information Processing Systems*, 2016, pp. 901–909.

[98] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit", *CoRR*, vol. abs/1704.04760, 2017.

[99] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima", *CoRR*, vol. abs/1609.04836, 2016.

[100] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware Async-SGD for Distributed Deep Learning", *CoRR*, vol. abs/1511.05950, 2015.

[101] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, "Asynchrony begets Momentum, with an Application to Deep Learning", *CoRR*, vol. abs/1605.09774, 2016.

[102] Message Passing Forum, "Mpi: A message-passing interface standard", Knoxville, TN, USA, Tech. Rep., 2012.

[103] F. Niu, B. Recht, C. Ré, and S. J. Wright, "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent", *CoRR*, vol. abs/1106.5730, 2011.

[104] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, 2014, pp. 571–582.

[105] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang, "GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training", *CoRR*, vol. abs/1312.6186, 2013.

[106] N. Strom, "Scalable Distributed DNN Training Using Commodity GPU Cloud Computing", in *INTERSPEECH 2015 16th Annual Conference of the International Speech Communication Association*, Dresden, Sep. 2015.

[107] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z. Ma, and T.-Y. Liu, "Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning", *CoRR*, vol. abs/1609.08326, 2016.

[108] J. Keuper and F.-J. Pfreundt, "Asynchronous Parallel Stochastic Gradient Descent - A Numeric Core for Scalable Distributed Machine Learning Algorithms", *CoRR*, vol. abs/1505.04956, 2015.

[109] J. Hermans, G. Spanakis, and R. Möckel, "Accumulated Gradient Normalization", *CoRR*, vol. abs/1710.02368, 2017.

[110] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent", *CoRR*, vol. abs/1705.09056, 2017.

[111] C. A. Coleman, D. Narayanan, D. Kang, *et al.*, "DAWNBench : An End-to-End Deep Learning Benchmark and Competition", in *NIPS ML Systems Workshop*, 2017.

[112] S. DAWN, *DAWNBench: An end-to-end deep learning benchmark and competition*. [Online]. Available: https://dawn.cs.stanford.edu/benchmark/index.html (visited on 03/20/2019).

[113] J. Hermans, "On Scalable Deep Learning and Parallelizing Gradient Descent", CERN Document Server, Tech. Rep., Aug. 2017. (visited on 02/07/2018).

[114] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous Decentralized Parallel Stochastic Gradient Descent", *ArXiv e-prints*, vol. 1710, arXiv:1710.06952, Oct. 2017. (visited on 04/26/2018).

[115] B. Polyak and A. Juditsky, "Acceleration of Stochastic Approximation by Averaging", *SIAM Journal on Control and Optimization*, vol. 30, no. 4, pp. 838–855, Jul. 1992. (visited on 05/04/2018).

[116] D. Povey, X. Zhang, and S. Khudanpur, "Parallel training of Deep Neural Networks with Natural Gradient and Parameter Averaging", *CoRR*, vol. abs/1410.7455, 2014.

[117] S. Zhang, A. Choromanska, and Y. LeCun, "Deep Learning with Elastic Averaging SGD", in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15, Cambridge, MA, USA: MIT Press, 2015, pp. 685–693.

[118] S. Lee, S. Purushwalkam, M. Cogswell, D. J. Crandall, and D. Batra, "Why M Heads are Better than One: Training a Diverse Ensemble of Deep Networks", *CoRR*, vol. abs/1511.06314, 2015.

[119] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network", *CoRR*, vol. abs/1503.02531, 2015.

[120] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training", *CoRR*, vol. abs/1712.01887, 2017.

[121] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs", *Microsoft Research*, Sep. 2014. (visited on 12/14/2017).

[122] MLPerf, *MLPerf: A broad ml benchmark suite for measuring performance of ml software frameworks, ml hardware accelerators, and ml cloud platforms*. [Online]. Available: https://mlperf.org/ (visited on 03/26/2019).

[123] NVIDIA Corporation, *NVIDIA data center products*. [Online]. Available: https://www.nvidia.com/en-us/data-center/ (visited on 03/20/2019).

[124] K. Lee, *Introducing Big Basin: Our next-generation AI hardware*. [Online]. Available: https://code.fb.com/data-center-engineering/introducing-big-basin-our-next-generation-ai-hardware/ (visited on 03/29/2019).

[125] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon, *TOP500 list - november 2018*. [Online]. Available: https://www.top500.org/list/2018/11/ (visited on 03/20/2019).

[126] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, *et al.*, "The Design, Deployment, and Evaluation of the CORAL Pre-exascale Systems", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18, IEEE Press, 2018, 52:1–52:12.

[127] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow", *CoRR*, vol. abs/1802.05799, 2018.

[128] Google, *gRPC: A high-performance, open-source universal rpc framework*. [Online]. Available: https://grpc.io/about/ (visited on 03/29/2019).

[129] Facebook, *Collective communications library with various primitives for multi-machine training*. [Online]. Available: https://github.com/facebookincubator/gloo (visited on 03/29/2019).

[130] NVIDIA Corporation, *NVIDIA gpudirect*. [Online]. Available: https://developer.nvidia.com/gpudirect (visited on 03/29/2019).

[131] F. Akgul, *ZeroMQ*. Packt Publishing, 2013.

[132] NVIDIA Corporation, *NVIDIA depp learning sdk documentation*. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/index.html (visited on 03/29/2019).

[133] T. Hoefler, *Twelve ways to fool the masses when reporting performance of deep learning workloads*. [Online]. Available: http://htor.inf.ethz.ch/blog/index.php/2018/11/08/twelve-ways-to-fool-the-masses-when-reporting-performance-of-deep-learning-workloads/ (visited on 03/26/2019).

[134] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database", in *CVPR09*, 2009.

[135] T. Lin, S. U. Stich, and M. Jaggi, "Don't Use Large Mini-Batches, Use Local SGD", *CoRR*, vol. abs/1808.07217, 2018.

[136] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters", *SIGPLAN Not.*, vol. 52, no. 8, pp. 193–205, Jan. 2017.

[137] M. Li, D. G. Andersen, A. Smola, and K. Yu, "Communication Efficient Distributed Machine Learning with the Parameter Server", in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'14, MIT Press, 2014, pp. 19–27.

[138] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance & Precision", *CoRR*, vol. abs/1803.04014, 2018.

[139] S. S. Ram, A. Nedic, and V. V. Veeravalli, "Asynchronous gossip algorithms for stochastic optimization", in *2009 International Conference on Game Theory for Networks*, May 2009, pp. 80–81.

[140] T. Hoefler, A. Barak, A. Shiloh, and Z. Drezner, "Corrected Gossip Algorithms for Fast Reliable Broadcast on Unreliable Systems", in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 357–366.

[141] P. H. Jin, Q. Yuan, F. N. Iandola, and K. Keutzer, "How to scale distributed deep learning?", *CoRR*, vol. abs/1611.04581, 2016.

[142] M. Blot, D. Picard, M. Cord, and N. Thome, "Gossip training for deep learning", *CoRR*, vol. abs/1611.09726, 2016.

[143] A. Demers, D. Greene, C. Hauser, *et al.*, "Epidemic Algorithms for Replicated Database Maintenance", in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87, ACM, 1987, pp. 1–12.

[144] A. G. Dimakis, S. Kar, J. M. F. Moura, M. G. Rabbat, and A. Scaglione, "Gossip Algorithms for Distributed Signal Processing", *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1847–1864, Nov. 2010.

[145] L. Cannelli, F. Facchinei, V. Kungurtsev, and G. Scutari, "Asynchronous Parallel Algorithms for Nonconvex Big-Data Optimization: Model and Convergence", *CoRR*, vol. abs/1607.04818, 2016.

[146] T. Tatarenko and B. Touri, "Non-Convex Distributed Optimization", *IEEE Transactions on Automatic Control*, vol. 62, no. 8, pp. 3744–3757, Aug. 2017.

[147] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency", in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug. 2015, pp. 34–39.

[148] M. Drocco, "Parallel programming with global asynchronous memory: Models, C++ APIs and implementations", PhD thesis, Computer Science Department, University of Torino, Oct. 2017.

[149] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd. Addison-Wesley Professional, 2012.

[150] OpenFabrics Alliance, *Libfabric OpenFabrics website*. [Online]. Available: https://ofiwg.github.io/libfabric/ (visited on 03/29/2019).

[151] M. Danelutto and M. Torquati, "Structured parallel programming with "core" fastflow", in *Central European Functional Programming School*, ser. LNCS, V. Zsók, Z. Horváth, and L. Csató, Eds., vol. 8606, Springer, 2015, pp. 29–75.

[152] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann, "Parallel programming environment for OpenMP", *Scientific Programming*, vol. 9, pp. 143–161, 2001.

[153] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: High-level and efficient streaming on multi-core", in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds., Wiley, 2017, ch. 13.

[154] Flink, *Apache Flink website*, https://flink.apache.org/. (visited on 03/07/2016).

[155] M. Aldinucci and M. Torquati, *Fastflow website*, http://mc-fastflow.sourceforge.net/, 2009.

[156] M. Danelutto and M. Torquati, "Loop parallelism: A new skeleton perspective on data parallel patterns", in *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, M. Aldinucci, D. D'Agostino, and P. Kilpatrick, Eds., Torino, Italy: IEEE, 2014.

[157] C. Pheatt, "Intel® threading building blocks", *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008.

[158] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on multi-core with fastflow", in *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, M. Danelutto, T. Gross, and J. Bourgeois, Eds., Pisa, Italy: IEEE, Feb. 2010, pp. 195–199.

[159] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", *CoRR*, vol. abs/1512.03385, 2015.

[160] A. Krizhevsky, *The CIFAR-10 dataset*. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html (visited on 04/09/2019).

Glossary

AI Artificial Intelligence

ANN Artificial Neural Network

API Application Programming Interface

ASIC Application-specific integrated circuit

BLAS Basic Linear Algebra Subprograms

BN Batch Normalisation

DAG Direct Acyclic Graph

DL Deep Learning

DNN Deep Neural Network

DRF Data Race Free

DSeL Domain-Specific Embedded Language

EASGD Elastic Averaging SGD

EC European Commission

FAST Flexible Asynchronous Scalable Training

FFT Fast Fourier Transform

FPGA Field Programmable Gate Array

GAM Global Asynchronous Memory

GAS Global Address Space

GEMM General Matrix Multiplication

GPU Graphics Processing Unit

HPC High Performance Computing

i.i.d. Independent and identically distributed random variables

IPoIb IP over InfiniBand

ML Machine Learning

MPI Message Passing Interface

MPMC Multiple Producer Multiple Consumer

MPSC Multiple Producer Single Consumer

MSE Mean Squared Error

NNT Nearest-Neighbours Training

OPS Operations per Second

PS Parameter Server

RMA Remote Memory Access

SGD Stochastic Gradient Descent

SPMC Single Producer Multiple Consumer

SPMD Single Program Multiple Data

SPSC Single Producer Single Consumer

ssh Secure Shell

TTA Time-To-Accuracy