Deep Learning at Scale

P. Viviani, M. Drocco, D. Baccega, and M. Aldinucci. Deep learning at scale. In Proc. of 27th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP), Pavia, Italy, 2019. IEEE.

Deep Learning at Scale

Paolo Viviani
Noesis Solutions NV, Belgium
Computer Science Department
University of Torino, Italy
paolo.viviani@noesissolutions.com

Maurizio Drocco
Pacific Northwest National Laboratory
Richland, WA, USA
maurizio.drocco@pnnl.gov

Daniele Baccega
Iacopo Colonnelli
Marco Aldinucci
Computer Science Department
University of Torino, Italy
{baccega,aldinuc}@di.unito.it
iacopo.colonnelli@unito.it

***Abstract*—This work presents a novel approach to distributed training of deep neural networks (DNNs) that aims to overcome the issues related to mainstream approaches to data parallel training. Established techniques for data parallel training are discussed from both a parallel computing and deep learning perspective, then a different approach is presented that is meant to allow DNN training to scale while retaining good convergence properties. Moreover, an experimental implementation is presented as well as some preliminary results.**

I. Introduction

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 reliable and efficient enterprise solutions. This aim involves several steps to be taken, and this work advocates the need to push the current state of the art in parallel training in order to achieve: 1) faster end-to-end training for large production datasets; 2) distributed training on the edge, namely on a number of heterogeneous, low-power, and loosely-coupled devices (i.e. for privacy constraints); 3) training code that can be redistributed, possibly in form of binaries (i.e. to train models at customer's premises without exposing sensitive Python code). To practically implement this vision, a number of advancements are required and this work represents a first step towards:

1) a better theoretical understanding of the different strategies of data parallelism in deep neural networks;
2) a consistent way to compare different deployments and strategies.

Issues related to point 1 will be presented, addressing some of them and discussing how it is possible to push further the model training efficiency; moreover, this paper will propose a design for a programming framework that would address point 2.

Sec. II presents a survey of parallel techniques for deep neural network training, the next section provides a further exploration of some theoretical highlights that can be exploited to improve training scalability. Sec. IV presents a design for an upcoming data parallel training framework and, finally, sec. V provides an outlook of the potential impact of the presented results as well as the opportunities.

II. Background

Performance issues in deep neural networks (DNNs) have been extensively investigated from many point of views: in particular it is possible to clearly discriminate between the training stage and the inference stage. The latter is usually characterised by smaller computational workloads that are, however, highly constrained by time, memory, and power consumption due to the deployment on portable devices that need predictions almost in real-time. This paper is focused on the former stage of deep neural network training. A comprehensive survey of the state of the art for parallel DNN training has been presented by Ben-Nun and Hoefler [1], it is among the goals of this paper to review a subset of the relevant work, providing a more critical insight.

To further define the research scope of this work, it is useful to highlight the main categorization of parallel training: namely *data parallelism* vs. *model parallelism*. Data parallelism focuses on distributing partitions of training data among workers, that cooperate to train replicas of the same model; model parallelism involves the partition of the model computation graph among different workers, that train different parts of the same model instance. While the latter (including layer pipelining) has been proved to be an efficient way to improve the performance of DNN training [2–5] it can be argued that its capacity to scale beyond the single machine is limited by the higher frequency of communications 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, this aspect is not explored in this work, but it can be quickly added to the data parallel strategies discussed later as a further optimization, without impacting the following discussion.

A. Mathematical notation

Despite the many attempts to implement different optimizations strategies [6], back-propagation [7–10] with some flavour of gradient descent [11] is still the most popular way to train deep neural networks, mostly due to its high efficiency on modern architectures like GPUs [12]. This section presents some useful notation for gradient descent-based neural network training.

For the rest of this section it will be considered that a dataset $X = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$, is used to train a neural

network represented here as a collection of parameters (*weights*) $\mathbf{w} = \{w_1, \ldots, w_m\}$. Hereafter, neither the network type and topology (i.e. convolutional, recurrent, number of hidden layers) nor the input dimensionality and shape are considered relevant, as the formalism is generally applied to all of them. Mini-batch gradient descent [10, 13, 14] has quickly became the standard, combining the faster convergence of Stochastic (*on-line*) Gradient Descent (SGD) [15–17], with the more efficient computation of *batch* gradient descent. The optimization step for training can be expressed as the following weight update, computed with respect to a mini-batch $X_{(i,n_b)} = \{\mathbf{x_i}, \ldots, \mathbf{x_{i+n_b-1}}\}$:

$$w_k(t+1) = w_k(t) - \frac{\eta}{n_b} \sum_{j=i}^{i+n_b-1} \frac{\partial L\left(\mathbf{w}(t), \mathbf{x_j}\right)}{\partial w_k} \tag{1}$$

where t represents the current gradient descent iteration (*step*), η is the so-called *learning rate* that defines the size of the step to be taken in the direction of the steepest descent, and $\partial L(\mathbf{w},\mathbf{x_j})/\partial w_k$is the partial derivative of the loss function of the neural network with respect to the weight w_k, when calculated on the training sample $\mathbf{x_j}$. The partial derivative is averaged over all the samples belonging to a given subset (the *mini-batch*) of the training dataset of size n_b. It is useful to recall the definition of all the versions of gradient descent by means of the value of n_b:

- $n_b = 1$, stochastic gradient descent
- $1 < n_b \ll n$, mini-batch gradient descent
- $n_b = n$, batch gradient descent

Note that batch averaging, as opposite of just summing, has a non-trivial impact on the convergence of the training [11]. It is also useful to define the gradient for all the weights of the model as following

$$\nabla L(\mathbf{w}, \mathbf{x_j}) = \left(\frac{\partial L\left(\mathbf{w}, \mathbf{x_j}\right)}{\partial w_1}, \ldots, \frac{\partial L\left(\mathbf{w}, \mathbf{x_j}\right)}{\partial w_m} \right) \tag{2}$$

this represents the direction of steepest slope of the loss surface calculated with respect to $\mathbf{x_j}$ in the parameter's space ($L : \mathbb{R}^m \to \mathbb{R}$); it is trivial to obtain the gradient and the step with respect to the whole mini-batch $X_{(i,n_b)}$ as

$$\begin{aligned} \frac{1}{n_b} \sum_{j=i}^{i+n_b-1} \nabla L(\mathbf{w}, \mathbf{x_j}) &\stackrel{\text{def}}{=} \Delta L(\mathbf{w}, X_{(i,n_b)}) \\ \mathbf{w}(t+1) &= \mathbf{w}(t) - \eta \Delta L(\mathbf{w}, X_{(i,n_b)}) \end{aligned} \tag{3}$$

Equation 1 represents the simplest form of mini-batch gradient descent. Several algorithms have been developed to improve the convergence rate of DNN training, a good review of them can be found in literature [18, 19]. 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* [20]); 3) defining a different learning rate for each weight $\eta(t) \to \eta(t, w_k)$ (e.g. ADAM [21]). These points have an impact on parallel training implementation that will be discussed later.

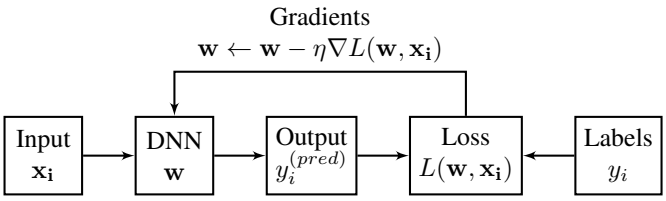

Fig. 1. Back-propagation diagram for on-line gradient descent.

B. Training parallelism

When considering the whole *feed-forward/back-propagation* [7] training process, it is important to remark that it is, to some extent, intrinsically sequential. Figure 1 and equation (1) 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. In principle this 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 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 1, 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 2 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 [22, 23]: a certain amount of noise and randomness associated to the gradient descent can be beneficial to the training outcome in terms of generalization. The next subsections will describe how this behaviour can be exploited to introduce some degree of parallelism into the training process.

1) Synchronous parallelism: As stated before, mini-batch gradient descent combines the best of both on-line and batch training; in particular, the fact that it can be expressed as a chain of matrix-matrix multiplication (GEMMs) [10] that allow for a very efficient implementation on multicore CPUs and GPUs [11], enabled a wide adoption of deep learning due to the better training feasibility. From the parallel computing point of view, mini-batches represent the most elementary approach to what is called *synchronous data parallel* training, as a global synchronization happens at the end of each mini-batch.

The amount of available parallelism depends on the size of the mini-batches, that in turn affects the convergence of the training. Apart from avoiding the extreme cases of on-line and

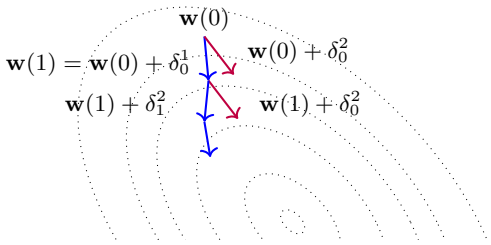

Fig. 2. Gradient descent in $\mathbf{w}$ space. $\delta_i^j = -\eta \nabla L(\mathbf{w}(i), \mathbf{x_j})$ represents the gradient calculated on the weights updated up to step i, based on sample (or mini-batch) $\mathbf{x_j}$. Therefore, the red update based on δ_0^2 is outdated with respect to $\mathbf{w}(1)$,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

batch gradient descent, the choice of the right mini-batch size is not trivial, and there is interaction with other hyper-parameters, like the learning rate, as widely discussed in literature [23–28] often concerning the *linear scaling* of η. In principle, larger mini-batches allow to process more samples per unit of time, while the convergence can be hindered if the size is too large.

Mini-batch parallelism is usually exploited by means of parallel GEMMs on suitable architectures [12, 29, 30]. However, recent works [31–33] 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-reduce* communication pattern to perform the global synchronization. Ignoring the communication bottlenecks that will be discussed in Sec. II-B4, it can be argued that this approach is problem-specific and can not always be pushed as far as [31] suggests. In fact, smaller mini-batches (~ 32) provide usually better generalization performance [10, 23, 28]. This induces a granularity problem: smaller batches can be effectively computed only if the size of the network and the complexity of the individual data sample (e.g. large RGB picture vs. small array of numerical data) are large enough to saturate the given platform even with only few samples being processed concurrently. This issue can heavily affect the capability of certain models to scale on large distributed clusters. A further issue is the so-called *batch normalization* (BN) [34], that introduces data dependencies between different samples among the same mini-batch, such that a full synchronization is required at each invocation of BN.

Parallelism at mini-batch level proved to be effective at node-level when implemented on GPUs, multi-core CPUs or other dedicate hardware (e.g. Google TPUs [35]); still, the scalability of its extension to distributed memory architectures is subject to a suitable problem granularity, that is far from being granted apart from specific problems.

Further parallel implementation of DNN training usually take mini-batch parallelism for granted, at least at node-level, considering mini-batches as atomic entities for which the data dependency defined in Figure 1 exists. From this point of view, mini-batches can be considered the only truly synchronous kind of parallel training: while other strategies that will be presented in the next sections might involve synchronizations at certain stages, they necessarily relax the dependency between subsequent mini-batches. Indeed, in the rest of this paper mini-batches will be considered as atomic entities, that cannot be further divided. Synchronous *distributed* parallelism at mini-batch level will also be addresses as *large mini-batch* parallelism.

2) Asynchronous parallelism: The success of momentum 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 can be also related a posteriori to the usage of mini-batches, as highlighted by Masters and Luschi [28], this notion 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 synchronization at the mini-batch level.

There is a common categorization [1] between centralized and de-centralized implementations, as well as based the degree of model consistency achieved. The latter is a property of a given implementation that measures how different are the weights of each model replica at a certain instant of time, while the former categorization 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. Sec. III will further discuss these classifications. Early notable implementations of asynchronous parallel gradient descent are HOGWILD! [36] and its deep learning-focused derivatives like Downpour SGD [2, 37]; followed by some other significant works [38–44]. Apart from the *DistBelief* [2] and Project Adam [37] 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 [45], at the time of writing, are still relatively small-scale, synchronous implementations.

While this review is far from being conclusive, it is possible to suggest 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 naive SGD, while the state of the art is moving to more sophisticated algorithms like ADAM [21]. Some effort in this directions exists [46], as well as a prominent theoretical work [47] 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 generalization effort that might help to understand the validity of the methodology. In this sense a relevant analysis has been performed by Lian et. al [48]: 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, lay the ground for a research that aims to fill the gap between sparse experimentation and mathematical modelling of convergence rates.

3) Other approaches: Synchronous and asynchronous SGD are not the only ways to exploit concurrency in DNN training. *Model averaging* [49–51] 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* [52, 53] performs the whole training on different model instances, 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, model averaging is strictly related to the techniques presented in Sec. II-B1 and II-B2 and, while it is out of the scope of this paper to formally draw the connection, it will be investigated in the near future.

4) Further parallelism issues: As said in Sec. II-B1, 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 ($\Delta L(\mathbf{w}, X_{(i,n_b)})$ for a state of the art DNN easily reaches a few hundred MB [54]. 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 [39, 55]; 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 [39]. A more recent work [54] also includes momentum in the discussion and presents interesting results. Also in this case, apart from the 1-bit quantization provided by Microsoft CNTK [56], 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 optimization 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: this paper highlights the need of a general purpose framework that provides the tools to experiment with existing techniques at different levels (i.e. asyncronous vs. synchronous, different communication patterns, quantization, etc.), as well as defining and testing new ones. Sec. IV will discuss the requirements for such framework.

III. Theoretical discussion

Assuming that using very large mini-batches is not suitable for any application, end-to-end training performance can be improved at two distinct levels:

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 optimization strategy that exploits parallelism better than gradient descent.

Point 1 is being researched actively [57, 58] 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. In this sense it is possible to show that, despite usually being treated as different approaches, all the techniques discussed in Sec. II-B1 and II-B2 can be placed on a spectrum of communication completeness, namely the property of parallel implementation to distribute each gradient update from each worker to all the other workers, regardless of the time at which this happens. It is indeed possible to argue that the model consistency spectrum usually proposed [1], provides limited insight to understand what happens to model replicas in implementations presented in previous works. A statement can be formulated in this sense that, while being quite naïve, it is still important to understand the behaviour of model replicas

Statement 1: Assuming mini-batch SGD without momentum in a distributed setting, if all the gradient updates (communications) are delivered to all the workers, regardless of the delay, all the model replicas will be consistent.

Figure 3 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

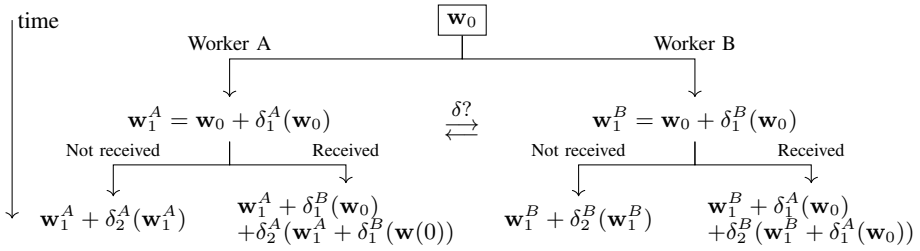

Fig. 3. Diagram of weights update between two workers. $\mathbf{w}_0$ is the common starting configuration. Assuming that all the updates that are not immediatly applied are queued somewhere, commutativity and associativity of vector sum guarantee that A and B will always be consistent once the queues are emptied.

pending updates (e.g. a global synchronization), whatever is the state of both workers before the event, their state will be consistent afterwards. Of course statement 1 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. It is also important remark that consistency is not implied at any given moment, but it is always achieved as most of the strategies proposed either accumulate all the updates in a parameter server or require a synchronization at each epoch [39] or both. This leads us to the following

Statement 2: There is no need to distinguish between centralized and de-centralized set-ups if the communication is complete; in fact there it becomes only matter of implementation to choose the approach, while the model consistency is granted.

In this view the centralized parameter server is only a way to simplify the implementation as well as inducing artificially some staleness, that can be beneficial to the training.

This discussion is relevant as our goal is to exploit more parallelism without resorting to large mini-batch training; however, workers in figure 3 always go through the *received* branch the outcome is, not surprisingly, 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 [39], making an asynchronous solution not different from a synchronous one. Of course it can be argued that not enforcing explicit synchronization can benefit scalability on very large-scale deployment, however, most of the current implementations are in fact still bound to a centralized parameter server.

It is useful at this point to define a new spectrum to discriminate between strategies:

1) Synchronous communication (large mini-batches)
2) Complete communication with bound delay (stale-synchronous [40])
3) Complete communication with unbound delay (Downpour SGD [2])
4) Incomplete topologies ([59, 60])

It is important to remark that, when applied in an homogeneous environment with high-bandwidth, low-latency interconnection (i.e. any common HPC set-up), the first three points are not significantly distinguishable in terms of training convergence, at least from the theoretical point of view. 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 than a property of this strategy, moreover the centralized approach introduces an obvious bottleneck. Point 4 would be, instead, a significant departure from large mini-batches, and its benefit on the training convergence should be definitely investigated, while its scalability can be expected to be almost linear in terms of samples processed per unit of time, regardless of the scale of the deployment. Moreover, this approach would significantly benefit in loosely-coupled heterogeneous environments (e.g. *edge*), where the communication is costly and unreliable.

A. Incomplete topology training

Incomplete topologies has been explored theoretically in generic optimization context [59, 61–63], with only one deep learning application known to the authors [64]. This last work, while very close to what envisioned here, presents a theoretical discussion that is not really applicable to real training, even if sound. In fact, the authors consider a convergence criteria that is too strict to be relevant in a training scenario where, for instance, a sudden drop in the loss function happens after a very long plateu. In this sense, it is hard to find a formulation of the convergence rate that can really capture the dynamics of the optimisation for a real training case. The most promising direction, at the time of writing, is to draw a connection between the work done in general-purpose distributed optimization, and deep learning, similarly to what is done by Tatarenko et al. [63] when highlighting how their formulation corresponds exactly to SGD as described by Robbins and Monro [65].

It is clear that allowing partial communication definitely gives up on model consistency, even in the long run. The impact of this on the training must be better understood, as well as the policy to determine which model to choose as representative when the training ends. 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 [2], but only from the point of view of fault tolerance of the training system, not in terms of training accuracy.

Finally, it is necessary to investigate the impact of partial communication when more sophisticated optimization algorithms are used in place of naïve SGD. Momentum arises implicitly when introducing stale gradients [47], but there is no clear understanding of what happens in case of incomplete communication, as well as for more sophisticated algorithms with variable learning rates. It is reasonable to expect that the discussion made for the synchronous case by Goyal et al. [31] on momentum correction and aggregation of gradients subject to momentum can be extended for asynchronous set-ups with also implicit momentum and investigation is in progress in this sense.

To wrap up the discussion, asynchronous gradient descent with partial communication seems a promising alternative to more popular methodologies. The next section will discuss the requirements of a framework that can enable efficient experimentation on this topic.

IV. FAST C++ FRAMEWORK

This library is currently[1] under development and not yet publicly available. In order to provide a truly general purpose tool, as well as to exploit the peculiarities of the different deep learning frameworks available, the proposed FAST (Flexible (A)synchronous Scalable Training) approach intends to decouple the intra-node execution of the training from the parallel coordination of workers; in fact it is reasonable that the user desires to keep using its framework of choice (e.g. Tensorflow, PyTorch, MxNet).

The library is designed from scratch with C++ training in mind, according to the aim of making training code redistributable, while potentially target training in production and keeping the overhead as low possible. However, due to the prevalence of Python for DNN training, Python wrappers will be provided compatible with selected frameworks.

Figure 4 presents the logical architecture of the framework. FAST is designed to provide the user a pre-defined worker node, that should be filled with the code worth for one iteration of training (namely, one mini-batch). Then, the communication of gradients between workers is completely handled by the framework. The user is exclusively responsible for the inclusion of gradients created by other workers and provided locally by FAST, in the update of the local model replica.

At higher level, the global training strategy is defined by the topology of communications that interconnect the workers. In particular, it leverages distributed FastFlow [66, 67] to hide the SPMD machinery to the end-user [68], who is able to simply define a topology attaching worker nodes and channels. In a typical deployment, workers will be logically arranged in a 2D grid with a toroidal topology with indices (i, j), where neighbour relationships are defined with $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$. This topology allows for a gradient information to propagate quickly to all the workers, even if it is mediated with the gradients of intermediate workers. This kind of neighbour relationships can be encoded easily thanks to the distributed FastFlow API, that also allows for non-blocking collective communications among neighbours.

Within the nodes, multi-core FastFlow [69, 70] is used to hide latencies related to host-device data transfer, as well as to handle node-node transfers asynchronously with respect to the local training. Differently from other mainstream approaches adopting different programming models to exploit shared-memory, GPUs, and distributed processing elements (e.g. OpenMP+CUDA+MPI), FastFlow targets heterogenous platforms with a singles programming model, which exploits message-passing to model data dependencies and a globally distributed memory to share data among processing elements.

[1] Thursday 21st March, 2019

V. CONCLUSION AND FUTURE WORK

It is very likely that the next breakthrough in training performance will either come from new dedicated silicon architectures or from theoretical advancements in optimizations techniques that departs from gradient descent. However, at this stage, the quest for training performance at scale has been met mostly by synchronous, large mini-batch, parallelism; unfortunately this strategy is heavily problem-dependent, moreover, it is not suitable for other platforms than conventional HPC clusters and tightly coupled cloud instances.

This paper, while still lacking experimental results, advocates a departure from both synchronous and conventional asynchronous training, as they both perform similarly in terms convergence when working within a high-performance infrastructure, with clear bottlenecks that can prevent them to really scale over 128-256 GPU nodes. Instead, training with incomplete communication topology is expected to introduce a degree of randomization in the interleaving of updates coming from different mini-batches that represents a novelty with respect to large mini-batches and might arguably be beneficial to the training.

This approach would require an effort on both the theoretical and experimental side, in order to investigate the potential issues reported in Sec. III. This work is currently taking place and tackles the issues related to model inconsistency that derives from partial communications, while the development of FAST library will allow to validate theoretical results on real models and datasets.

Acknowledgments: This research has been supported by the *Competency Center on Scientific Computing* (C3S) at University of Turin [71], by the HPC4AI project funded by the Region Piedmont POR-FESR 2014-20 programme (INFRA-P) [72], and the OptiBike experiment in the H2020 projects Fortissimo2 (no. 680481).

REFERENCES

[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] 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.

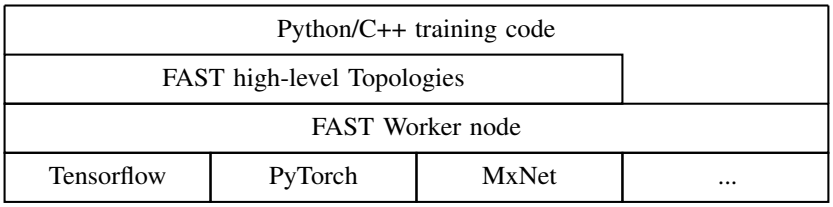

Fig. 4. FAST logical stack.

[3] J. Ngiam, Z. Chen, D. Chia, *et al.*, "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.

[4] 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.

[5] 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.

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, no. Supplement C, pp. 85–117, Jan. 2015.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," En, *Nature*, vol. 521, no. 7553, p. 436, May 2015.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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 (http://books.nips.cc), 2008, pp. 161–168.

[16] 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.

[17] 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.

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

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

[20] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145–151, Jan. 1999.

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

[22] 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.

[23] 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.

[24] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning," *CoRR*, vol. abs/1606.04838, 2016.

[25] S. Jastrzebski, Z. Kenton, D. Arpit, *et al.*, "Three Factors Influencing Minima in SGD," *CoRR*, vol. abs/1711.04623, 2017.

[26] S. L. Smith, P.-J. Kindermans, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," *CoRR*, vol. abs/1711.00489, 2017.

[27] J. Chen, R. Monga, S. Bengio, and R. Józefowicz, "Revisiting Distributed Synchronous SGD," *CoRR*, vol. abs/1604.00981, 2016.

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

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

[30] S. Chetlur, C. Woolley, P. Vandermersch, *et al.*, "cuDNN: Efficient Primitives for Deep Learning," *CoRR*, vol. abs/1410.0759, 2014.

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

[32] M. Cho, U. Finkler, S. Kumar, *et al.*, "PowerAI DDL," *CoRR*, vol. abs/1708.02188, 2017.

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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.

[39] N. Strom, "Scalable Distributed DNN Training Using Commodity GPU Cloud Computing," Dresden, Sep. 2015.

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

[41] S. Zheng, Q. Meng, T. Wang, *et al.*, "Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning," *CoRR*, vol. abs/1609.08326, 2016.

[42] 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.

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

[44] X. Lian, C. Zhang, H. Zhang, *et al.*, "Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent," *CoRR*, vol. abs/1705.09056, 2017.

[45] C. A. Coleman, D. Narayanan, D. Kang, *et al.*, "DAWNBench : An End-to-End Deep Learning Benchmark and Competition," 2017.

[46] J. Hermans, *On Scalable Deep Learning and Parallelizing Gradient Descent*, it, Syntethic version: http://joerihermans.com/ramblings/distributed-deep-learning-part-1-an-introduction/ Code https://github.com/cerndb/dist-keras, Aug. 2017.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] D. Yu, A. Eversole, M. Seltzer, *et al.*, "An Introduction to Computational Networks and the Computational Network Toolkit," *Microsoft Research*, Aug. 2014.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

[64] 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.

[65] H. Robbins and S. Monro, "A Stochastic Approximation Method," EN, *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, Sep. 1951.

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

[67] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, ser. LNCS, vol. 7640, Springer, 2013, pp. 47–56.

[68] M. Drocco, C. Misale, and M. Aldinucci, "A cluster-as-accelerator approach for SPMD-free data parallelism," in *Proc. of Intl. Euromicro PDP 2016: Parallel Distributed and network-based Processing*, Crete, Greece: IEEE, 2016, pp. 350–353.

[69] 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.

[70] M. Aldinucci, S. Ruggieri, and M. Torquati, "Porting decision tree algorithms to multicore using FastFlow," in *Proc. of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, ser. LNCS, vol. 6321, Barcelona, Spain: Springer, Sep. 2010, pp. 7–23.

[71] 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 898 (CHEP 2016)*, San Francisco, USA, 2017.

[72] M. Aldinucci, S. Rabellino, M. Pironti, *et al.*, "HPC4AI, an AI-on-demand federated platform endeavour," in *ACM Computing Frontiers*, Ischia, Italy, May 2018.