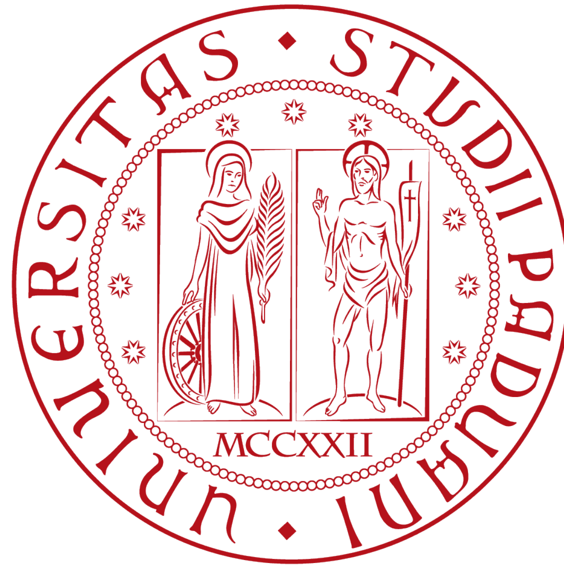
Università degli Studi di Padova

Department of Mathematics
"Tullio Levi-Civita"

Master Degree in Data Science



Distributed optimization for big data applications

Candidate
Matteo Migliorini
N. 1182367

Supervisor
Prof. Francesco Rinaldi

Anno Accademico 2019-2020

Do. Or do not. There is no try.
— Yoda

Contents

1 Introduction **1**
1.1 Problem definition . . . 3
1.2 Aim of the work . . . 4

2 State-of-the-art algorithms **6**
2.1 Parallel SGD . . . 6
2.2 Parallel Decentralized SGD . . . 9
2.3 Swarm SGD . . . 13

3 Momentum in distributed SGD **17**
3.1 Convergence of SlowMo . . . 19
3.2 Removing the average . . . 23

4 Applications **25**
4.1 Implementation . . . 25
4.2 Standard Benchmark . . . 26
4.2.1 Scalability . . . 26
4.2.2 Convergence . . . 28
4.3 High Energy Physics . . . 31
4.3.1 Problem description . . . 31
4.3.2 Models . . . 35
4.3.3 Results . . . 36

5 Conclusions **41**

Abstract

Speeding up gradient based methods via distributed optimization has been a subject of interest over the past years, especially in the context of training deep learning models. Despite the fact that many improvements have been done on hardware level, the convergence time of models on large datasets remains problematic. For this reason, methods based on data parallelism paradigm and mini batch SGD have been proposed to decrease the training time.

However, the most common synchronous implementation of parallel SGD, which can be found in many deep learning libraries, has a behaviour that heavily depends on the hardware performances. For this reason, methods that are robust to heterogeneous environments are starting to get more and more interest. In this thesis, two decentralized and communication efficient variants of parallel SGD will be presented and compared. Furthermore, we will study how momentum can be used to improve the performance of distributed decentralized methods. Then a modification of this method will be proposed and numerically validated: we will empirically show that even by removing the periodical synchronization barrier present in the method we are still able to improve the performance of the base optimizer.

These methods will then be used to extend a work done at CERN in the context of large scale machine learning. We will show how the algorithms combined with momentum presented in this thesis can improve the final performance of the classifiers.

Chapter 1

Introduction

In recent years, it has been shown that the efficient training of large and deep neural networks guarantees state-of-the-art performances in many fields such as computer vision [Krizhevsky et al., 2012] and language processing [Devlin et al., 2018]. Furthermore, increasing the number of parameters in the model, the number of training examples, or both, leads to a better model accuracy [Sun et al., 2017] [Cireşan et al., 2010]. However, even when performed on a GPU, training of a neural network on a large dataset can take excessively long time. For example, training a model such as ResNet-50 on the ImageNet dataset would take around 10 days, while for bigger models this time could be of the order of 2 weeks. Moreover, training on a single machine with limited resources will limit the size of the models that can be trained.
The amount of data collected keeps growing constantly; not only in the context of big technology companies and government organization, but also in scientific experiments. This is the case for example of CERN: in 2025 its main accelerator will be upgraded [Apollinari et al., 2015] and experiments will produce over 100PB per year. In this scenarios, being able to accelerate the training of models using multiple machine is of fundamental importance.

Distributed deep learning tries to address these issues by decomposing the problem into multiple machines and devices. In [Dean et al., 2012] two new paradigms to decrease the training time for large models and large datasets are introduced: *model parallelism* and *data parallelism*.

The first paradigm, *model parallelism*, is the most straightforward since it deals with the parallelization of the computation within a single model: computations of different parts of the model are carried on different processes or machines. The benefits of this approach heavily depends on the structure of the model: for a large number of parameters there is a computational benefit in using multiple cores up

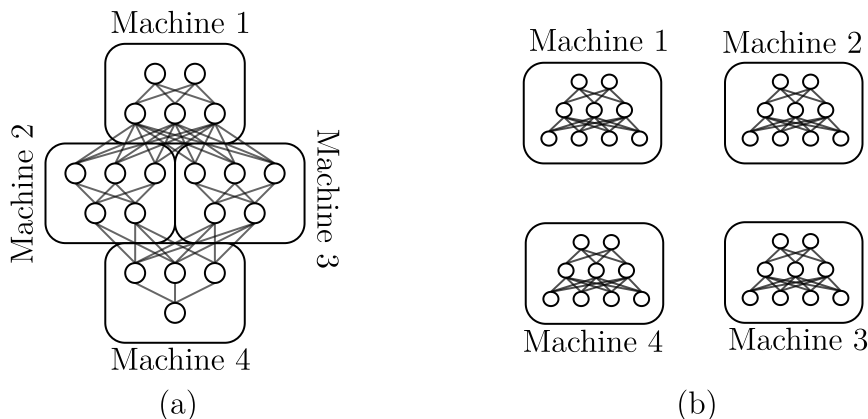


Figure 1.1: Example of model parallelism (a) and data parallelism (b): in the former the model is divided into four parts distributed on four different machines, in the later the same model is replicated in four different machines.

to the point where the communication time becomes too high. The communication time in this case is related to the propagation of weights between different parts of the model. An example of model parallelism is shown on figure 1.1(a). If a model is too big to fit into a single machine this provides a possible solution. This is often the case for GPUs, where memory is of the order of 10MB.

The second paradigm is *data parallelism*, which will be the main focus of this thesis. In data parallelism the training set is divided into m partitions, called data shards, that will be used by m different processes, called workers or agents, to train a replica of the model (figure 1.1(b)). In other words, each worker has an identical copy of the model and trains it on a subset of the data. If no communication between workers is involved the result would be m independent training process of the same model on different subsets of the dataset, and the result of the training would be different in each worker. Furthermore, an instance would not be able to obtain any kind of information coming from other partition therefore losing the benefit of a large dataset. For this reason workers periodically communicate to synchronize their local model with others and to obtain information on the training over other partitions. This allows us to avoid the divergence of the local models, i.e. becoming too different from models trained by other workers. How and how frequently workers synchronize is defined by the optimization algorithm.
Ideally, if the cost of communicating between workers is zero, with this approach the model will be trained m times faster with respect to the non distributed setting, since each worker uses a fraction $1/m$ of the training examples. In practice often the communication becomes a bottleneck thus nullifying the benefits of adding more workers. For this reason, developing optimization algorithms capable of reducing the

impact of communication time while keeping workers synchronized, i.e. making sure that each instance is benefiting from the information obtained by other workers on different data, is of fundamental importance.

1.1 Problem definition

In the training of a neural network we are interested in minimizing the expected risk, that is equivalent at solving the stochastic optimization problem

$$\min_{x \in \mathbb{R}^N} f(x) := \mathbb{E}_{\xi \sim \mathcal{D}} \left[F(x; \xi) \right] \tag{1.1.1}$$

where F is a function involving model parameters x and a random variable ξ sampled from $\mathcal{D}$. In our case ξ will refer to a data sample and its label. In the distributed training there is a network of m agents, each of them with its local data and a local loss function

$$f_i(x) = \mathbb{E}_{\xi \sim \mathcal{D}_i} \left[F_i(x; \xi) \right]. \tag{1.1.2}$$

where $\mathcal{D}_i$ is the local data distribution. In this way the minimization problem (1.1.1) can be rewritten as

$$\min_{x \in \mathbb{R}^N} f(x) = \frac{1}{m} \sum_{i=1}^{m} f_i(x) \tag{1.1.3}$$

This optimization problem is called consensus optimization, where m agents cooperate to find the parameters x minimizing the average objective function with respect their local data.
There are two possible strategies used to distribute data between workers:

- **Strategy 1**: $\mathcal{D}_i = \mathcal{D}$, i.e. all worker can access all training examples. Consequently we have that $F_i(\cdot;\cdot) = F(\cdot;\cdot)$, that is all $f_i(\cdot)$'s are the same.

- **Strategy 2**: Data are split between all workers and $\mathcal{D}_i$ is a uniform distribution over the samples assigned to the i-th worker.

The second strategy is of particular interest in the case when data are not centrally-collected: each agent collect its set of data and wants to train the same model as other agents without sharing local dataset. A practical application of this is training a model when data cannot be moved from the source due to privacy constraints, e.g. for medical data collected by a hospital.
However, in practical cases the distributions are not too different. That is, if the variance in the data is not too high, it is possible to assume that the distribution on each worker $\mathcal{D}_i$ can be approximated with the same distribution $\mathcal{D}$. In this thesis we will make this assumption.

1.2 Aim of the work

In this thesis, after introducing the standard way of parallelizing gradient descent, we will focus on two state of the art decentralized methods for the training of neural networks. There is a growing interest in this kind of methods since they are communication efficient hence they scale well with the number of parallel workers, which is a key component when working on large dataset. Therefore, they are robust in heterogeneous environments such as clouds, therefore they are accessible to a larger pool of researchers that cannot benefit from High Performance Computing centers.

Furthermore, a recent proposal on how to combine momentum with distributed algorithms will be examined. The idea behind it is to perform a certain amount of iterations of a base optimizer and use this information to perform the next update, as for the classical momentum. Then, one possible modification of the method suggested by the authors will be numerically studied and empirically motivated.

Some of the key questions when studying a distributed optimization algorithm that we will address in this work are:

- Is the method able to recover the performance of the sequential case?
- Can it achieve a linear speedup with respect to the number of workers?
- Is including momentum improving the performance of the algorithms?

Numerical tests will be performed on standard benchmarks before analyzing a large scale problem coming from the world of High Energy Physics. In this field, algorithms capable of dealing with a huge amount of data while being efficient in terms of resource utilization are of primary importance. The work carried on at CERN in the context of building a new machine learning pipeline for large datasets [Migliorini et al., 2019] will be extended and we will see how it can be improved using the methods described in this thesis.

Chapter 2

State-of-the-art algorithms

In this chapter, we will first describe the standard approach adopted to parallelize stochastic gradient descent, then two recently proposed variants will be introduced. The common procedure among all distributed methods is that every worker performs a certain number of local steps before communicating with the others. These local steps are usually based on mini-batch SGD, which computes the update using a batch of B samples. In the training of neural network, the local loss function of each worker introduced in (1.1.2) can be viewed as the sum of losses ℓ_i incurred on its local examples

$$f_i(x) = \frac{1}{N_i} \sum_{j=1}^{N_i} \ell(x, \xi_j)$$

Therefore, considering B independent realizations of $\xi \sim \mathcal{D}_i$, mini-batch SGD computes the following gradient

$$\nabla_B f_i(x) = \frac{1}{B} \sum_{j=1}^{B} \nabla \ell(x, \xi_j) \tag{2.0.1}$$

which is an approximation of ∇f_i. Then, it can be used to compute the update

$$x_{t+1} = x_t - \gamma \nabla_B f_i(x_t)$$

2.1 Parallel SGD

The easiest way of parallelizing SGD is built on top of the idea of mini-batch presented before. If we consider a mini-batch of size $m \cdot B$ samples, each agent can compute a local mini-batch stochastic gradient using B of its local samples. Then all the gradients are aggregated and the average is computed in every node, i.e.

$\frac{1}{m}\sum_{i=1}^{m}\nabla_B f_i\left(x\right)$. Recalling that the objective function we are interested in minimizing is (1.1.3) we have that the computed average provides a stochastic gradient of the objective function f:

$$\nabla f = \frac{1}{m}\sum_{i=1}^{m}\nabla f_i\left(x\right). \tag{2.1.1}$$

This gradient can be used to update the weight of each local model. Notice that if all the m workers start with the same initial model x_0 then they will generate a sequence $\{x_t\}_{t=1,\cdots,T}$ of identical weights. It has been proved that sequential SGD [Ghadimi and Lan, 2013] admits the optimal convergence rate $\mathcal{O}\left(1/\sqrt{T}\right)$ for non-convex functions. Similarly, for PARALLEL-SGD the convergence rate is proved to be $\mathcal{O}\left(1/\sqrt{mT}\right)$ on non convex problems [Ghadimi et al., 2016][Lian et al., 2015] hence obtaining a linear speedup with respect to the number of workers m.
Gradients can be aggregated in two possible way: using a central node or via an ALLREDUCE operation.

When gradients are aggregated by a central node we have a centralized algorithm (CENTRALIZED-PARALLELSGD). In this scheme, for each iteration, every worker fetches the model x saved in a central node, called parameter server, and computes the mini-batch gradient using its local data. Gradients are then pushed back to the parameter server, which averages them as in (2.1.1) and uses this average to update the central model x' before moving on to the next iteration. This procedure is shown in Figure 2.1(a). This approach used to be leveraged by many popular machine learning frameworks such as Tensorflow[Abadi et al., 2016] and CNTK[Seide and Agarwal, 2016]. One problem of this method is that if one parameter server is used it will likely become a computational or networking bottleneck. For this reason, methods based on collective operations that do not require a parameter server started to arise interest.

In an ALLREDUCE operation a target quantity is reduced, e.g summed, in every process. This kind of operation is implemented in many communication libraries such as OpenMPI[Gabriel et al., 2004a]. This is useful because if every worker starts with its gradient ∇f_i then after the ALLREDUCE it will end up with $\sum_{i=1}^{m}\nabla f_i$ which is the quantity we are interested to have in order to update the model.
A straightforward way of implementing this operation would require each worker to send its gradient to all the other workers. This cannot be done because the communication pattern would be “all-to-all” which may saturate the network and we would fall back to the parameter server case. For this reason, in modern communication frameworks this is implemented in many efficient ways; for example, the most used in deep learning is the RING-ALLREDUCE. This approach has been introduced by

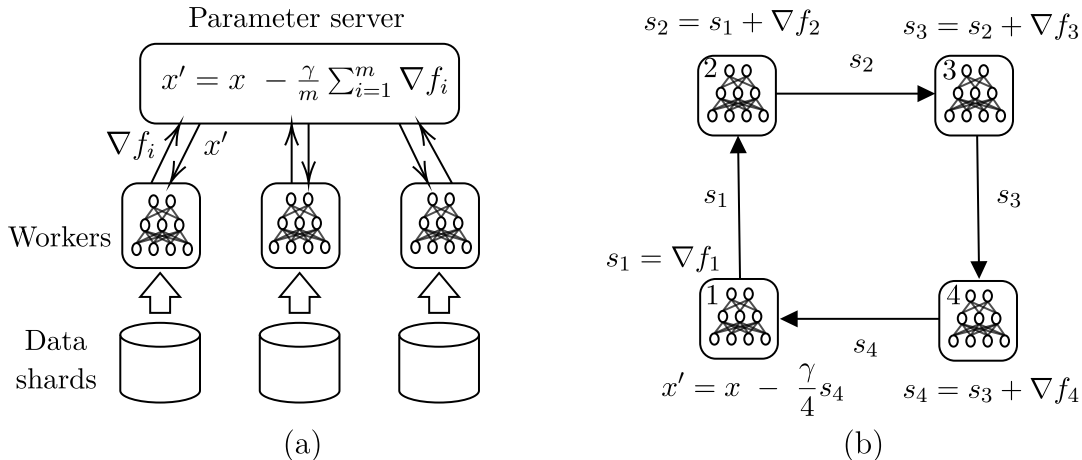


Figure 2.1: Strategies for worker synchronization in parallel SGD: (a) Parameter server strategy for centralized-parallel SGD(C-PSGD), (b) Gradient aggregation based on a Ring-AllReduce operation (AR-SGD).

Baidu Research[Gibiansky., 2017] inspired by the work [Patarasuk and Yuan, 2009].

In this algorithm nodes are on a ring and each of them communicates only with its two peers. The basic idea is that each worker sends its gradient to the next peer in the ring, which receives it and adds it to its local gradient. That is, node 1 sends $s_1 = \nabla f_1$ to node 2 which will then have $s_2 = \nabla f_1 + \nabla f_2$ as shown in Figure 2.1(b). Node 2 will then proceed to send s_2 to node 3 which will add its gradient and so on. In the last step worker 1 will receive $s_m = \sum_{i=1}^{m} \nabla f_i$ that can be used to update its local model. The same procedure is carried at the same time for each other worker hence in the last iteration they will be all synchronized. This is just a simple example, since in this case $\mathcal{O}(m)$ gradients would be sent amongst the workers. In a more realistic implementation, the gradient in a worker is split into m chunks: at each iteration only one chunk is communicated with the procedure described above, resulting in a $\mathcal{O}(1)$ gradients communication and $\mathcal{O}(m)$ time. In [Patarasuk and Yuan, 2009] it is suggested that this algorithm is bandwidth optimal, i.e. it will optimally utilize the available network. This method is the most used nowadays to perform data parallel training and it is implemented in many frameworks such as Pytorch[Paszke et al., 2019], Tensorflow (replacing parameter server)[Abadi et al., 2016] and Horovod [Sergeev and Del Balso, 2018]. In the rest of this work we will refer to it as AR-SGD and the pseudocode can be found in Algorithm 1.

With modern networks capable of handling bandwidth of the order $10 - 10\,\text{GB/s}$ combined with neural network parameter sizes on the order of $10\,\text{MB}$ the commu-

Algorithm 1: AR-SGD

Input: Learning rate γ_t; Number of workers m; Number of iterations T; Initial point $x_{0,0}$

1 **for** $t \in \{0, 1, \ldots T-1\}$ **at worker** i **in parallel do**
2   compute local gradient: $\Delta x_i = \nabla f_i(x_t)$
3   reduce gradients: $\Delta x = \textsc{AllReduce}(\Delta x_i / m)$
4   update local model: $x_{t+1} = x_t - \gamma_t \Delta x$
5 **end**

nication of gradient across workers can be very fast. The main bottleneck is in the synchronous nature of the algorithm: before updating the local model every node must wait for all the others to finish their local computation. Thus this system is extremely sensible to the stragglers effect and the iteration time is defined by the time of the slowest worker.
Moreover, there are also intrinsic problems related to mini-batch SGD: it has been observed that training on large batches, i.e. with a large number of workers, leads to a significant decrease of the model performance[Goyal et al., 2017]. This effect has been studied in [Keskar et al., 2016] and it is believed that is due to the fact that large batch SGD tends to converge to sharper minima of the training loss function and this leads to worse generalization performance.

One way to tackle both problems is the one adopted in LocalSGD[Lin et al., 2018], where workers perform H local steps before synchronization. After H local steps parameters are averaged via an AllReduce operation, reducing in this way the effect of stragglers since the cost of communication is "spread" over H iterations. In [Wang and Joshi, 2018] it is shown how this approach reduces communication time. However, also in this case an AllReduce operation is required, hence this method may be problematic for a large number of workers or slow networks.

2.2 Parallel Decentralized SGD

Recently, decentralized training algorithms are getting a significant amount of attention. The interesting aspect is that each worker communicates with only a subset of other workers thus removing the global synchronization required by algorithms such as AR-SGD. This results in a lower idling time, especially in high latency networks and heterogeneous environments such as cloud computing.
A recent work has shown that decentralized algorithms can outperform the centralized counterpart for distributed training[Lian et al., 2017a]. In this work it is proved

that decentralized parallel SGD (D-PSGD) has a comparable total complexity to the centralized SGD while involving less communication.

To understand the idea behind this method, we can make the following consideration. In centralized methods averaging local models every iteration leads to the same result as aggregating gradients. Every worker at step t uses the same model x_t to compute the local gradient $\nabla f_i\left(x_t\right)$ and to update the local model $x_{t+1}^{(i)} = x_t - \nabla f_i\left(x_t\right)$. The average of the local model is computed either using a parameter server or an AllReduce operation:

$$x_{t+1} = \frac{1}{m}\sum_{i=1}^{m} x_{t+1}^{(i)} = x_t - \frac{\gamma}{m}\sum_{i=1}^{m} \nabla f_i\left(x_t\right)$$

which is the same update as the one described in the previous chapter for centralized parallel SGD (C-PSGD). The idea behind decentralized methods is to use a gossip algorithm to approximate the distributed averaging. If $x_t^{(i)} \in \mathbb{R}^d$ is the vector containing the parameters at node i, we are interested in approximating $\frac{1}{m}\sum_{i=1}^{m} x_t^{(i)}$. We can concatenate the models into a matrix $X_t \in \mathbb{R}^{m\times d}$, where each row will be a different node. A gossip iteration has the form $X_{t+1} = W_t X_t$ where $W_t \in \mathbb{R}^{m\times m}$ is the mixing matrix and defines the communication topology. The index t represent the fact that the mixing matrix can be different for each iteration. This corresponds to the update at node i

$$x_{t+1}^{(i)} = \sum_{j=1}^{m} w_t^{i,j} x_t^{(i)}$$

Therefore, a node needs to communicate only with nodes j for which $w_t^{i,j} \neq 0$ so for sparser matrices there is less communication. If $w_t^{i,j} = 1/m$ for all i, j then case of C-PSGD is recovered.

In D-PSGD[Lian et al., 2017a] the communication topology is described by an undirected graph (V, W). V denotes the computational nodes $V = 1, \ldots, m$ and $W \in \mathbb{R}^{m\times m}$ a symmetric doubly stochastic matrix. As in C-PSGD every worker has a local copy of the model. In each iteration, all workers compute stochastic gradients locally and at the same time average the local model with their neighbours. Finally, the local gradients are used to update the averaged local models. The pseudocode of the basic scheme, i.e. without mini-batches, is reported in Algorithm 2. Notice that lines 3 and 4 can be run in parallel: if the communication time used to average models is smaller than the computation time, then it can be completely hidden. At line 5 there is a synchronization barrier: before updating the local models and moving into the next iteration, all workers must have computed the gradient and averaged their local model with neighbours.

Algorithm 2: D-PSGD

Input: Learning rate γ_t; Number of workers m; Number of iterations T; Weight matrix W; Initial point $x_{0,0}$

1 **for** $t \in \{0, 1, \dots T-1\}$ **at worker** i **in parallel do**

2 Sample $\xi_t^{(i)}$ from local data

3 Compute a local stochastic gradient based on $\xi_k^{(i)}$ and current model $x_t^{(i)}$ $\nabla F_i\left(x_t^{(i)}, \xi_t^{(i)}\right)$

4 Compute the neighbourhood weighted average $x_{t+\frac{1}{2}}^{(i)} = \sum_{j=1}^{m} w^{i,j} x_t^{(j)}$

5 Synchronization barrier: wait until all workers reach this point

6 Update local model $x_{t+1}^{(i)} = x_{t+\frac{1}{2}}^{(i)} - \gamma \nabla F_i\left(x_t^{(i)}, \xi_t^{(i)}\right)$

7 **end**

8 **return** $\frac{1}{m}\sum_{i=1}^{m} x_T^{(i)}$

Under the following standard assumptions it is possible to analyze the convergence rate of D-PSGD:

Assumption 2.1. *(Smooth Gradients). Each local objective function $f_i(x)$ is L-Lipschitz continuous, i.e. for all $x, y \in \mathbb{R}^d$ and $i \in \{1, \dots m\}$*

$$\|\nabla f_i(x) - \nabla f_i(y)\| \leq L\|x - y\|$$

Assumption 2.2. *(Spectral Gap). Given the symmetric doubly stochastic matrix W and define*

$$\rho = \left(\max\{|\lambda_2(W)|, |\lambda_n(W)|\}\right)^2$$

where $\lambda_i(\cdot)$ denotes the i-th largest eigenvalue. ρ is assumed to be $\rho < 1$.

Assumption 2.3. *(Bounded Variance). The variance of the stochastic gradient is bounded, i.e. there exists a finite positive constant σ^2 such that for all $i \in \{1, \dots m\}$*

$$\mathbb{E}_{\xi \sim \mathcal{D}}\|\nabla F_i(x; \xi) - \nabla f_i(x)\|^2 \leq \sigma^2$$

Under these assumptions it is possible to prove that by setting a learning rate $\gamma = \frac{1}{2T + \sigma\sqrt{T/m}}$ and considering a sufficient large number of iterations then the convergence rate is

$$\frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}\|\nabla f(x_t)\|^2 \leq \frac{8(f(0) - f^\star)L}{T} + \frac{(8f(0) - f^\star + 4L)\sigma}{\sqrt{mT}} \tag{2.2.1}$$

where $x_t = \frac{1}{m}\sum_{i=1}^{m} x_t^{(i)}$ is the average of all local models. The number of iterations needed to obtain this result depends on ρ: smaller values implies a higher connectivity of the graph. In other words more workers are communicating at each iteration

thus obtaining a better approximation of the average. Therefore, considering a sufficient number of iterations T the convergence rate is $\mathcal{O}\left(\frac{1}{T}+\frac{1}{\sqrt{mT}}\right)$. For a large T the term $\frac{1}{T}$ is dominated by $\frac{1}{\sqrt{mT}}$ i.e. the centralized SGD convergence rate is recovered even in the decentralized case. In the experiment section of the work, the author used a ring communication topology: each worker communicates only with its two peers every iteration, resulting in a much smaller communication cost with respect to C-PSGD ($\mathcal{O}(1)$ vs $\mathcal{O}(n)$).

The same authors of D-PSGD proposed a slight variation of this method called asynchronous decentralized parallel SGD (AD-PSGD)[Lian et al., 2017b]. In particular they proved that by removing the synchronization barrier in D-PSGD, which could cause long idle time of the faster workers, it is still possible to recover C-PSGD convergence rate. This approach makes the algorithm robust in heterogeneous environment and more scalable than D-PSGD.
The algorithm is nearly the same as Algorithm 2. In every iteration, each worker performs the following steps:

- Compute the local gradient $\nabla F_i\left(\hat{x}_t^i, \xi_k^i\right)$ where $\hat{x}_t^i$ is the model read from the worker memory.
- In parallel the model $\hat{x}_t^i$ may be averaged with other workers to obtain x_t^i. This average is performed between two peers i and j

$$x_t^i = x_t^j = \frac{\hat{x}_t^i + \hat{x}_t^j}{2}$$

- The gradient is used to update the local model

$$\hat{x}_{t+1}^i = x_t^i - \nabla F_i\left(\hat{x}_t^i, \xi_t^i\right)$$

In this case the mixing matrix W_t depends on the iteration since the average is done between random workers and it is obtained as a sample from G. To prove the convergence of this method the same assumptions as for D-PSGD are made, with the only difference in the fact that the spectral gap in Assumption 2.2 is expressed in terms of expectation of the i-th singular value of W_t, that is

$$\max\{|\lambda_2\left(W_t^\top W_t\right)|, |\lambda_n\left(W_t^\top W_t\right)|\} \leq \rho, \quad 0 \leq \rho < 1.$$

Furthermore the staleness is assumed to be bounded by a constant τ_t: $\hat{x}_t^i = x_{t-\tau_t}^i$, i.e. the model update is not performed on a model that is too many iterations ahead with respect to the one used to compute the gradient. This is caused by the fact that while the gradient is being computed, τ averages with different workers may

happen. Under these assumptions it is possible to prove that the convergence rate is the same as SGD $\mathcal{O}\left(1/\sqrt{T}\right)$. In the analysis every SGD update in a worker is counted as one iteration, even though they are happening in parallel. Hence, m parallel workers will make the iteration counter proceed m times faster in the sense of wall-time thus obtaining a linear speedup with respect to the number of workers.

2.3 Swarm SGD

Algorithm 3: swarmSGD

Input: Learning rate γ_t; Number of workers m; Number of iterations T; Number of local steps H_i in worker i; Graph G; Initial point $x_{0,0}$

1 **for** $t \in \{0, 1, \ldots T-1\}$ **at worker** i **in parallel do**
2   **for** $q \in \{0, \cdots, H_i - 1\}$ **do**
3   Sample $\xi_t^{(i)}$ from local data
4     $x_{t,q+1}^{(i)} = x_{t,q}^{(i)} - \gamma_t \nabla F_i \left(x_{t,q}^{(i)}, \xi_{t,q}^{(i)}\right)$
5   **end**
6   Sample edge (i, j) from G
7   Average local parameters of workers i and j:

$$x_{t+1}^{(i)}, x_{t+1}^{(j)} = \frac{x_{t,H_i}^{(i)} + x_{t,H_j}^{(j)}}{2}$$

8 **end**
9 **return** $\frac{1}{m} \sum_{i=1}^{m} x_T^{(i)}$

In decentralized methods the impact of communication between workers is reduced by communicating only with a subset of peers. The same goal can be pursued in methods involving a global communication, such as All-Reduce, by allowing workers to perform local model updates before synchronization, as for LocalSGD. In SwarmSGD [Nadiradze et al., 2019] these two approaches are combined in order to have a method capable of scaling to a large number of nodes.

Conceptually this method is very similar to AD-PSGD: a population of m workers is given and each of them can perform sequential SGD steps on its local model using a subset of training data. Workers are identified as node of a graph and edges are the possible communication links. After executing a certain number of local optimization steps a node communicates with one of its neighbours that is chosen randomly amongst its peers in the network. During the interaction the two model

are averaged in each worker. Combining the pairwise interactions with local steps leads to a negligible communication time. The pseudocode for the basic scheme without mini-batches is reported in Algorithm 3.

It is possible to prove that even this extremely decentralized SGD variant can converge. The convergence can be proved under standard assumptions such as smooth gradients (Assumption 2.1), bounded variance of the stochastic gradient on each agent i (Assumption 2.3) and bounded second moment, i.e. there exist a $M^2 > 0$ such that

$$\mathbb{E}\|\nabla F_i\left(x\right)\|^2 \leq M^2.$$

The last assumption is necessary only in the case of the analysis carried on for a random number of local steps H_i. If this value is fixed it is still possible to prove the convergence even without this assumption. Furthermore, the graph G describing the communication topology it is assumed to be a r-regular graph with a spectral gap λ_2.

In the case of a random interaction time H_i and H_j are independent geometrically distributed random variables with mean H. By assuming Lipschitz continuous gradients, bounded second moment of the gradients and using a learning rate $\gamma = \frac{m}{\sqrt{T}}$ it is possible to prove that, for a sufficiently large number of interactions the following bound holds:

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T-1}\mathbb{E}\|\nabla f\left(x_t\right)\|^2 \leq & \frac{\left(f\left(x_0\right) - f\left(x^{\star}\right)\right)}{\sqrt{T}H} \\ & + \frac{11H^2 \max\left(1, L^2\right)M^2}{\sqrt{T}} \max\left(1, 2r/\lambda_2 + 4r^2/\lambda_2^2\right)\end{aligned} \tag{2.3.1}$$

where x_t is the average of $x_t^{(i)}$ amongst all nodes. The first term represent the reduction of the loss with respect to the initial point, divided by the square root of T and H. The linear speedup in term of number of worker is not evident here since the analysis is carried on modeling one interaction between two agents at every iteration t. But m of such iterations can occur in parallel, hence there is a liner speedup in terms of wall clock time. The second term represents the influence of every local steps, H^2M^2, and the impact of the network topology. For large values of λ_2, which corresponds to a well-connected graph, this term impact less negatively the convergence.

For a fixed interaction time H, assuming the smoothness of the gradients and

bounded variance it is possible to obtain a similar convergence rate:

$$\frac{1}{T}\sum_{t=1}^{T-1}\mathbb{E}\|\nabla f\left(x_t\right)\|^2 \leq \frac{\mathbb{E}[f\left(x_0\right) - f\left(x^\star\right)]}{\sqrt{T}H} + \frac{28H^2 \max\left(1, L^2\right)\sigma^2}{\sqrt{T}} \max\Big(1, 2r/\lambda_2 + 4r^2/\lambda_2^2\Big). \quad (2.3.2)$$

This result is almost identical to the one obtained in the previous case, with the only main difference that the bound on the second moment M^2 is replaced by the variance σ^2.

In both cases the proof is built around two ideas. The first is that due to pairwise averaging nodes' parameters $x^{(i)}$ are clustered around their mean, i.e the quantity

$$\Gamma_t = \sum_{i=1}^{m} \|x_t - x_t^{(i)}\|^2$$

is bounded, thanks to the periodic pairwise averages. The second is that even though stochastic gradients are taken as noisy estimates of this mean, the impact of the noise can be bounded.

Chapter 3

Momentum in distributed SGD

Momentum plays an important role in the training of neural networks, and it has been empirically demonstrated that it leads to better generalization performances [Sutskever et al., 2013]. However it is not well defined how momentum can be used to improve distributed training. In many recent works momentum has been incorporated in the training by allowing each worker to have its local momentum [Lian et al., 2017a][Assran et al., 2018][Lin et al., 2018].

Recently, a framework called *slow momentum* (SlowMo) [Wang et al., 2019] has been proposed to increase the accuracy of distributed training methods. In SlowMO workers after taking some number of local steps τ of a base optimizer, which could be SGD or a distributed optimizer, average their parameters via an AllReduce operation and perform a momentum-like update. The AllReduce operation allows to keep the system decentralized, otherwise a parameter server would be needed.

As said, SlowMo is built on top of a base optimizer and presents a nested loop structure, as shown in Algorithm 4. Each worker i maintains two local variables used during the training: the model parameters $x_{t,k}^{(i)}$ and a slow momentum buffer u_t. In the former t, k indicate the k-th step of the inner optimizer at the t-th iteration of SlowMo. Models are initialized with the same parameters $x_{0,0}$ and slow momentum buffer $u_0 = 0$ then the superscript i in the slow momentum buffer is not necessary because it will be always the same in every worker.

In the steps of the base optimizer the update is indicated by $x_{t,k+1}^{(i)} = x_{t,k}^{(i)} - \gamma_t d_{t,k}^{(i)}$ where γ_t is the learning rate and $d_{t,k}^{(i)}$ the update direction. If communication or local momentum updates are involved then $d_{t,k}^{(i)}$ represent the full update at the worker i at step k. In the case of SGD the direction is $d_{t,k}^{(i)} = \nabla F_i \left(x_{t,k}^{(i)}; \xi_{t,k}^{(i)} \right)$.

Algorithm 4: Slow Momentum

Input: Base optimizer with learning rate γ_t; Base optimizer steps τ; Slow learning rate α; Slow momentum factor β; Number of workers m; Initial point $x_{0,0}$

1 Initialize slow momentum buffer $u_0 = 0$
2 **for** $t \in \{0, 1, \dots T-1\}$ **at worker** i **in parallel do**
3   **for** $k \in 0, 1, \dots, \tau - 1$ **do**
4   Base optimizer step: $x^{(i)}_{t,k+1} = x^{(i)}_{t,k} - \gamma_t d^{(i)}_{t,k}$
5   **end**
6   Exact-Average: $x_{t,\tau} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}_{t,\tau}$
7   Slow Momentum update: $u_{t+1} = \beta u_t + \frac{1}{\gamma_t}(x_{t,0} - x_{t,\tau})$
8   Outer iterate updated: $x_{t+1,0} = x_{t,0} - \alpha \gamma_t u_{t+1}$
9 **end**

After τ base optimizer steps the workers compute the exact average $x_{t,\tau}$ of all the models parameters via an AllReduce operation. This average can be written in terms of all the update directions and the previous iteration of the outer loop

$$x_{t,\tau} = x_{t,0} - \frac{\gamma_t}{m} \sum_{i=1}^{m} \sum_{k=0}^{\tau-1} d^{(i)}_{t,k}.$$

Notice that this is a blocking operation, i.e. every worker must reach this point before continuing. After computing the average the momentum update is then performed in lines 7-8:

$$u_{t+1} = \beta u_t + \frac{1}{\gamma_t}(x_{t,0} - x_{t,\tau}) \tag{3.0.1}$$

$$x_{t+1,0} = x_{t,0} - \alpha \gamma_t u_{t+1}. \tag{3.0.2}$$

The difference between the model weight before and after the base optimizer phase in (3.0.2) is divided by the base optimizer learning rate in order to make it invariant with respect to the learning rate that can change during the training. Furthermore, in the update (3.0.2) the learning rate is multiplied by a factor α. However it has been empirically observed by the authors that $\alpha = 1$ leads to the best performance. By setting different values of β, α and τ other optimizers such as LocalSGD [Lin et al., 2018], BMUF [Chen and Huo, 2016] and LookAhead [Zhang et al., 2019] can be expressed in this framework allowing to provide bounds on the convergence.

Authors of SlowMo carried out an extensive empirical evaluation on multiple datasets and models and observed that it consistently improved optimization and

generalization performances, especially in cases where a decentralized optimizer is used as base optimizer.

3.1 Convergence of SlowMo

It is possible to provide convergence guarantees for SlowMo and prove that it leads to a linear speedup in terms of workers. We can denote with $f_i(x) = \mathbb{E}_{\xi_i \sim \mathcal{D}_i} F_i(x; \xi_i)$ the expected objective function at worker i, $f(x) = \frac{1}{m}\sum_{i=1}^{m} f_i(x)$ the objective function we are interested in minimizing and make the following standard assumptions

Assumption 3.1. *(Smooth Gradients). Each local objective function $f_i(x)$ is L-Lipschitz continuous, i.e. for all $x, y \in \mathbb{R}^d$ and $i \in \{1, \dots m\}$*

$$\|\nabla f_i(x) - \nabla f_i(y)\| \leq L\|x - y\|$$

Assumption 3.2. *(Bounded Variance). The variance of the stochastic gradient is bounded, i.e. there exists a finite positive constant σ^2 such that for all $i \in \{1, \dots m\}$*

$$\mathbb{E}_{\xi \sim \mathcal{D}}\|\nabla F_i(x; \xi) - \nabla f_i(x)\|^2 \leq \sigma^2$$

Assumption 3.3. *Let $d_{t,k} = \frac{1}{m}\sum_{i=1}^{m} d_{t,k}^{(i)}$ be the average descent direction across the m workers. There exists a finite positive constant V such that*

$$\mathbb{E}\|d_{t,k} - \mathbb{E}_{t,k}[d_{t,k}]\|^2 \leq V$$

where $\mathbb{E}_{t,k}$ represent the expectation conditioned on all the randomness from stochastic gradients up to the k-th step of the t-th outer iteration.

Example. We can look if SwarmSGD satisfies these assumptions. The first two are already required by the method, hence we only need to verify the third assumption by finding the constant V. Defining the following quantities

$$\begin{aligned} \mathbf{X}_k &= [x_k^{(1)}, \dots, x_k^{(m)}]^\top \in \mathbb{R}^{m \times d} \\ \nabla \mathbf{F}(\mathbf{X}_k) &= [\nabla F_1\left(x_k^{(1)}; \xi_k^{(1)}\right), \dots, \nabla F_m\left(x_k^{(m)}; \xi_k^{(m)}\right)]^\top \in \mathbb{R}^{m \times d} \end{aligned}$$

and calling $\mathbf{W}_k$ the mixing matrix, i.e. the double stochastic matrix encoding the communication in the k-th iteration of the algorithm, we can write the update rule in a matrix form as

$$\mathbf{X}_{k+1} = \mathbf{W}_k\left(\mathbf{X}_k - \gamma \nabla \mathbf{F}(\mathbf{X}_k)\right)$$

in the case where $H = 1$ local updates are performed. When $H \geq 1$ this update rule can be rewritten as:

$$\mathbf{X}_{k+1} = \mathbf{W}_k\left(\mathbf{X}_k - \gamma \sum_{q=1}^{H} \nabla \mathbf{F}(\mathbf{X}_k^q)\right)$$

where $\nabla \mathbf{F}\left(\mathbf{X}_k^q\right)$ is the gradient computed on the model after applying q local SGD updates to the model X_k.
By multiplying both sides of the update rule by the unitary vector scaled by the number of workers $\frac{1}{m}\mathbf{1} = \frac{1}{m}[1, \ldots, 1]$ we obtain

$$\frac{1}{m}\mathbf{1}\mathbf{X}_{k+1} = \frac{1}{m}\mathbf{1}\mathbf{W}_k \left(\mathbf{X}_k - \gamma \sum_{q=1}^{H} \nabla \mathbf{F}\left(\mathbf{X}_k^q\right)\right)$$

and by defining $x_k = \frac{1}{m}\mathbf{1}\mathbf{X}_k$, denoting the average model across all workers, and noticing that $\frac{1}{m}\mathbf{1}\mathbf{W}_k\mathbf{X}_k = x_k$, since $\mathbf{W}_k$ is a stochastic matrix, we obtain

$$x_{k+1} = x_k - \frac{\gamma}{m}\sum_{i=1}^{m}\sum_{q=1}^{H} \nabla F_i\left(x_k^{q,(i)}; \xi_k^{q,(i)}\right)$$

Recalling that the general update can be written as

$$x_{t,k+1} = x_{t,k} - \gamma d_{t,k}$$

we have that the average descent direction is $d_{t,k} = \frac{1}{m}\sum_{i=1}^{m}\sum_{q=1}^{H} \nabla F_i\left(x_k^{q,(i)}; \xi_k^{q,(i)}\right)$ hence

$$\mathbb{E}_{t,k}[d_{t,k}] = \frac{1}{m}\sum_{i=1}^{m}\sum_{q=1}^{H} \mathbb{E}_{t,k}[\nabla F_i\left(x_k^{q,(i)}; \xi_k^{q,(i)}\right)] = \frac{1}{m}\sum_{i=1}^{m}\sum_{q=1}^{H} \nabla f_i\left(x_k^{q,(i)}\right) \tag{3.1.1}$$

Now it is possible to compute, keeping in mind that mini batches are independent:

$$\begin{aligned}
\mathbb{E}\|d_{t,k} - \mathbb{E}_{t,k}[d_{t,k}]\|^2 &= \mathbb{E}\left\|\frac{1}{m}\sum_{i=1}^{m}\sum_{q=1}^{H}\left[\nabla F_i\left(x_k^{q,(i)}; \xi_k^{(i)}\right) - \nabla f_i\left(x_k^{q,(i)}\right)\right]\right\|^2 \\
&= \frac{1}{m^2}\sum_{i=1}^{m}\sum_{q=1}^{H}\underbrace{\mathbb{E}\left\|\nabla F_i\left(x_k^{q,(i)}; \xi_k^{q,(i)}\right) - \nabla f_i\left(x_k^{q,(i)}\right)\right\|^2}_{\leq \sigma^2 \text{ (bounded variance)}} \\
&\leq \frac{\sigma^2 H}{m}
\end{aligned}$$

Therefore the constant satisfying the third assumption is $V = \frac{\sigma^2 H}{m}$ which is a finite value since H, σ and m are finite quantities. An equivalent result can be proved for AD-PSGD with a similar procedure, with the only difference that X_k needs to keep into account the updates coming for delayed workers. How this type of analysis can be performed for asynchronous methods is presented in [Assran and Rabbat, 2020].

We can now report the main convergence result of SlowMO.

Theorem 3.1. *Suppose all workers start from the same initial point $x_{0,0}$ and the initial slow momentum is $u_0 = 0$. By setting α, β, $\gamma_t = \gamma$, τ and T such that $\frac{\alpha\gamma}{1-\beta} = \sqrt{\frac{m}{\tau T}}$ and the total iterations τT satisfies $\tau T \geq mL^2 \left(1 + \sqrt{3} \max \frac{3\tau(1-\beta-\alpha)}{\alpha}, \frac{4\tau\beta}{1-\beta}, 1\right)$, then under assumptions 3.1 to 3.3 the following result holds:*

$$\begin{aligned} \frac{1}{\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}\|\nabla f\left(x_{t,k}\right)\|^2 \leq\ & \frac{2\left(f\left(x_{0,0}\right) - f_{inf}\right) + mVL}{\sqrt{m\tau T}} \\ &+ \underbrace{\frac{1}{\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}\|\nabla f\left(x_{t,k}\right) - \mathbb{E}_{t,k}[d_{t,k}]\|^2}_{\textit{Effect of base optimizer}} \\ &+ \underbrace{\frac{4mVL^2\left(\tau-1\right)}{\tau T}\left(\frac{1-\beta}{\alpha} - 1\right)^2 + \frac{8mVL^2\tau}{\tau T}\frac{\beta^2}{1-\beta^2}}_{\textit{Effect of slow momentum}} \end{aligned} \tag{3.1.2}$$

where $f_{inf} = \inf_x f\left(x\right)$.

First we can notice that this result is consistent with the AR-SGD convergence. By taking $\tau = 1$, $\alpha = 1$, $\beta = 0$ and using SGD with learning rate γ as base optimizer all the terms on the right hand side of (3.1.2) vanish except the first. Since it is easy to see that for SGD one has $V = \sigma^2/m$ we recover the rate $\mathcal{O}\left(1/\sqrt{mT}\right)$.

The second term in (3.1.2) depends on the base optimizer and measures the bias between the full batch gradient $\nabla f\left(x_{t,k}\right)$ and the expected average across workers $\mathbb{E}_{t,k}[d_{t,k}]$. As an example we can compute this quantity for SwarmSGD. For simplicity this example will be carried out with $H = 1$.

Example. Recalling that the expected value of the average descent direction computed for SwarmSGD in (3.1.1) in the case $H = 1$ is

$$\mathbb{E}_{t,k}[d_{t,k}] = \frac{1}{m}\sum_{i=1}^{m}\nabla f_i\left(x_k^{(i)}\right)$$

we can write:

$$\frac{1}{\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}\|\nabla f\left(x_{t,k}\right)-\mathbb{E}_{t,k}[d_{t,k}]\|^2 = \frac{1}{\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}\left\|\nabla f\left(x_{k,t}\right)-\frac{1}{m}\sum_{i=1}^{m}\nabla f_i\left(x_{t,k}^{(i)}\right)\right\|^2$$
$$\leq \frac{1}{m\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\sum_{i=1}^{m}\mathbb{E}\underbrace{\left\|\nabla f\left(x_{k,t}\right)-\nabla f_i\left(x_{t,k}^{(i)}\right)\right\|^2}_{\leq L^2\|x_{t,k}-x_{t,k}^{(i)}\|^2}$$
$$\leq \frac{L^2}{m\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\sum_{i=1}^{m}\mathbb{E}\|x_{t,k}-x_{t,k}^{(i)}\|^2 \qquad (3.1.3)$$

where in the second step the following Jensen's inequality has been used:

$$\left\|\sum_{i=1}^{m}\frac{1}{m}[\nabla f\left(x_{k,t}\right)-\nabla f_i\left(x_{t,k}^{(i)}\right)]\right\|^2 \leq \sum_{i=1}^{m}\frac{1}{m}\left\|\nabla f\left(x_{k,t}\right)-\nabla f_i\left(x_{t,k}^{(i)}\right)\right\|^2$$

To prove the convergence of SwarmSGD, in [Nadiradze et al., 2019] a quantity Γ_t denoting the variance of a local model after t iterations is introduced:

$$\Gamma_t = \sum_{i=1}^{m}\|\mu_t - X_t^i\|^2$$

where $\mu_t = 1/m\sum_{i=1}^{m}X_t^i$ is the average of all the worker models. This can be rewritten in our notation as

$$\Gamma_{t,k} = \sum_{i=1}^{m}\|x_{t,k}-x_{t,k}^{(i)}\|^2 \qquad (3.1.4)$$

Under the the base assumptions 3.1 and 3.2 plus bounded second moment of the gradient, i.e. there exist a positive constant M^2 such that $\mathbb{E}\|\nabla F_i\left(x_{t,k};\xi_{t,k}^{(i)}\right)\|^2 \leq M^2$ it is possible to prove that the expectation of potential is bounded by [Nadiradze et al., 2019]

$$\mathbb{E}[\Gamma_{t,k}] \leq 2m\gamma^2 M^2 G \qquad (3.1.5)$$

where G is encoding the properties of the communication topology and $\gamma = m/\sqrt{\tau T}$ is the learning rate. Thus we can rewrite (3.1.3) as

$$\begin{aligned}\frac{1}{\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}\|\nabla f\left(x_{t,k}\right)-\mathbb{E}_{t,k}[d_{t,k}]\|^2 &\leq \frac{L^2}{m\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}[\Gamma_{t,k}]\\ &\leq \frac{L^2}{m\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}2m\gamma^2M^2G\\ &\leq \frac{2m^2L^2M^2G}{\tau T}\end{aligned}$$

Therefore, we can see that the term associated to the base optimizer vanishes with the rate of $1/\tau T$, as observed by the authors of SlowMo for other base optimizers. This term is also measuring how fast agents reach consensus: as expected, this rate depends on the number of workers m and the graph topology. For a larger number of workers or sparser graph communication this negatively impacts convergence.
Hence, we can write

$$\frac{1}{\tau T}\sum_{t=0}^{T-1}\sum_{k=0}^{\tau-1}\mathbb{E}\|\nabla f\left(x_{t,k}\right)\|^2=\mathcal{O}\left(\frac{1}{\sqrt{m\tau T}}\right)+\mathcal{O}\left(\frac{m^2}{\tau T}\right)+\mathcal{O}\left(\frac{1}{\tau T}\right) \tag{3.1.6}$$

3.2 Removing the average

Even if it is distributed over τ iterations, the AllReduce operation in Algorithm 4 is a blocking operation: if one of the workers is slow, this will translate into idle time of all other workers. Furthermore, as we have seen for SwarmSGD in equation (3.1.4), one of the key points in proving the convergence of decentralized methods is to prove that the agent's parameters are clustered around their mean. This is reasonable since we want ideally to reach consensus of the models, i.e. similar model's weights.
For this reason we could remove the exact average in line 6 of algorithm 4. This only makes sense for decentralized methods because in algorithm such as LocalSGD an exact average every τ local iterations is required, therefore in this case SlowMo is not adding any communication overhead.

Removing the exact average will impact also the momentum update since in this case it will not be the same for every agent. Every worker have its local momentum buffer u_t^i hence line 7 and 8 of algorithm 4 become:

$$u_{t+1}^{(i)} = \beta u_t^{(i)}+\frac{1}{\gamma_t}\left(x_{t,0}^{(i)}-x_{t,\tau}^{(i)}\right) \tag{3.2.1}$$

$$x_{t+1,0}^{(i)} = x_{t,0}^{(i)}-\alpha\gamma_t u_{t+1}^{(i)} \tag{3.2.2}$$

What we will observe with the numerical experiments is that even by loosing this requirement the accuracy of the method remains the same. This result is suggesting that slow momentum updates, and not the periodic synchronization of the models, is bringing the biggest contribution to the performance gain of SlowMo.

Chapter 4

Applications

In this chapter numerical results of the algorithms previously described will be presented. First the performance will be measured on standard benchmarks. Then a big data application related to high energy physics will be presented and the original work improved using the optimization methods described in this thesis. The original work carried out at CERN is described in a paper that we recently published [Migliorini et al., 2019].

4.1 Implementation

All the algorithms described in this work have been implemented using Pytorch and OpenMPI [Gabriel et al., 2004b]. MPI one-side primitives have been used to implement AD-PSGD since they allow one worker to access other workers' models via a remote memory access, without a blocking operation. This allows us to reach a higher throughput in systems where remote direct memory access is available, since a worker can perform a fast direct memory access in the memory of another worker.

For AD-PSGD the wait-free version of the method has been used, where communication and gradient computation are done in parallel in each worker. This ensures a better convergence with respect to wall time. To achieve this, two threads have been used, a communication and a computation thread, with a shared buffer. The shared buffer is used to pass model's parameters between the two threads. In this way, while the computation thread is computing the gradient with respect to the model pulled from the shared buffer, in the communication thread the average with other workers is happening and the new model is put in the shared buffer. Once the computation is done, the current model is pulled and updated using the gradient.

The tests were performed on Cloudveneto [Andreetto et al., 2019], an IaaS (infras-

tructure as a service) cloud hosted by the INFN units in Padova and Legnaro National Labs. Hosts are connected via a 10 Gbps ethernet links. A total of 4 Virtual machines have been used, each of them having 8 virtual CPUs and 16 GB of RAM. In order to have the same environment on every machine, OpenMPI and Pytorch have been compiled on one virtual machine, then the resulting image has been used to spawn the others. Data have been shared between workers using a Network File System. Since the resources available where limited, each vCPU has been seen as one worker in order to be able to test the work with $m > 5$ workers. If GPUs are available each one of them can be viewed as a worker. Furthermore it is common to have more than one GPU installed on the same machine.

This cloud provide a good environment to test robustness of the implemented methods given the low-bandwidth with respect to modern HPC infrastructures, which use InfiniBand networking. Furthermore, cloud hosts are shared by many users hence slowdown are happening frequently. Despite this facts it is worth studying performance of the methods in such harsh environment since it is accessible by more users with respect to HPC centers.

4.2 Standard Benchmark

The first tests were performed training LeNet-5 [LeCun et al., 1998] on the MNIST dataset [LeCun and Cortes, 2010]. This dataset consists of 60k training examples and a test set of 10k examples. Each example consists of a 28x28 pixel in grey scale image. In our case this dataset can be used to measure the impact of communication in the training time since the computing time of the model on CPU is comparable to the communication time. In the case of more complex models and datasets this statement is true when GPUs are used.
Then, the convergence has been test in a slightly more complex classification task by training ResNet-20 [He et al., 2016] on CIFAR-10 dataset [Krizhevsky, 2009]. CIFAR-10 consists of 60k 32x32 pixel colour images in 10 classes where images are equally spread across classes. The dataset is split into 50k training examples and 10k test images.

4.2.1 Scalability

The first quantity that we can measure is the distribution of the epoch time and compare it with the communication time. To do this we let the model train for 300 epochs on MNIST in order to obtain a sufficient number of samples to report the distribution. In this case we were not interested in the performance of the model, since such a small model in 300 epochs will most likely overfit. The communication

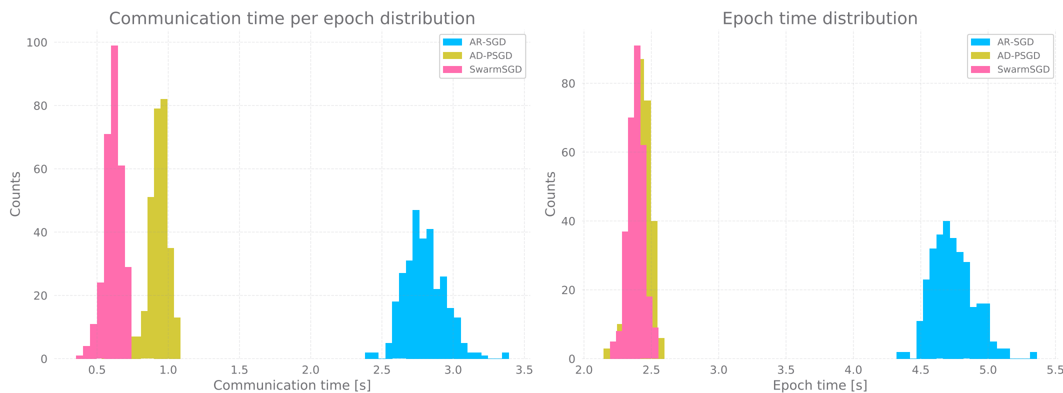


Figure 4.1: Distribution of the total communication time and wall-time per epoch. The times are measured by iterating 300 times over MNIST dataset using 16 workers.

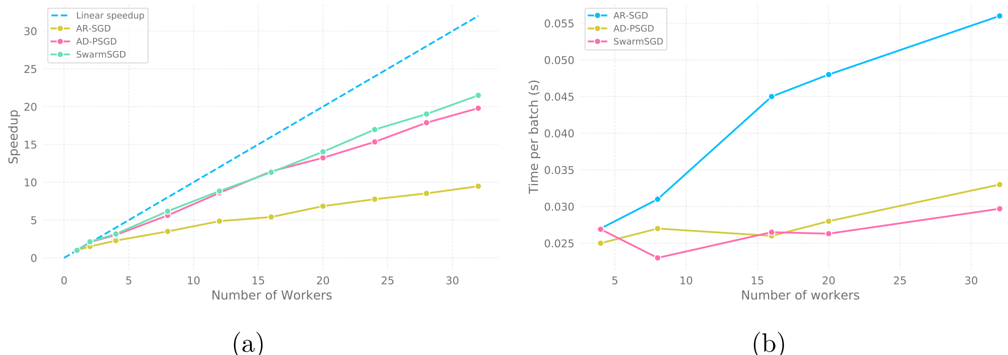
time is intended as the sum of all the communication times during one epoch. In this way it is possible to compare the resulting distributions. For this test 16 workers have been used and the timing measurements averaged. Furthermore, in SwarmSGD 3 local steps are performed before averaging models. The resulting distributions are shown in Figure 4.1.
The first thing that one can notice is that the communication time for both decentralized methods is much lower with respect to the one used by AR-SGD. This will translate into a lower wall time per epoch. In the communication time distribution we can see that the average time spent communicating by AD-PSGD is bigger than SwarmSGD as expected, because it is communicating every step. However, since this process happens in parallel with gradient computation, this time is hidden and the resulting time per epoch is comparable with the one of SwarmSGD. In theory, thanks to the wait-free implementation, this should be a bit smaller. The reason this is note the caseresides in how this approach is implemented, which can be improved.

Furthermore, the variance for AR-SGD is twice the one measured for the other two methods. This is because each iteration is influenced by all the workers therefore in noisy environments such as a shared cloud this variability is expected to be high.

We can also measure the speedup of each algorithm, defined as

$$\text{speedup}\left(n\right) = \frac{t\left(1\right)}{t\left(n\right)}$$

where $t\left(n\right)$ represents the training time when using n parallel workers. In other words it is measuring the ratio between the time took to run an algorithm using one process and the computational time for running the same algorithm using n



(a) (b)

Figure 4.2: (a): Measured speedup. (b): Average time per batch.

processors. Ideally we would like methods with speedup $= n$, which means that every worker is contributing with all its computational power. However, such scenario is often far from reality due many practical problems. In our case $t(1)$ is the time taken by a single worker to complete the training using the standard sequential SGD, where no communication is involved. The measured speedup versus the ideal one for each algorithm is reported in figure 4.2(a). We can see that both decentralized methods scale better than AR-SGD, leading to a twice bigger speedup on average with respect to the latter. In theory both decentralized algorithms should scale better since the communication time doesn't depend on the number of workers. In practice this quantity is strongly influenced by the hardware where the training is running and how the algorithm is implemented. This can be seen in 4.2(b) where the average time per batch is plotted as function of the number of workers. In the case of decentralized algorithms this time remains almost constant, whereas it increases for AR-SGD. Despite that, we are obtaining similar scaling results as the ones presented in AD-PSGD and SwarmSGD papers in the case of 10 Gbps network.

4.2.2 Convergence

We can now focus on the convergence with respect to the number of epochs and time. First we can measure if adding SlowMO to the decentralized methods lead to better performance. For the rest of this work we will use the version without the exact average, since in this way there is not a tangible performance degradation. An example of SlowMo is reported in Figure 4.3. In both cases the test accuracy increases while maintaining the same computational time, since the computational overhead added by the momentum update is minimal.

In this case the hyperparameters used for the base optimizer are the same as the ones used without slow-momentum. Also, each local step of the two decentralized algorithms is performed using mini-batch SGD with classical momentum. The two

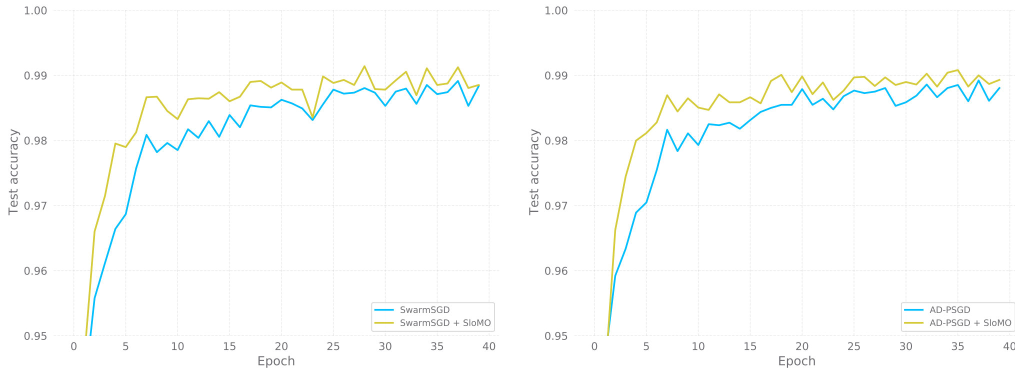


Figure 4.3: Effect of SLOWMO on AD-PSGD and SWARMSGD on MNIST dataset.

additional hyperparameters are the number of steps performed by the base optimizer τ and the slow-momentum factor β: in this case we choose $\tau = 12$ and $\beta = 0.7$, similarly to the values used in the original paper.

The convergence with respect to time and number of epochs is presented in Figure 4.4. Both SWARMSGD and AD-PSGD combined with SLOWMO reach similar performance as AR-SGD. A mini-batch of 128 examples has been used for every worker, as for the sequential case. This is equivalent to an overall batch size of $16 \cdot 128 = 2048$ in the case of AR-SGD leading to a poor test accuracy. To compensate for this we adopted one of the techniques presented in [Goyal et al., 2017], that is scaling the learning rate by the number of parallel workers. This problem is not observed for the decentralized algorithm since only averages between two peers are performed, therefore the same parameters, e.g. the learning rate, ca be used without any modification.

To train RESNET-20 on CIFAR-10, the standard training hyperparameters for this model [Lian et al., 2017b] are used: A batch size of 32 per worker, learning rate starting from 0.1 and decaying by a factor of 10 at the 81st and 121st epochs, a momentum factor of 0.9 and weight decay (ℓ_2 penalty) of 10^{-4}. The model has been trained for a total 150 epochs. Also in this case the AR-SGD's learning rate had to be increased in order to converge to comparable results as the other two methods. For SLOWMO a slow momentum factor of $\beta = 0.6$ and $\tau = 12$ local steps have been used leaving base optimizer hyperparameters unchanged. Figure 4.5 shows that with respect to the number of epochs all the methods converge similarly. Since RESNET-20 is a relatively small model, around 1MB, the impact of communication in this training is much smaller with respect to the computational one: with 16 workers completing one epoch takes around 40s where only 6s are spent communicating. For

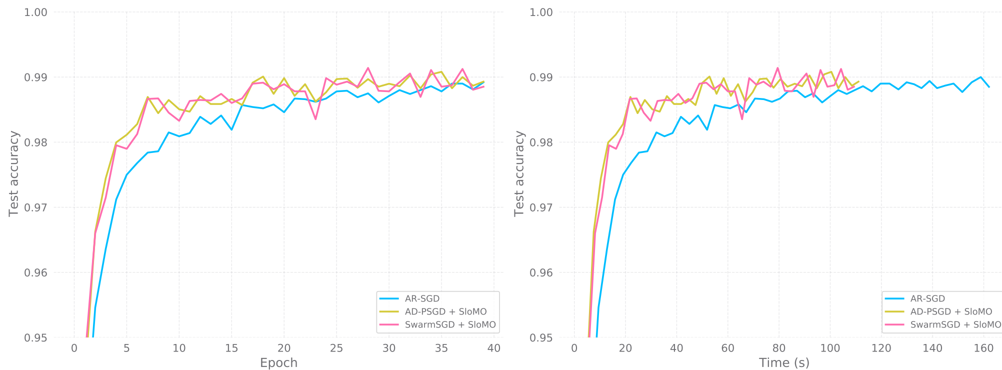


Figure 4.4: Convergence with respect to number of epochs and wall time on MNIST for fixed number of workers.

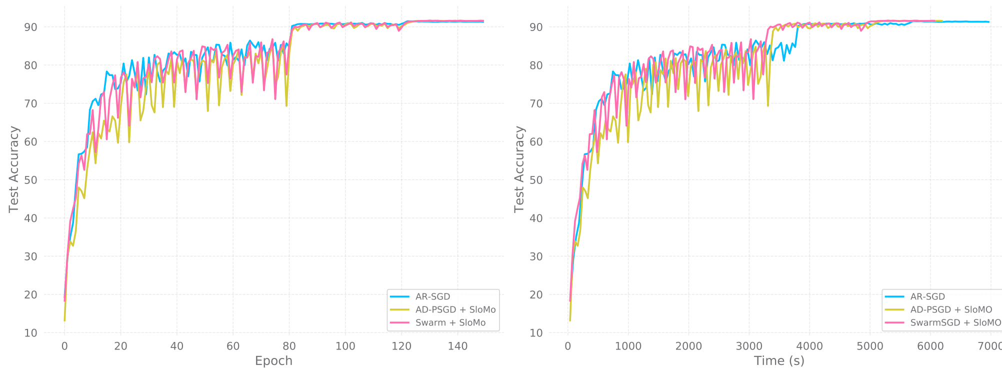


Figure 4.5: Convergence of ResNet-20 on CIFAR-10 with respect to the number of epochs.

this reason in the convergence with respect to time a big improvement as the one seen in Figure 4.4 is obtained. This would change if gradients where computed on a GPU since the computational time would be an order of magnitude slower. But also in this scenario the convergence with respect to the number of workers would be the same, hence the results obtained using SlowMo would be equivalent. The comparison of the results with and without slow momentum is shown in Table 4.1. SlowMo without average consistently improved the results of the baseline algorithm without adding any overhead. There is not a substantial difference with respect to the version adopting the global average. Furthermore it is robust with respect to the slow momentum factor: choosing $\beta \in 0.5, 0.6, 0.7, 0.8$ changes only slightly the performance in all the tests we performed. As a reference, the accuracy for ResNet-20 on CIFAR-10 in the original paper is 91.25% and it is possible to obtain a score of 92.16%[1] in the sequential case. The results obtained in our case are

[1] https://keras.io/examples/cifar10_resnet/

Algorithm	Validation accuracy		
	Baseline	SlowMo	SlowMo no avg.
AR-SGD	91.29%	-	-
AD-PSGD	91.32%	91.60%	91.58%
SwarmSGD	91.30%	91.59%	91.61%

Table 4.1: Comparison of the test accuracy for the ResNet-20 on CIFAR10. The column baseline reports the accuracy of the algorithm while SlowMO the accuracy when slow momentum is applied. SlowMo no average represent the case when the global average is removed.

in line with the ones in the literature, but with a more careful tuning of the training hyperparameters they can be further improved. Indeed in both SwarmSGD and AD-PSGD papers, the score of the base optimizer is around 91.70. Hence, it is reasonable to think that by combining a fine tuned training procedure for the base optimizer with SlowMo these results could be further improved.

4.3 High Energy Physics

In this section, a brief description of the work carried out during my internship at CERN will be provided. The aim of the project was to build a scalable machine learning pipeline, from the data ingestion to the training of a neural network, using modern big data tools. A detailed description of the pipeline and its performance can be found in [Migliorini et al., 2019].
After a brief description of the pipeline developed there, we will focus on how the methods studied in this thesis can be used to train those models. While the pipeline was run using CERN computing resources, the training with the new methods has been performed on Cloud Veneto.

4.3.1 Problem description

High energy physics (HEP) is a data-intensive domain: chasing extremely rare physics processes requires producing, managing and analyzing large amounts of complex data. Data are produced by detectors, where proton-proton collisions happens. Collisions happen every 25ns (40MHz) and given that the size of an event recorded by all the electronics is around 2MB this will produce almost 100TB/s. Storing all produced events would be too costly and not useful since the interesting events are only a small fraction of them. For this reason, right after the production, an event goes trough a chain of online filters with the function of deciding whether or not an event is interesting. Given the high throughput of the detector, having only one filtering stage where a difficult classification task is performed with only raw

informations from electronics would be impossible. For this reason, for example in the CMS experiment[2], there is a chain of two sequential filters. The first one, called *Level 1 trigger*, uses only low level information from the electronics to provide a first classification of the event: if it is not interesting it gets discarded whereas if it has some characteristics indicating that is interesting it is sent to the next filter in the chain. Having to operate at 40MHz such filters need to be extremely fast: a large number of false positive is produced, but most importantly the number of true negatives is reduced. Indeed, at this stage the rate is decreased from 40MHz to 100kHz. This trigger is implemented in electronic boards near the detector.
Events passing the *Level 1 trigger* are moved into the *High Level Trigger* (HLT). At this stage the informations coming from all sub-detectors are put together to generate a list of particles present in the events and to provide a better classification. In the HLT the rate is further reduced from 100kHz to 1kHz and the resulting events are stored on disk permanently. Since the size of an event is 1MB this translates into 1GB/s of data stored permanently to disk. The case study considered in this work is to improve the filtering at HLT level for a particular type of event with the help of deep learning. In particular, we develop the classification described in [Nguyen et al., 2019]. We studied how modern big data tools can help to perform such analysis, starting from reading the data up to the training the classifier.

The dataset used to perform this analysis consists of 55 million simulated detector and consequent trigger responses to three particular classes passing the same preliminary selection in the filters chain inside HLT (e.g. requiring the presence in the event of at least one electron with a certain energy): QCD, $W + jets$ and $t\bar{t}$. QCD is the predominant class in all the events and ideally we would like to remove all this kind of events at L1 trigger since they are the main background. However this is the most frequent while the most interesting one, $t\bar{t}$, is the less frequent: in this dataset there is a $t\bar{t}$ event every 100 QCD events. The other class is less frequent than QCD but more frequent than $t\bar{t}$. For this reason having a classifier capable of having a good accuracy in the classification allows us to store only the events that are interesting for further studies, thus avoiding to waste disk space with events that will be discarded in the analysis. This is a key point since in the next upgrade of CERN main accelerator, the Large Hadron Collider (LHC), the amount of data produced will increase by one order of magnitude.

Now a brief description of the pipeline will be provided. The main computational backend used is Apache Spark [Zaharia et al., 2016], which represents a promising tool to extend traditional HEP analysis tools. Spark is one of the most popular analytics engine for big data: thanks to its mature API and engine it allows us to

[2] https://cms.cern/detector

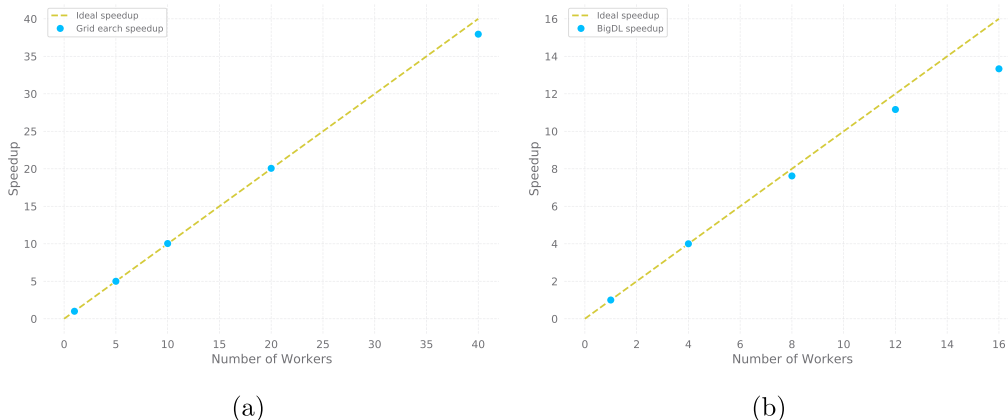
perform data processing, explorative data analysis and machine learning on large datasets via an efficient usage of cluster resources. The pipeline can be logically divided in four steps.

Data Ingestion The dataset is stored in the storage system used at CERN for physics data called CERN EOS. Furthermore data are stored in a common format of HEP called ROOT. Hence, if we are interested in using standard big data tools to perform the analysis, enabling these tools to understand physics data format and reading from storage such as EOS it is of paramount importance. For this reason two libraries enabling spark to read from EOS and to create dataframes from ROOT files have been developed. The size of this dataset is of the order of 10TB. However, while developing this libraries, tests of data processing at the scale of 1PB were performed, proving that this system scales.

Feature Engineering In this step the ingested data are processed to compute the quantities that will be used to train the models. Following the work described in [Nguyen et al., 2019] two datasets have been produced and additional standard filters to reduce the number of events have been used. In this way we end up working with around 5 million events instead of 50 million. Then two datasets are built: the Low Level Features and High Level Features datasets. Each event of low level feature dataset consists of the list of the 801 most energetic particles present in the original event. To have an order, the particles in the list are sorted in terms of increasing distance from a particular particle which is present in all of these events. Each particle is described by 19 features indicating its position in the space, kinematics and type. Therefore each event can be viewed as a matrix with shape 801x19. From the low level features a set of 14 high level features can be computed for each event. These additional features are motivated by the physics of the process under consideration. Such features, e.g. the angle between two particular particles or the total energy of a specific subset of particles are known to be good discriminants between the signal and background of this type of processes. At this step any kind of classifier can be used, e.g. decision trees. However, a lot of domain knowledge is necessary to build these features and an additional analysis is required. For this reason it is interesting to see if the same results can be obtained starting from the low level features, since this would avoid us the extra step of crafting the high level features.

The resulting datasets have been split in train and test datasets and saved as parquet files[3], a common data format for big data, in the Hadoop distributed file system of the cluster. At this stage, the size of the training dataset on disk is around 500GB.

[3] https://parquet.apache.org/



(a) (b)

Figure 4.6: Measured speedup for: (a) grid search, (b) training using BigDL.

Parameter Tuning At this stage we performed a grid search on a subset of the dataset to find the best model architecture, i.e. the one with the better tradeoff between model size and accuracy. Model size and inference speed is an important parameter since such model will be possibly run inside the HLT and therefore there are time constraints it must satisfy. For the grid search spark has been used as a scheduler, i.e. the master kept track of the map architecture-score and scheduled the training on the workers. Since the training in every worker is independent from the one in the others, the time required to complete the search scales with the number of workers, as shown in figure 4.6(a).

Model Training To perform the distributed training different solutions have been tested. First, thanks to a collaboration with intel, we trained the models using Intel BigDL [Dai et al., 2019], a library built on top of Spark using spark primitives to implement AR-SGD. This solution runs on CPU and uses Intel MKL to accelerate the training on CPU. For this scaling test each worker had 16 cores and 16GB of RAM. As shown in figure 4.6(b), BigDL seems to scale well in the regime tested. However, also in this case the problems of synchronous SGD have been observed: lot of time is spent waiting for slow workers. This effect had a bigger impact in our test when other jobs where running in the cluster, since occasional node slowdowns were frequent.
Tensorflow has also been used to perform distributed training, showing similar scaling properties as BigDL. Tensorflow has also been used to perform the training using up to 10 GPUs on Oracle cloud. Again, also in tensorflow AR-SGD is implemented as distribution strategy.

Since BigDL is implemented on top of Spark it was possible to read the parquet

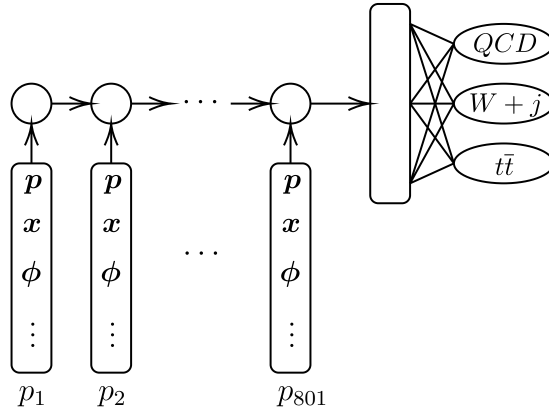


Figure 4.7: Visual representation of the particle-sequence classifier. Each of the input particles is described by 19 features such as momentum $\vec{p}$ and position $\vec{x}$. The fixed-size encoding of the last recurrent unit is fed into a fully connected layer with the role of classifier. The sequence of particles has always the same order, i.e. p_1 will always be the same type of particle and the others are ordered with increasing distance from it.

files produced during the feature engineering step while for tensorflow an additional step was required to convert the parquet into TFRecords. To train models using the methods developed in this thesis, Petastorm[4] was used. Petastorm is a tool developed at Uber that allows one to read parquet files into common deep learning frameworks such as Pytorch. This library allows to avoid the extra step of converting parquet files into other formats such as HDF5. Later in this chapter the results obtained from the training with the algorithms described in this thesis will be presented.

4.3.2 Models

In this section we describe two models that have been trained on the two datasets; The high level features and low level features datasets.

HLF classifier The High Level Features classifier (HLF) is a fully connected feed-forward deep neural network taking as input the vector of 14 high level features. More precisely it consists of three hidden layers with 50, 20 and 3 nodes. In each hidden node ReLu activation is used, while softmax is used for the last three neurons. This small model is capable of obtaining good performance thanks to the knowledge used to craft the 14 high level features. Other type of classifiers have been tested, such as boosted decision trees, and the performance were similar.

[4] https://github.com/uber/petastorm

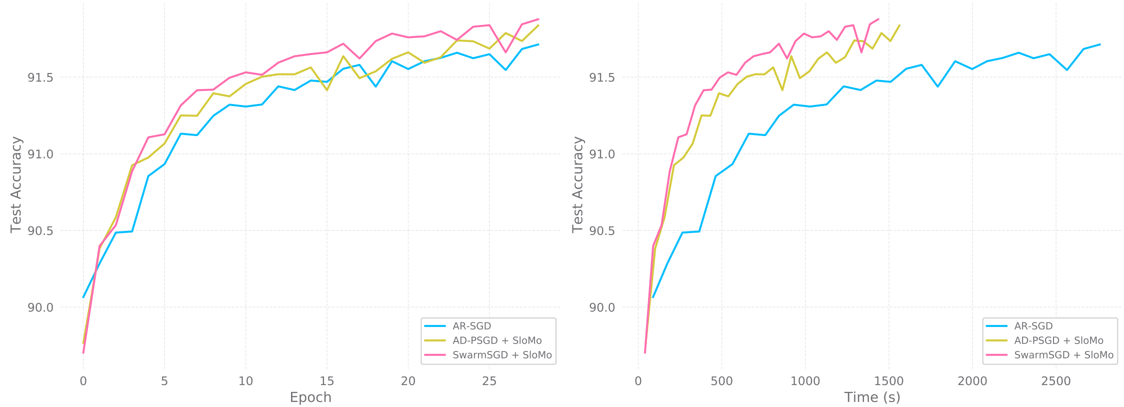


Figure 4.8: Validation accuracy of the High Level Features classifier plotted in function of number of epochs and training time.

Particle-sequence classifier The particle sequence classifier is a recurrent neural network taking as input the 801 particles contained in each event. The output of the recurrent network is fed into a fully connected layer with three softmax activated nodes. Gated Recurrent Units (GRU) have been used to aggregate the input sequence of particles into a fixed size encoding. The internal width of the recurrent layers is 50. The zero-padded entries in the particle list are skipped using a masking layer before the recurrent network.

A visual representation of this network is shown in Figure 4.7. Each input particle is composed by the 19 features describing its kinematics and type. These features are produced directly by the detector, i.e. they are not manually crafted as the high level features. In this sense, the recurrent network can be seen as a feature extractor and the dense layer as a classifier.

4.3.3 Results

Both High Level Features and Particle-sequence classifiers have been trained using Adam [Kingma and Ba, 2014] optimizer to compute the descent directions instead of SGD, which led to better results. The training of the HLF classifier has been performed on the entire dataset, whereas the particle-sequence classifier has been trained only on a subset of 50k examples due to time and disk space constraints.

The convergence with respect to the number of epochs and wall time for the High Level Feature classifier is shown in Figure 4.8. In this case the decentralized methods outperform AR-SGD both in terms of converge time and final accuracy. The lower training time required to compute a fixed number of epochs (Figure 4.8) is

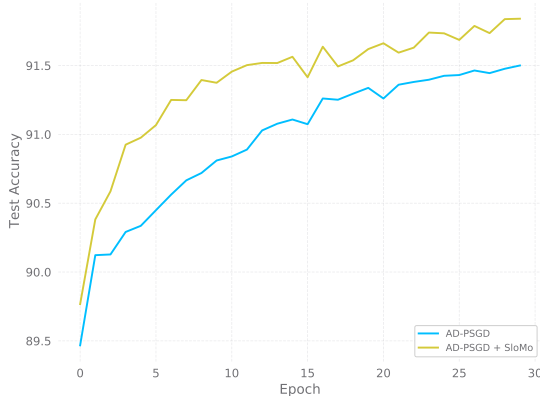


Figure 4.9: Effect of SlowMo on AD-PSGD and SwarmSGD on HLF dataset.

expected, given the small model that we are training, therefore the impact of communication time in this case is very high since synchronization happens frequently with respect to computing time. Furthermore, the better final accuracy is mainly due to SlowMo(Figure 4.9): Both methods without SlowMo would have slightly worst performance with respect to AR-SGD. A numerical comparison is reported is reported in Table 4.2.

Algorithm	Validation accuracy		
	Baseline	SlowMo	SlowMo no avg.
AR-SGD	91.68%	-	-
AD-PSGD	91.50%	91.86%	91.83%
SwarmSGD	91.54%	91.88%	91.87%

Table 4.2: Comparison of the test accuracy for the High Level Features classifier. The column baseline reports the accuracy of the algorithm, while SlowMo corresponds to the accuracy when slow momentum is applied. SlowMo no average represents the case when the global average is removed.

To have a comparison, the accuracy obtained in the work at CERN for this classifier both in the sequential and distributed training using Tensorflow and BigDL is similar to the one obtained with AR-SGD.

The convergence result for Particle-sequence classifier are presented in Figure 4.10. In this case the speedup is less evident with respect to the previous case. The reason for this is that the computing time is much bigger than the communication time, as it happened for the training of ResNet on CIFAR10.

Furthermore, the addition of SlowMo is not as evident as in the case of the HLF

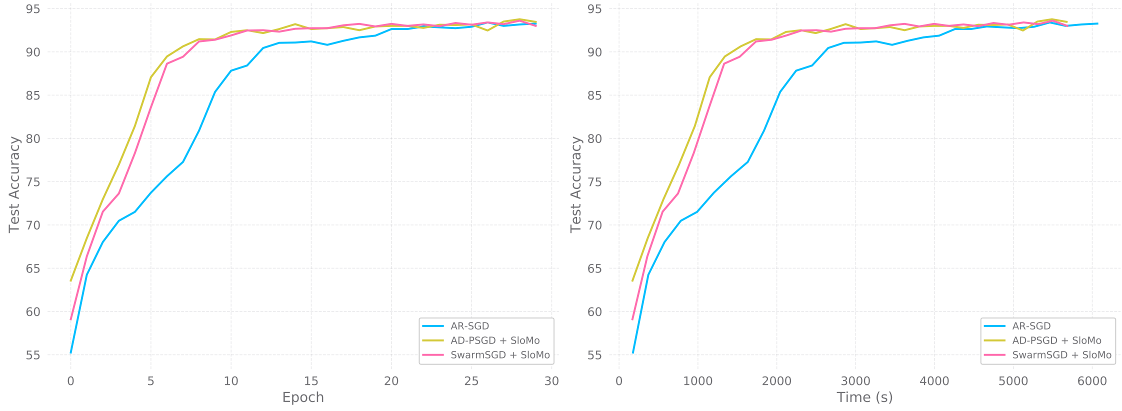


Figure 4.10: Validation accuracy of the Particle-sequence classifier plotted in function of number of epochs and training time.

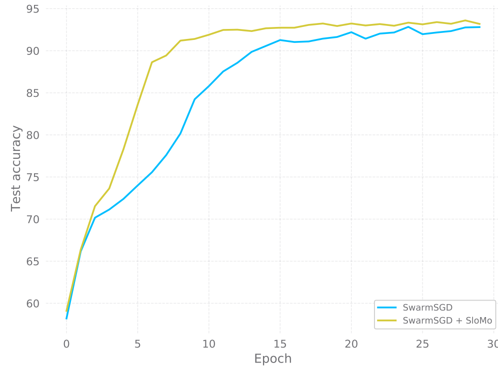


Figure 4.11: Effect of SlowMo on the training of the particle sequence classifier.

classifier. Indeed all the methods have more or less the same final accuracy on the validation dataset. The main difference seems to be in the first epochs of the training, where the two decentralized methods seems to reach the final accuracy faster. Again, this improvement is due to SlowMo, as shown in Figure 4.11. Without momentum, the training accuracy for SwarmSGD and AD-PSGD would have the same trend as in the case of AR-SGD. This initial “slow down” disappears if the learning rate is increased. However, this would make the loss diverge in later stages and to avoid this a careful tuning of the learning rate scheduling is required. With the addition of slow momentum this effect has not been observed.

In the reduced version of the dataset all methods converge to the same final accuracy, which is 94% (Table 4.3). If trained on the full dataset, this model will reach an accuracy of 96%.

Algorithm	Validation accuracy		
	Baseline	SlowMo	SlowMo no avg.
AR-SGD	94.06%	-	-
AD-PSGD	93.89%	94.11%	94.08%
SwarmSGD	93.97%	94.13%	94.02%

Table 4.3: Comparison of the test accuracy for the Low Level Features classifier. The column baseline reports the accuracy of the algorithm while SlowMo the accuracy when slow momentum is applied. SlowMo no average represents the case when the global average is removed.

It is reasonable to think that the validation loss is saturating at that value due to the lack of training examples. If this is the case, methods with SlowMo are reaching the "final accuracy" faster than SGD. For this reason, if trained on the full dataset they could lead to better results faster, as happened for HLF classifier.

Chapter 5

Conclusions

In this thesis we studied some of the algorithms available in the distributed optimization landscape, with a focus on deep learning. We started in Chapter 2 by introducing the standard way of parallelizing stochastic gradient descent while highlighting its current limitations. In particular, synchronous parallel SGD has two main problems: the accuracy degradation observed when training with large mini-batches, i.e. large number of workers, and the communication inefficiency on slow or busy networks.
Starting from this, decentralized methods have been introduced since they are communication efficient and are not affected by large batch problems. We studied two novel algorithms with promising performance, SwarmSGD and AD-PSGD and compared them with standard SGD. We showed that they are capable of recovering SGD accuracy while being faster. Furthermore, since they involve communications only between pairs of node, these methods present a good scalability.

In Chapter 3 a method inspired by the idea of momentum, SlowMo, is introduced. This algorithm allows to improve the performance of a base optimizer, which in our case is one of the two decentralized methods mentioned before. In SlowMo a periodic global synchronization is required. However, starting from the idea presented in the paper introducing slow momentum, we removed the average and observed that the performance were similar. This further validates the idea that the periodic global average is not required. The reason for this could be due to the fact that workers reach the consensus fast and the models are all close to their mean.

In Chapter 4 the performance of the algorithms in standard benchmarks are presented. Then, an extension of a work performed in high energy physics has been presented. In all the benchmarks, the decentralized algorithms combined with momentum obtain better or equivalent scores as standard parallel SGD, while converging faster is terms of wall time.

Therefore, in the tests performed in this thesis we observed that is possible to recover SGD performance in the distributed case while being robust to an heterogeneous environment. Moreover, slow momentum increased the final accuracy in all the benchmarks.

Future works include a better understanding of the momentum for distributed optimizer and proving that for SlowMo the periodic average is not required to get the convergence of the method. Moreover, new tests can be performed using other datasets, models and base optimizers, to reinforce the claim that it is consistently improving the performance. The communication can be further improved by using compressed communication schemes, where model update communicated between workers are compressed, e.g. quantized or sparsified [Koloskova et al., 2019].

List of Figures

1.1 Example of model parallelism (a) and data parallelism (b): in the former the model is divided into four parts distributed on four different machines, in the later the same model is replicated in four different machines. 2

2.1 Strategies for worker synchronization in parallel SGD: (a) Parameter server strategy for centralized-parallel SGD(C-PSGD), (b) Gradient aggregation based on a Ring-AllReduce operation (AR-SGD). . . 8

4.1 Distribution of the total communication time and wall-time per epoch. The times are measured by iterating 300 times over MNIST dataset using 16 workers. 27
4.2 (a): Measured speedup. (b): Average time per batch. 28
4.3 Effect of SlowMo on AD-PSGD and SwarmSGD on MNIST dataset. 29
4.4 Convergence with respect to number of epochs and wall time on MNIST for fixed number of workers. 30
4.5 Convergence of ResNet-20 on CIFAR-10 with respect to the number of epochs. 30
4.6 Measured speedup for: (a) grid search, (b) training using BigDL. . . . 34
4.7 Visual representation of the particle-sequence classifier. Each of the input particles is described by 19 features such as momentum $\vec{p}$ and position $\vec{x}$. The fixed-size encoding of the last recurrent unit is fed into a fully connected layer with the role of classifier. The sequence of particles has always the same order, i.e. p_1 will always be the same type of particle and the others are ordered with increasing distance from it. 35
4.8 Validation accuracy of the High Level Features classifier plotted in function of number of epochs and training time. 36
4.9 Effect of SlowMo on AD-PSGD and SwarmSGD on HLF dataset. 37
4.10 Validation accuracy of the Particle-sequence classifier plotted in function of number of epochs and training time. 38

4.11 Effect of SlowMo on the training of the particle sequence classifier. 38

List of Tables

4.1 Comparison of the test accuracy for the ResNet-20 on CIFAR10. The column baseline reports the accuracy of the algorithm while SlowMO the accuracy when slow momentum is applied. SlowMo no average represent the case when the global average is removed. . . 31
4.2 Comparison of the test accuracy for the High Level Features classifier. The column baseline reports the accuracy of the algorithm, while SlowMo corresponds to the accuracy when slow momentum is applied. SlowMo no average represents the case when the global average is removed. 37
4.3 Comparison of the test accuracy for the Low Level Features classifier. The column baseline reports the accuracy of the algorithm while SlowMo the accuracy when slow momentum is applied. SlowMo no average represents the case when the global average is removed. . . 39

Bibliography

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

[Andreetto et al., 2019] Andreetto, P., Chiarello, F., Costa, F., Crescente, A., Fantinel, S., Fanzago, F., Konomi, E., Mazzon, P., Menguzzato, M., Segatta, M., Sella, G., Sgaravatto, M., Traldi, S., Verlato, M., and Zangrando, L. (2019). Merging openstack-based private clouds: the case of cloudveneto.it. *EPJ Web of Conferences*, 214:07010.

[Apollinari et al., 2015] Apollinari, G., Béjar Alonso, I., Brüning, O., Lamont, M., and Rossi, L. (2015). High-luminosity large hadron collider (hl-lhc): Preliminary design report. Technical report, Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States).

[Assran et al., 2018] Assran, M., Loizou, N., Ballas, N., and Rabbat, M. (2018). Stochastic gradient push for distributed deep learning. *arXiv preprint arXiv:1811.10792*.

[Assran and Rabbat, 2020] Assran, M. and Rabbat, M. (2020). Asynchronous gradient-push. *IEEE Transactions on Automatic Control*.

[Chen and Huo, 2016] Chen, K. and Huo, Q. (2016). Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *2016 ieee international conference on acoustics, speech and signal processing (icassp)*, pages 5880–5884. IEEE.

[Cireşan et al., 2010] Cireşan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220.

[Dai et al., 2019] Dai, J. J., Wang, Y., Qiu, X., Ding, D., Zhang, Y., Wang, Y., Jia, X., Zhang, C. L., Wan, Y., Li, Z., et al. (2019). Bigdl: A distributed deep

learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60.

[Dean et al., 2012] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Gabriel et al., 2004a] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004a). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Gabriel et al., 2004b] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004b). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Ghadimi and Lan, 2013] Ghadimi, S. and Lan, G. (2013). Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368.

[Ghadimi et al., 2016] Ghadimi, S., Lan, G., and Zhang, H. (2016). Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. *Mathematical Programming*, 155(1-2):267–305.

[Gibiansky., 2017] Gibiansky., A. (2017). Bringing HPC Techniques to Deep Learning.

[Goyal et al., 2017] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Keskar et al., 2016] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Koloskova et al., 2019] Koloskova, A., Stich, S. U., and Jaggi, M. (2019). Decentralized stochastic optimization and gossip algorithms with compressed communication. *arXiv preprint arXiv:1902.00340*.

[Krizhevsky, 2009] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[LeCun and Cortes, 2010] LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

[Lian et al., 2015] Lian, X., Huang, Y., Li, Y., and Liu, J. (2015). Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745.

[Lian et al., 2017a] Lian, X., Zhang, C., Zhang, H., Hsieh, C.-J., Zhang, W., and Liu, J. (2017a). Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340.

[Lian et al., 2017b] Lian, X., Zhang, W., Zhang, C., and Liu, J. (2017b). Asynchronous decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1710.06952*.

[Lin et al., 2018] Lin, T., Stich, S. U., Patel, K. K., and Jaggi, M. (2018). Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217*.

[Migliorini et al., 2019] Migliorini, M., Castellotti, R., Canali, L., and Zanetti, M. (2019). Machine learning pipelines with modern big datatools for high energy physics. *arXiv preprint arXiv:1909.10389*.

[Nadiradze et al., 2019] Nadiradze, G., Sabour, A., Alistarh, D., Sharma, A., Markov, I., and Aksenov, V. (2019). SwarmSGD: Scalable Decentralized SGD with Local Updates. *arXiv e-prints*, page arXiv:1910.12308.

[Nguyen et al., 2019] Nguyen, T. Q., Weitekamp, D., Anderson, D., Castello, R., Cerri, O., Pierini, M., Spiropulu, M., and Vlimant, J.-R. (2019). Topology classification with deep learning to improve real-time event selection at the lhc. *Computing and Software for Big Science*, 3(1):12.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Patarasuk and Yuan, 2009] Patarasuk, P. and Yuan, X. (2009). Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124.

[Seide and Agarwal, 2016] Seide, F. and Agarwal, A. (2016). Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135.

[Sergeev and Del Balso, 2018] Sergeev, A. and Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799.*

[Sun et al., 2017] Sun, C., Shrivastava, A., Singh, S., and Gupta, A. (2017). Revisiting unreasonable effectiveness of data in deep learning era. In *The IEEE International Conference on Computer Vision (ICCV).*

[Sutskever et al., 2013] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147.

[Wang and Joshi, 2018] Wang, J. and Joshi, G. (2018). Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313.*

[Wang et al., 2019] Wang, J., Tantia, V., Ballas, N., and Rabbat, M. (2019). Slowmo: Improving communication-efficient distributed sgd with slow momentum. *arXiv preprint arXiv:1910.00643.*

[Zaharia et al., 2016] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65.

[Zhang et al., 2019] Zhang, M., Lucas, J., Ba, J., and Hinton, G. E. (2019). Lookahead optimizer: k steps forward, 1 step back. In *Advances in Neural Information Processing Systems*, pages 9593–9604.