Master in High Performance Computing

Distributed Systems for Neural Network models

Supervisor:
Stefano Cozzini

Candidate:
Luca Ciuffreda

4th edition
2017–2018

Acknowledgements

I want to thank Stefano Cozzini and CNR-IOM for sponsoring this work and proposing such an interesting and challenging project. I thank everybody in Stefano's group who have helped for any technical or scientific question.

I thank also everybody who have made this master a wonderful experience: from the lecturers, to the *tutors* and *collegues*, who have become just *friends* by now. In particular I am grateful to Piero for sharing ideas about our projects and interesting views about every subject we have been able to discuss.

Contents

Chapter 0

Introduction

The activity that led to this thesis is developed as part of the NFFA-EUROPE project and is funded by CNR-IOM. In the following we briefly describe this context and the scientific problems that have been faced in the past within the project. This chapter is organized as follows: we first introduce the NFFA-EUROPE project and then discuss the main tool used to implement the classification service, convolutional neural networks. Finally we give an overview of the rest of the thesis.

0.1 The NFFA-EUROPE project

The NFFA-EUROPE project [1] is a Horizon 2020 activity which aims at the integration and developement of a unified and multidisciplinary research infrastructure for the world of nanoscience[1]. One of the goals is to create an Information and Data management Repository Platform (IDRP), the creation of which is coordinated by CNR-IOM [2]. The present work is part of this activity. In addition to data storage and metadata registration, the data repository aims to offer also analysis tools for raw data. The first of these services is a scanning electron microscope (SEM) image classificator, which has been developed in collaboration with CNR-IOM.

The CNR-IOM institute [3] provided us with a set of images collected by different research groups, for a total of 146916 unique images in the *Tagged Image File Format* format (`tif`), mostly of size 1024×768 pixels (and 3 color channels). Among them, 18577 have been manually labelled to create a full dataset, called `dataset1`, for the classifier: labels associate each image to one out of ten categories as in Fig.1.

[1] See [7] for more information.

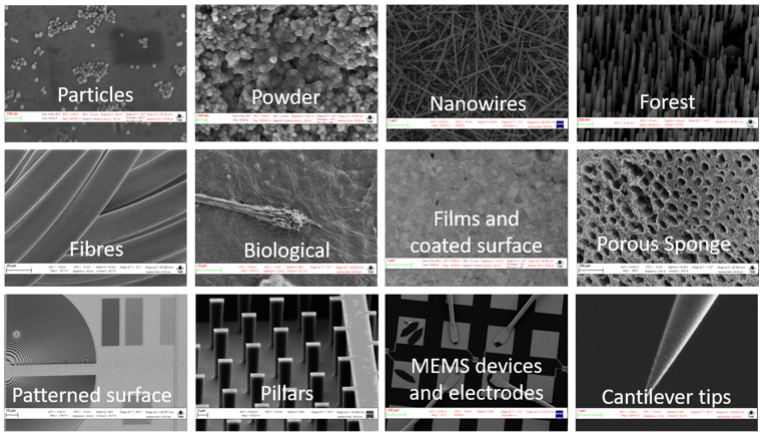


Figure 1: Classification labels of the SEM dataset from [7]. The classes used for the final classification *Particles*, *Fibres*, *Biological*, *Patterned surface*, *Cantilever Tips*, *Nanowires*, *Powder*, *MEMS devices and electrodes*, *Porous Sponge* and *Films and Coated Surface*. The image here shows 12 categories since the first analysis in [7] was made over these classes.

The published `dataset1` [6] collection is ultimately split in two sub-datasets: a training set, which contains 90% of the images (16724), and a test/validation set, with the remaining 1853 images. Details about the labelling and the selection criteria can be found in [7]. During our project we organized a central repository with the full set of SEM images available, collecting information such as the scale of the scanning in order to be able to perform finer analysis over selected subsets of the images. Detailed information about this work have been reported by Coronica in his 2018 MHPC work [9].

The classification service is based on the use of convolutional neural network models, which represent the state of the art in computer vision and image classification. For this reason in the next section we introduce the main features of convolutional neural networks and briefly describe the pscific architecture used for the project.

0.2 Convolutional Neural Networks

The issue of developing statistical models able to classify inputs, also known as *pattern recognition* or *machine learning*, has been around for about 60 years now. Such kind of problems are the ones regularly faced by a wide range of actors in the scientific community, but the first attempts to solve

the pattern classification problem with neural network models date back to the definition of the *perceptron* algorithm by the psychologist Frank Rosenblatt in 1958 [28], whose work was based on the mathematical modelling of neural systems by McCulloch and Pitts. The perceptron, loosely inspired by neurosientific insights about the brain architecture[2], is the first instance of artificial neural network model, but it did not gain much success, mainly due to the inability to work with non linearly-separable datasets.

Some neural network models were developed in the following years, e.g. Fukushima's *Neocognitron* [30] or Weng's *Cresceptron* [31], but neural networks have been spectators for a long time before finding poweful applications in the mid-2000s when Hinton and Salakhutdinov [32] were able to make the training phase feasible in terms of computational time. *Deep learning*, the subfield of machine learning defined by the use of multi-layered neural networks, has since then gained more and more attention thanks to the possibility of solving complex tasks while outperforming most of the pre-existing algorithms for image classification and speech recognition.

The rapid expansion of the field in the last decade is also due to the development of accelerator technology (e.g. GPUs) made available to the scientific community and the number of computational frameworks created to exploit the power of these systems.

For applications with images, particular attention has been devoted to the development of *convolutional neural networks* (CNNs), a particular class of networks that achieve state of the art accuracy in computer vision. Given their ability at classifying images, these have been used in the context of the NFFA project to develop an on-line classification tool within the IDRP platform. We will discuss here the main ideas about CNNs and present the specific implementations used for the project.

A neural network is a model made up of different layers of computational units, the *neurons*; the 0-th layer is also known as the *input* layer, where the input is fed to the network, while the last one is also called *output* layer as it returns the output of the computation. In a typical (feed-forward) *fully connected* architecture (see Fig.2) every neuron of a layer l is *connected* to every neuron in layer $l-1$, that is it receives the collection of outputs of the $l-1$ layer, $\mathbf{z}^{(l-1)} \equiv \{z_i^{(l-1)}\}_{i=0}^{N_{l-1}-1}$ [3] and performs a non-linear computation

[2]Some models of activity in the cerebellum are shaped around the idea of the perceptron, see [29] for instance.

[3]N_l is the number units in layer l.

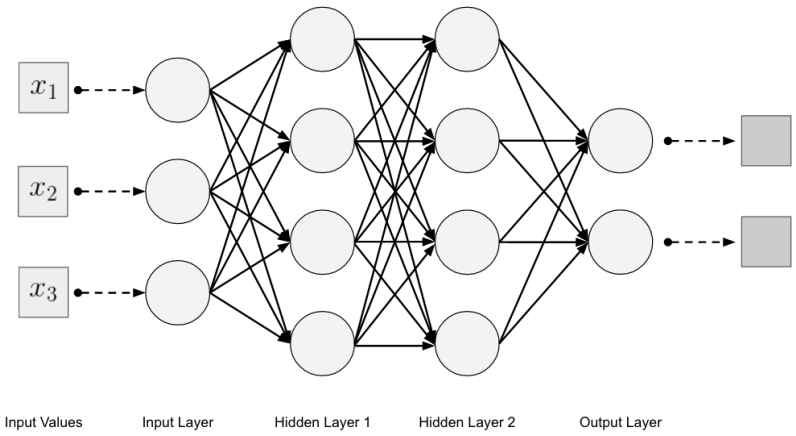


Figure 2: Fully connected architecture

to output to the next layer,

$$\begin{aligned} \mathbf{z}^{(l)} &\equiv \sigma_l \left(\mathbf{W}^{(l-1)} \mathbf{z}^{(l-1)} + \mathbf{b}^{(l-1)} \right) = \\ &\equiv L_{l-1}(\mathbf{z}^{(l-1)}) \end{aligned} \tag{1}$$

where the parameters $\mathbf{W}^{(l-1)}$ and $\mathbf{b}^{(l-1)}$ are called *weights* and *biases* of the layer and σ_l is a proper non-linear function, also called *activation* function.

Overall any network implements a function (f) that maps its input to its output ($f : \mathbf{x} \mapsto y$), which in the simple case just discussed is a composition of the N layers,

$$\mathbf{x} \equiv \mathbf{z}^{(0)} \mapsto y \equiv \mathbf{z}^{(N_L-1)} = L_0 \circ L_1 \circ \ldots \circ L_{N_L-1}(\mathbf{z}^{(0)}), \tag{2}$$

where N_L is the total number of layers.

The ability of the network to capture the correct association f is usually measured by defining a *loss* (or *objective*) function and evaluating it over a *training* dataset, a sub-dataset for which the correct label of each datum is known a priori. The training phase of the model is then the *minimization* of

the loss function[4] $\mathcal{L}$ in the parameter space $\Theta \equiv (\mathbf{W}, \mathbf{b})$,

$$\hat{\Theta} \equiv \arg\min_{\Theta} \mathbb{E}\left[\mathcal{L}\left(\mathbf{x}, y; \Theta\right)\right]_{\mathbf{x}\in\mathcal{D}_{train}} \tag{3}$$

The optimal values $\hat{\Theta}$ are the *learnt* parameters and the network can be tested on new examples to perform *inference* or *generalization*[5].

To summarize, the *training* phase of a neural network model is the tuning of its parameters in order to minimize some kind of error measure, the *loss* function, on the training set. Once the parameters are set one can test the ability of the network on novel inputs and validate it over a *test* set, usually a portion of the whole labelled dataset.

There are numerous algorithms to perform the minimization in Eq.3, the most common of which is a statistical generalization of the *gradient descent* algorithm also known as *stochastic gradient descent* (SGD): a gradient descent procedure consists of updating the set of parameters (given an initial condition) in the following way:

$$\Theta \to \Theta - \eta \nabla_{\Theta} \mathbb{E}\left[\mathcal{L}\left(\mathbf{x}, y; \Theta\right)\right]_{\mathbf{x}\in\mathcal{D}_{train}} \tag{4}$$

where the parameter η is called the *learning rate*. This implies that the estimation of the gradient is performed taking into account the whole training set and this can be computationally expensive.

The stochastic gradient procedure instead makes use of a simple statistical hypothesis: if one splits the dataset in large enough *batches* $\mathcal{B}_i$ then the average loss function can be approximated by the average across the batch:

$$\Theta \to \Theta - \eta \mathbb{E}\left[\nabla_{\Theta} \mathcal{L}\left(\mathbf{x}, y; \Theta\right)\right]_{\mathbf{x}\in\mathcal{B}_i} \qquad \mathcal{D}_{train} \equiv \bigcup_i \mathcal{B}_i \tag{5}$$

While this modification of the learning process also allows to limit the variability of the parameter updates, possibly leading to better convergence,

[4]It must be noted here that simply minimizing the loss function can lead to the issue of *overfitting*, that is the inability of the model to generalize to new unseen inputs. One common technique used to mitigate this effect it the introduction of a *regularization* term (usually a constraint on the solution),

$$\mathcal{L}\left(\mathbf{x}, y; \Theta\right) \longrightarrow \mathcal{L}\left(\mathbf{x}, y; \Theta\right) + \lambda \Omega\left(\Theta\right).$$

[5]The optimization of the loss function generically leads to several different minima, due to the non-convex nature of the problem. For this reason the initial choice of the parameters (as well as the initialization of any other *hyper-parameter*) can significantly impact on the accuracy of the final solution.

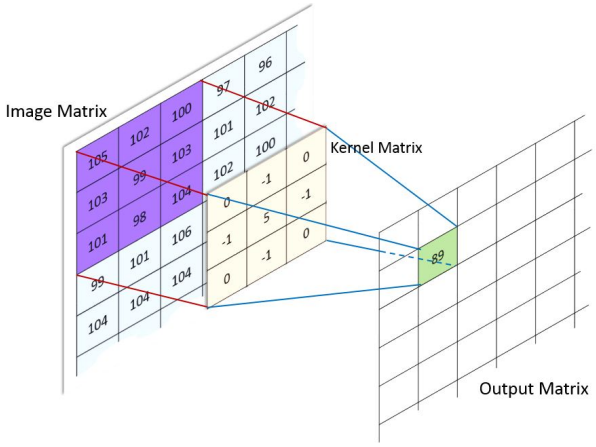
it is a necessary approach when dealing with limited computational resources, especially taking into account that often GPUs memory is quite small.

Typically, in fact, during learning a batch is sent to the device which computes the gradients and updates the parameters; this is repeated a number of times that depends on the number of batches; when the whole dataset has been processed we say that an *epoch* has been completed. Hundreds of thousands of epochs can be necessary for the algorithm to converge, according to the complexity of the problem.

Ordinary fully connected networks are not a feasible solution for any problem or structured input data. For this reason different architectures are realized for different kind of inputs; for the particular case of images, the standard has become convolutional neural networks.

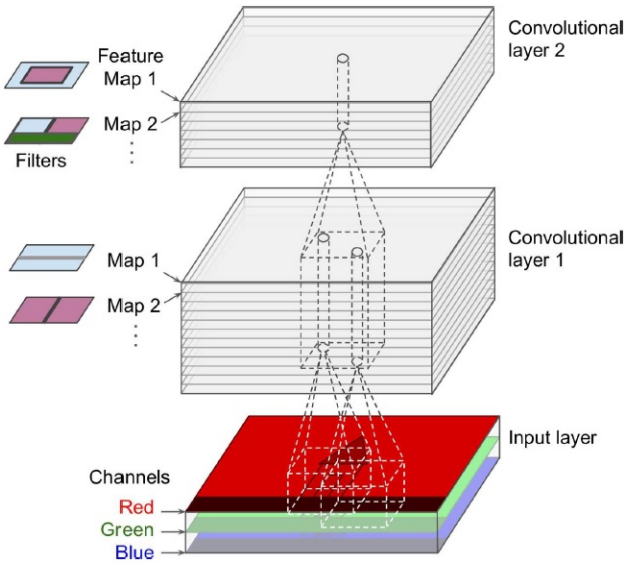
In CNNs data are represented in multidimensional arrays, or *tensors*; in fact images are typically stored as rank-3 tensors of dimensions $[H, W, C]$, where H and W are the height and width of the image and C is the number of color channels (also called the *depth* of the layer). The tensorial structure of the data is matched by the architecture of the convolutional layers, in that neurons are virtually assigned a position in a 3D structure (see Fig.3b).

In order to simplify the exposition, let us consider convolutional layers of depth $C = 1$, that is 2D layers, as in Fig. (3a).



(a) 2D layer

Figure 3: Representation of convolutional layers: (a) is a simplified example of 2D convolution.



(b) 3D layer

Figure 3: Representation of convolutional layers: (b) is a representation of ordinary 3D convolutional layers.

To each unit in the convolutional layer we assign a *kernel* (or *filter*), a matrix K of parameters usually of small dimensions $[k_h, k_w]$, and a specific region in the input area. These regions are uniquely identified by the position (i, j) in the output layer and a parameter called *stride*, say of dimension $[s_h, s_w]$, which is the offset between neighbouring regions. Neuron (i, j) receives inputs form the input neurons delimited in the region

$$(l, m) \quad : \quad \begin{cases} i\, s_h \leqslant l \leqslant i\, s_h + k_h - 1, \\ j\, s_w \leqslant m \leqslant j\, s_w + k_w - 1 \end{cases}$$

The output of this neuron is the *dot product* between the kernel in the inputs,

$$z_{ij} = \sum_{l=i\, s_h}^{i\, s_h + k_h - 1} \sum_{m=j\, s_w}^{j\, s_w + k_w - 1} \sum_{u=0}^{k_h - 1} \sum_{v=0}^{k_w - 1} K_{uv} I_{lm} \tag{6}$$

where I is the total input matrix. It could happen that for boundary neurons the kernel does not fit over the original input, so a common practice is to fill the input with zeros in such a way that the output can be correctly computed. This is called *zero padding*.

The kernel K is a learnable parameter and is shared by every neuron in the layer, thus reducing greatly the number of model parameters, a feature which

is a great advantage over ordinary fully connected networks, for which the number of parameters would easily explode dealing with complex inputs. The final output of the layer is called also a *feature map*; typically to each layer one can decide to assign multiple filters so that the output of the convolutional layer is a 3D tensor of dimension of depth N_k, where N_k is the number of filters of the layer.

General images are not 2D inputs as they are also characterized by the *channel* dimension for colors: in these cases Eq.6 generalizes in the following way,

$$z_{ijk} = b_k + \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} \sum_{k'=0}^{N_k^{(l-1)}-1} K_{uvk'k} I_{i'j'k'} \tag{7}$$

where

$$\begin{cases} i' = i\, s_h + u, \\ j' = j\, s_w + v \end{cases}$$

$N_k^{(l-1)}$ it the number of features map in the previous layer and b_k is the common bias for the k-th feature map.

Convolution layers are usually followed by non-linear layers, e.g. *ReLu* layer, which apply a non linear transformation to the convoluted outputs,

$$z_{ijk}^{(l-1)} \mapsto \sigma_l(z_{ijk}^{(l-1)}).$$

This kind of transformation does not alter the structure of the data. In order to induce invariance in the representation of the data and abstraction of the features from the images, a crucial transformation is performed by *pooling* layers. These layers make use of kernels not to perform convolution, but to *undersample* data. Each neuron in the pooling layer extracts a single statistic from the inputs received from a specific region of the input volume; a typical example is the max pooling operation, where the layer outputs the maximum value of the input in a neighbour. In this way the representation obtained is locally translational invariant, a property that we expect to realize for image classification as the identity of any object is an extremely high level feature that cannot depend on the precise location of the object in the image.

Once the convolution layers have learnt correctly a good number of high level feature it is common practice to use these features as inputs to a fully connected classifier to obtain the final classification.

All of these properties combined make CNNs an extremely powerful tool to classify images and therefore a lot of effort has been devoted by the community in finding more and more advanced architectures to achieve better

performances. Among them one succeseful architecture has been employed in the project to obtain the results in [5]. In the following we briefly discuss this architecture as it is the model used for performing inference in Chapter 1.

0.2.1 Inception

The Inception architecture reached 21.2% top-1 and 5.6% top-5 error for single crop evaluation on the ILSVR 2012 classification, while being six times computationally cheaper and using at least five times less parameters with respect to the best published single-crop inference at the time. The architecture is characterized by the presence of Inception *modules* (Fig.4) , sets of layers that perform different kind of operations in parallel and concatenate their output to form a single module-level output.

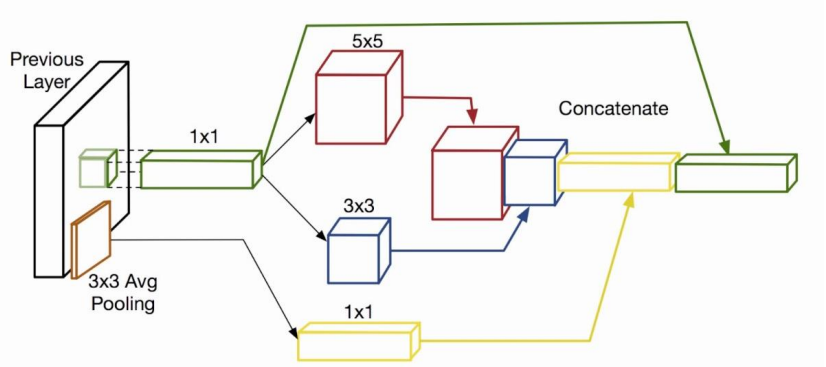


Figure 4: Schematic representation of an Inception module.

In the module, three different convolutional layers perform convolutions with kernel sizes 1×1, 3×3, 5×5 and a pooling layer performs either average or max pooling, with kernel size 3×3; the essential idea behind this branching is to capture correlations of local features of the input volume at different scales and combine this information. In addition to this layers, before any of the three convolutions is performed, an extra 1×1 convolution is performed in order to decrease the depth of the input volume. This *bottleneck* vastly reduces the number of computations and parameters needed to perform every other convolution within the module and thus boosting the computational performance of the network.

In Fig.5 the full architecture is represented for version v3 of the Inception network, that is the architecture employed in [5] and that we use in the project to perform inference.

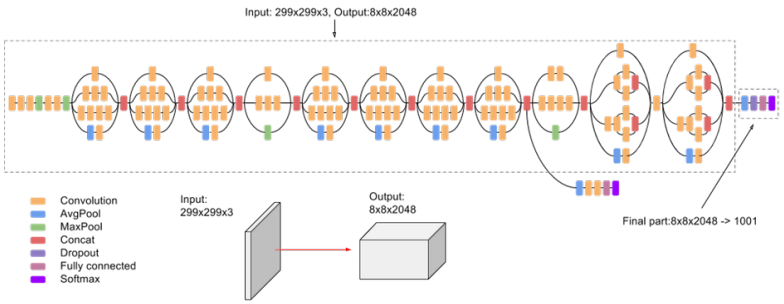


Figure 5: Inception v3 architecture.

Eleven different modules are stacked on top of each other, while the final classification is performed by an ordinary fully connected classifier.

0.3 Thesis overview

In the present thesis we discuss the implementation of distributed systems for the evaluation of convolutional neural network models, exploiting the isolation properties of Docker containers. The results are split in two main chapters: in the first chapter we build and analyse the performance of a cluster managed by Apache Spark for the inference of images starting from pre-trained models; in the second one we discuss a possible implementation of a distributed multi-hosts and multi-gpus system for the training phase of different models, considering its scalability limits.

The focus of the exposition is on the principles behind the implementations, without dealing too much with the details, trying instead to describe every component in a logical order with specific examples encountered in the project. Most of the work has been a non-systematic effort to interpolate between official documentations and public available resources, in order to gain a knowledge as linear as possible. If this linearity does show up along the essay, then one of the main goals of this thesis will be achieved, as the logical dots will be properly connected. In the chapters we will refer directly to the code if needed; every piece of code discussed in the following as well as documentations can be found at the public repository `https://gitlab.com/NFFA-Europe-JRA3/lciuffreda-mhpc-17-18%%`.

The results presented in the following chapters constitute an original body of work which is meant to be a building block for the development of new distributed neural network systems for future projects. As such they open to new lines of investigation for several classification problems.

Chapter 1

Distributed Inference

Inference on a neural network model is a straightforward operation on pre-trained networks as it is a direct feed-forward computation of the classification label for images that were not necessarily used in the training phase,

$$\mathbf{x} \longmapsto y = f\left(x; \hat{\Theta}\right), \tag{1.1}$$

In our project we developed a distributed software infrastructure in order to process massive amounts of inputs; this has been accomplished containerizing the application with Docker. The isolation property of containers is a great advantage in developing applications that can run on multiple platforms but unfortunately so far this isolating ability of Docker containers limited the possibility of exploiting a large number of physical hosts. In fact all the previous results obtained within the NFFA-EUROPE project [5, 7, 8] are based on single-host applications.

In this thesis we present one possible strategy to overcome this limit, thanks to the use of the Docker native Swarm [15] service manager and a virtual overlay network between the containers. Inside each container we installed and configured the distributed computational framework Spark [17], on which we are then able to run our TensorFlow [24] application on multiple CPUs.

This chapter is split in two main sections: in the first one we present the tools used to create the infrastructure, *Docker Swarm* and *Spark*. We then show how to create a cluster able to run an image classificator developed with *TensorFlow* and Spark APIs. In the last section instead we show a possible strategy to tune the cluster using two specific workloads and then proceed to test the performance of the optimal configuration over two different file systems. While discussing about performance we discuss about how metrics are collected in Spark and a tool to extract those measures, *sparkMeasure*.

1.1 Tools

The computational environment in which this work is developed is the C3E Cloud Computing Environment of the Carnia Industrial Park cluster [4] provided by eXact-lab srl [14]. Each node of the cluster has two main Intel Xeon CPUs E5-2697 at 2.70 GHz, for a total of 24 cores and a main memory of 64 GB; additionally two K20s Nvidia GPUs are provided per node, each with 5GB of RAM. For the work presented in this chapter four nodes have been used.

1.1.1 Docker Swarm

Starting from Docker Engine version 1.12, Docker introduced a native cluster management and orchestration tool, Swarm [15]. This tool allows for the creation of so-called *services* across multiple hosts; for our use case we ran services which are simple replicas of the same Docker image.

A Docker Swarm cluster is a set of hosts that can be incrementally joined to form a full structure, allowing individual Docker daemons to communicate and orchestrate activity. There exist two fundamental modalities in which a cluster can be set up:

- a *standalone* cluster is the minimal configuration in which the membership of other nodes is transparent to every node of the cluster;
- a *swarm* mode instead is able to perform more advanced operations such as load balancing and runtime updates.

The Swarm cluster created for our infrastructure is in standalone mode and, as stated before, is made up of four nodes in the C3E cluster.

1.1.2 Creating the cluster

To create a Swarm cluster we just need to select one node as master or *manager* and run

```
docker swarm init --advertise-addr <MANAGER-IP>
```

from it. This will initialize the cluster. In order to configure our Docker cluster we associated the nodes using the Infiniband network IP addresses, to minimize the communication time in the computational tasks. The initialization will return a message containing a command of the form

```
docker swarm join --token <...> <MANAGER-IP>:2377
```

where the value of the token flag is a string generated by the Docker daemon and uniquely associated to the cluster instance we are building. This command can then be run from any other host we want to include in the cluster. The manager node can then promote other nodes to the role of cluster manager, if needed; these manager nodes are in charge of handling the overall status of the cluster, accessible at any time with

```
docker node ls
```

The output of this command returns salient information about every node, for instance:

```
ID                            HOSTNAME    STATUS AVAILABILITY
h0x45d8ywx68isa6m2q3tcp8c   b22         Ready  Active
uek91lb6qa966d3q8536ih7q8   b24.hpc.c3e Ready  Active
7h2gn5hyvm5myqoofsc7pwj7n * b25.hpc.c3e Ready  Active
yar05586zy4pfmp3e2yciawj3   b26.hpc.c3e Ready  Active
```

```
MANAGER STATUS ENGINE VERSION
Leader         18.03.1-ce
               18.03.1-ce
Reachable      18.03.1-ce
               18.03.1-ce
```

Once the cluster is created new services can be launched from any manager node, but in order to do so and have the daemons communicate properly it is necessary to set up an *overlay* network over which internal communications in the cluster can be sent and received.

From any manager node it suffices to run

```
docker network create -d <NETWORK-NAME>
```

This creates an entry in the Docker network list, accessible with

```
docker network ls
```

which returns as output the list of possible network choices

```
NETWORK ID     NAME              DRIVER    SCOPE
02deb513adb3   bridge            bridge    local
d8a05b5adb41   docker_gwbridge   bridge    local
bac987ed7311   host              host      local
fe8f868c4b19   none              null      local
rkaydcw44e84   over-net          overlay   swarm
```

A network, here called *over-net*, can be used by the cluster manager among different machine as indicated by the *swarm* scope. Other *local* networks are present by default and are in charge of the communication among containers running on the same machines.

We are able now to launch replicas of images to different hosts. From any manager node we can start our Docker cluster with a one line command:

```
docker service create <OPTIONS> --network <NETWORK-NAME> \
                      --replicas <N> <IMAGE> <COMMAND>
```

These steps show a possible solution to the problem of distributing a job over containers spawned over multiple machines, but as simple as it looks, this solution has limited functionalities. Other projects have been developed to scale containerized applications; among them it is worthwhile citing Kubernetes [16]: this open-source program, developed at Google, designed for heavy workloads and large number of nodes, is gaining a lot of attention in the world of big data analytics. Kubernetes puts a lot of emphasis on redundancy and resiliency of the jobs; it offers overall a much wider range of features with respect to Docker Swarm but it tends to have more overhead and it is generally more complex to build and run.

We decided to deploy our software with Docker Swarm as it is powerful enough for our purposes and is also made readily available. The nature of the task at hand also does not provide any obstacle for Swarm.

1.1.3 Spark

The Docker cluster architecture discussed in the previous section constitutes the lowest layer of our software infrastructure: on top of it we deployed a *Spark cluster*. Spark is a widely used framework in the Big Data world, being it explicitly developed to handle large workloads and datasets. In the next sections we discuss the main features of the framework and the configuration of our cluster. Finally we show performance results for the system.

Spark has gained great success in the world of data engineering and analytics because it overcomes many of the limits of its predecessor, Hadoop

MapReduce [10]. MapReduce is a distributed framework developed at Google, in which operations on large datasets are performed locally on different workers over batches, while a master node coordinates the job with remote calls to *reduce* and collect data. Other frameworks are designed specifically to handle different ways of processing incoming data, such as stream processing (e.g. Apache Storm [18]), interactive processing (e.g. Apache Tez [19]) and graph processing (e.g. Neo4j [20]). Spark allows instead for unified processing pipelines where these different possible solutions can be combined with the use of the same high-level interface for languages like Java, Scala, Python or R.

At the same time Spark works with multiple data sources and formats and can be combined with different schedulers, such as Yarn [21] and Mesos [22], other then the built-in Standalone scheduler. The end result is that Spark is a very flexible tool to create large Big Data/machine learning pipelines with complex sources.

Among its features the one that probably sets Spark apart from most other frameworks is the ability to work in-memory and exploit data locality, through the realization of the *resilient distributed dataset (RDD)* abstraction: data are collected as distributed partitions across different machines. RDDs can be acted upon with usual *map*s and user-defined functions or cached in RAM. This allows much faster processing with respect to other frameworks like Hadoop, in which data can be reused only if written to disk.

Spark architecture

Spark handles jobs on the cluster by means of starting Java virtual machine (JVMs) processes over the hosts and assigning them user-defined roles. Two main classes of JVMs are spanned: a *driver* process, in charge of distributing data and tasks to the *executor* processes, which are distributed across the hosts (also called *workers*).

The overall architecture (see Fig.1.1) is a *master/slave* one, where the driver process interacts with a *Cluster Manager* and hosts a *Spark Context*, the entry point to any Spark core application, which is in charge of setting up internal services and define the execution environment.

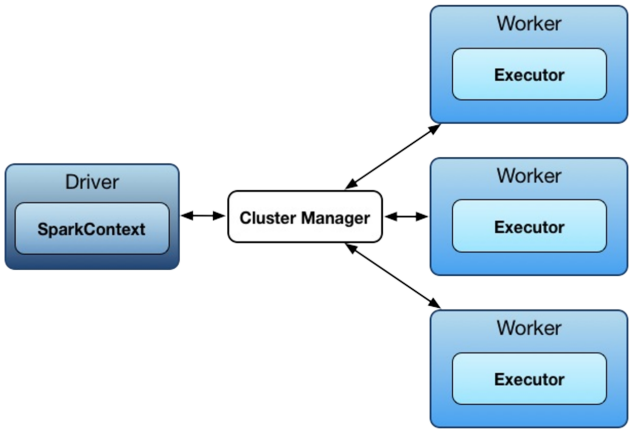


Figure 1.1: Spark architecture

Any job submitted to the cluster with the `spark-submit` command is split in *tasks* by the driver, which takes care of resolving any data dependency in the computational graph associated to the application and grouping independent tasks in *stages*. Tasks and stages are ultimately distributed to executors and executed in parallel. The placement of tasks can also be handled dynamically by the Spark Core in order to maximize the occupation of the hosts; this approach is currently under development and did not provide any advantage for our use case, so at each run we fix the total number of executors and disable any dynamical placing.

Installing and configuring Spark

In order to run Spark over the Docker cluster the simplest solution is to create a Docker image where Spark and its dependencies are prepared, so we created a `spark_tf` image based on the official latest TensorFlow image from the Google repository:

```
1 FROM tensorflow/tensorflow:latest
2
3 ENV HDF5_USE_FILE_LOCKING=FALSE
4
5 EXPOSE    4000-9000
6
7 RUN add-apt-repository ppa:webupd8team/java && \
8     apt-get update -y && \
9     echo oracle-java8-installer shared/accepted-oracle-\
```

```
10      license-v1-1
11      select true | /usr/bin/debconf-set-selections && \
12      apt-get install -y \
13          oracle-java8-installer \
14          libtiff5-dev libjpeg8-dev zlib1g-dev libfreetype6-dev\
15          liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev \
16          python-tk \
17          nano python-pip python-dev links hdf5-tools net-tools\
18      && \
19      pip install --no-cache-dir Pillow h5py && \
20      pip install --upgrade protobuf && \
21      apt-get clean && \
22      rm -rf /var/lib/apt/lists/* && \
23      cd /opt && \
24      SPARK_URL=https://archive.apache.org/dist/spark/\
25      spark-2.3.0/\
26      spark-2.3.0-bin-hadoop2.7.tgz\
27      curl ${SPARK_URL} -o spark.tgz && \
28      tar -xzf spark.tgz && f=\$(ls -d spark*/) && \
29      mv $f spark/ && \
30      rm spark.tgz && \
31      SPARK_HOME=$(pwd)/spark && \
32      echo "SPARK_HOME=$SPARK_HOME" >> $HOME/.bashrc && \
33      echo "export PATH=$SPARK_HOME/bin:$PATH" >> $HOME/.bashrc
34
35  WORKDIR /opt/spark
36
37  CMD ["/bin/bash"]
```

Listing 1: Docker file for the `spark_tf` image.

The image created in this way is able to run Spark 2.3.0 and TensorFlow 1.8.0.

It is crucial at this stage to *expose* a wide range of ports through which the Docker daemons can communicate, since these ports will be used by Spark for communications. Spark defaults any master to slave communication to port 7077, but for the opposite communication the port is randomly assigned. To avoid any issue with ports one could manually define the ports used by Spark and expose only these ports.

Once the image is created it is possible to start a service with any given number of replicas. Most of the work has been done interactively, that is

running configurations manually inside the containers and the way to achieve this is to run from a manager node the following:

```
docker service create --name spark-test --hostname=\
"{{.Node.Hostname}}" --network over-net  \
--replicas 4 spark-tf sleep infinity
```

The `sleep infinity` command just allows the containers to be active indefinitely without any running job. To enter the containers and work from them we just need to *exec* them,

```
docker exec -it <CONTAINER-ID> bash
```

At this point it is possible to configure the Spark cluster.

Spark allows to set default configuration parameters by means of different configuration files, for which different templates can be found in the `SPARK_HOME/conf` folder. For our use case we focused on three particular files:

(i) `spark-env.sh`: used to set environmental variables;

(ii) `spark-defaults.conf`: used to set options to be passed to the Spark context;

(iii) `slaves`: defines the list of slaves' hostnames (only needed on master node).

Every hostname and associated IP must be declared on each node and specified in the `/etc/hosts` configuration file.

With this in mind, a python script (`autoconfigure.py`) has been written in order to autoconfigure the cluster given just a list of hostnames and roles (`cluster.conf`), e.g.

```
b22    master
b24.hpc.c3e    slave
b25.hpc.c3e    slave
b26.hpc.c3e    slave
```

Given this list, the script produces at first the correct `/etc/hosts` on each machine and the three configuration files in the following way:

```
# 1] create spark-env.sh
with open(str(SPARK_HOME)+'/conf/spark-env.sh','w') as f:
     f.write('SPARK_MASTER_HOST='+str(master_list[0].ip+'\n'))
     f.write('SPARK_LOCAL_IP='+str(cluster[LOCALHOST].ip+'\n'))
     if LOCALHOST==master_list[0].hostname:
         f.write('SPARK_EXECUTOR_CORES=0\n')

# 2] create slaves
if LOCALHOST==master_list[0].hostname:
    with open(str(SPARK_HOME)+'/conf/slaves','w') as f:
        for sl in hosts.loc[hosts.role=='slave'].hostname:
            f.write(sl+'\n')

# 3] create spark-defaults.conf
with open(str(SPARK_HOME)+'/conf/spark-defaults.conf','w') \
as f:
    f.write('spark.master\t spark://'+master_list[0].ip+\
    ':7077\n')
    f.write('spark.serializer\t
             org.apache.spark.serializer.KryoSerializer\n')
```

Listing 2: Code from `autoconfigure.py`.

Here just a minimal set of options is defined, since other options can be implemented using flags of the `spark-submit` command. In particular we define once and for all the following environmental variables on each node:

- `SPARK_MASTER_HOST` as the hostname/IP of the Spark master;
- `SPARK_LOCAL_IP` as the hostname/IP of host;
- `SPARK_EXECUTOR_CORES`=0 forbids any task to be executed on the master node, that will then only host the driver process. This is not necessary, but we had enough resources to avoid any interference during the execution.

The minimal configuration of the Spark cluster is thus completed and we can now start the cluster by running the `start-slave.sh`/`start-master.sh` scripts in `SPARK_HOME/bin`. If the daemons do not report any configuration error then the cluster is ready to be used.

1.2 Results

The infrastructure described above is intended to be used for a specific goal in the context of this project: processing a large number of images to obtain an automatic classification for all of them. In particular we looked at how the model developed to produce the published results in [5] can be distributed and if this gives any advantage. In the 2016 MHPC work by Aversa [7], the usage of Spark for such purpose has started being investigated, but only at the scale of a single machine. The results there presented show a weak tendency of Spark to scale under a number of different conditions. In particular Aversa shows that the file system choice is a crucial one and finds that Lustre and the local file system represent the two extreme cases of best and worst scaling, respectively. In the following we discuss again these two conditions on our distributed infrastructure. We use as our starting point the existing scripts created by Aversa in order to fit them in the new structure.

The computational task is approached using TensorFlow (TF), a widely used framework in the field of machine learning and neural networks. The fundamental concept in TF is the *computational graph* that is built before execution: each node in the graph represents a computation and/or action on data, that define the edges of the graph. We will discuss it in some detail in the next chapter.

Our code `measure_inference.py` takes as input a list of `tif` files, the original format of the SEM images, converts them into `jpg`s, feeds the `jpg`s to the network, computes the top predictions for the image and saves both the image and the classification labels in `hdf5` format. The problem at hand is embarrassingly parallel since the outputs for different images do not depend on each other and there is no exchange of information in the process. Thus also the expected modifications using the Spark API should be minimal.

In order to have a functioning Spark code in fact the following steps are sufficient:

(i) Initialize a Spark context

```
sc = SparkContext()
```

(ii) Define the problem in terms of RDDs: given a batched (split across executors) list of images `tif_list_batched`, we just need to send data to every executor,

```
tif_RDD = sc.parallelize(tif_list_batched,
                         numSlices=len(tif_list_batched))
```

(iii) Apply transformation on RDDs:

```
jpg_RDD = tif_RDD.map(tif2jpg_batch)
labelled_images = jpg_RDD.map(run_image_batch).persist()
```

(iv) Save the results and *collect*:

```
hdf5_RDD = labelled_images.flatMap(tif2hdf5_batch).collect()
```

The RDDs are acted upon by the `map` method which parallelizes the action of the user-defined function calls, in order `tif2jpg_batch` (converts each batch of `tiff` to `jpg`) and `run_image_batch` (performs inference on the batch). The output RDD is kept in memory by explicitly calling the `persist` method, which caches data when computed for later reuse. Finally the `collect` method brings the RDD to the driver process. Collecting the result is a necessary action in Spark since the execution is *lazy* and any operation is only performed if the result of a transformation is acted upon and/or saved.

Measuring performance

Once the system is ready and working, it is necessary to understand its performances. Nonetheless measuring and understanding performance in Spark is not an easy task and in fact a multitude of components are running under the hood during execution. A typical job is split in a huge number of small tasks that can be either executed in parallel or are possibly waiting for some kind of information to be shared. The picture is the generally unclear and identifying bottlenecks can be extremely challenging.

Some effort has been directed at this analysis in the literature, for instance in [11] the authors are able to isolate some measurements and show that, for some SQL benchmark and production workloads, surprisingly, I/O is not the main bottleneck but CPU is. This is due to *straggles*, tasks that usually takes significant longer times to complete, the presence of which is usually attributed to scheduler delays, shuffling data (data dependencies to be resolved across tasks) and JVMs' garbage collection.

Spark exposes information about the execution via the web interface accessible by default on port 4040 from the driver's machine; this is a tool that allows to check for the status of the cluster and display some statistics about a job, as in Fig.1.2[1]

General event statistics are collected by the SparkListener class which is designed to intercept events from the DAGScheduler and TaskScheduler

[1]Adapted from `https://www.cloudera.com/documentation/enterprise/5-9-x/topics/operation_spark_applications.html#concept_uhd_zpc_3w__section_klx_xnr_yv`.

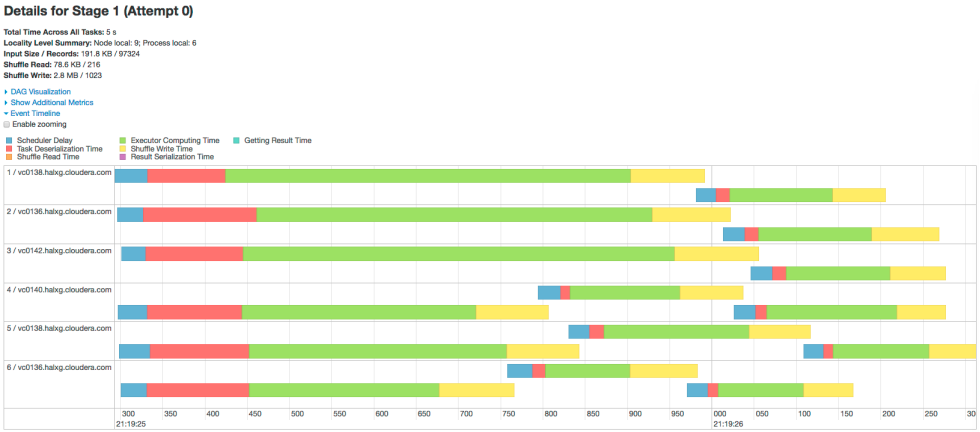


Figure 1.2: Spark web UI statistics

and collecting low-level details of the execution at runtime. Spark also uses SparkListener for keeping track of the executor processes and handling internal communication between its components. Spark only starts its default SparkListener, whose details can be reached via the web UI; nonetheless custom listeners can be defined in order to collect specific values.

Creating a new listener requires a fairly deep understanding of the Spark Core and its components and in general is not a simple task. Fortunately we have been able to import and use a package, *sparkMeasure* [23], which aggregates measurements for each job using the SparkListener interface and collects metrics at task and stage level. The package can be used to output either at standard output or to save results on disk in `JSON` format for off-line analysis. A typical standard output result looks like the following[2]:

```
Scheduling mode = FIFO
Spark Context default degree of parallelism = 8
Aggregated Spark stage metrics:
numStages => 3
sum(numTasks) => 17
elapsedTime => 9103 (9 s)
sum(stageDuration) => 9027 (9 s)
sum(executorRunTime) => 69238 (1.2 min)
sum(executorCpuTime) => 68004 (1.1 min)
sum(executorDeserializeTime) => 1031 (1 s)
```

[2]Taken from `https://github.com/LucaCanali/Miscellaneous/blob/master/Spark_Notes/Spark_Performace_Tool_sparkMeasure.md`

```
sum(executorDeserializeCpuTime) => 151 (0.2 s)
sum(resultSerializationTime) => 5 (5 ms)
sum(jvmGCTime) => 64 (64 ms)
sum(shuffleFetchWaitTime) => 0 (0 ms)
sum(shuffleWriteTime) => 26 (26 ms)
max(resultSize) => 17934 (17.0 KB)
sum(numUpdatedBlockStatuses) => 0
sum(diskBytesSpilled) => 0 (0 Bytes)
sum(memoryBytesSpilled) => 0 (0 Bytes)
max(peakExecutionMemory) => 0
sum(recordsRead) => 2000
sum(bytesRead) => 0 (0 Bytes)
sum(recordsWritten) => 0
sum(bytesWritten) => 0 (0 Bytes)
sum(shuffleTotalBytesRead) => 472 (472 Bytes)
sum(shuffleTotalBlocksFetched) => 8
sum(shuffleLocalBlocksFetched) => 8
sum(shuffleRemoteBlocksFetched) => 0
sum(shuffleBytesWritten) => 472 (472 Bytes)
sum(shuffleRecordsWritten) => 8
```

Listing 3: Example of `sparkmeasure` standard output.

The results can be aggregated either for tasks or stages (i.e. groups of task that are executed concurrently). In our case the two measures collapse since there is only one execution stage due to the extreme data-parallelism of the problem. We thus focus on task measurements.

Tuning the cluster

The output of sparkMeasure is used at first to tune the configuration of the cluster: in order to do so we run the inference application `measure_inference.py` for two different workloads:

- $N_{images} = 15000$;
- $N_{images} = 120000$.

As the inference software is CPU only, we exploit all of the 72 cores available on three worker nodes are used but we vary the number of executor processes (and thus the number of cores assigned to each executor); executor

memory is modified accordingly, setting the value close to the maximum possible value (that is the limit value for which the application returns correctly). The configurations are summarized in the following table:

Executors per node	1	2	3	4	6	8	12	24
Cores per executor	24	12	8	6	4	3	2	1
Executor memory (GB)	50	25	15	12	8	6	2	1

All measurements here are done with a total of four nodes, one master and three workers, and all the input images are stored on a parallel file system (Lustre) accessible by every container.

Among the metrics returned by the Task Metric class we focused on the aggregated results for some of the information collected by sparkMeasure. In particular we look at the following measures:

- *elapsedTime*: total execution time;
- *executorRunTime*: total time spent during execution by the executor processes;
- *executorCpuTime*: total CPU time of the executors;
- *executorDeserializeTime*: total time spent deserializing the portions of RDDs broadcast to the tasks;
- *jvmGCTime*: time the executor JVMs spent in garbage collection while executing the task.

During the analysis we looked also at other measures less directly related to execution, e.g. *diskBytesSpilled* (number of on-disk bytes spilled by the task), *shuffleTotalBytesRead* or *shuffleBytesWritten* (total number of bytes read or written during shuffles, i.e. exchange of data among tasks), but they show no significant impact on performance. This is the case since the problem is embarrassingly parallel and there are no dependencies between tasks.

Fig.1.3 shows the main results for the $N_{images} = 15000$ images workload.

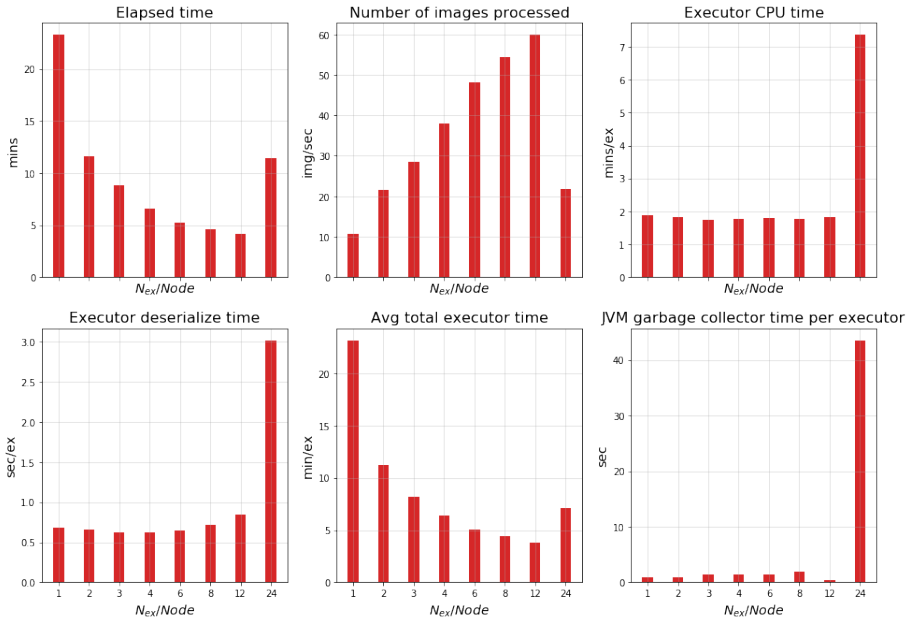


Figure 1.3: Task metrics for $N_{images} = 15000$

There is a clear improvement in the number of images processed as the number of cores per executor is shrank to two, but this behaviour is broken abruptly when the ratio is fixed to one. This is true also for CPU time and deserialization time. The clear drop in performance is strongly correlated to the time spent by the executor JVMs in their garbage collectors: this result highlights the fact that virtualization can be an issue for the performance of Spark applications and thus a careful check of this aspect should be taken into account when tuning the system.

If we change the workload though, this effect seems to disappear: there is no clear correlation now between the JVMs garbage collection time and the performance of the system for the $N = 120000$ workload (see Fig.1.4).

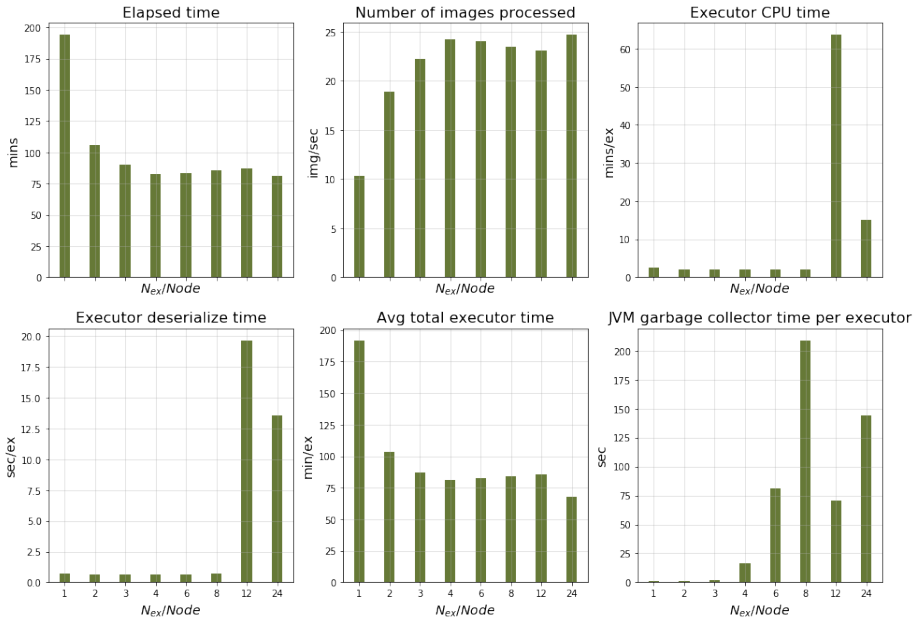


Figure 1.4: Task metrics for $N_{images} = 120000$

In this case there is no clear advantageous choice in the number of CPUs assigned to the executors. The most conservative choice is then the combination of the two results: fixing the number of cores per executor to two is almost optimal for both cases. This choice will allow to have an infrastructure as flexible as possible, as it is highly desirable. With this settings we can test the ability of the system to scale.

Speed-up analysis

Once the optimal configuration has been set up, we want to understand how well the infrastructure is able to scale. Also we want to investigate how this property is affected by the file system on which the software relies. In order to do so we fix the number of cores per executor to two and increase the number of executors per node from one (two CPUs) to twelve (24 CPUs, maximal capacity), and since we place executors independently on each of the three workers the number of CPUs ranges from 3 to 72. The tests are performed for the largest dataset of $N_{images} = 120000$ images.

The outputs of sparkMeasure are collected as before and Fig.1.5 shows the results with Lustre FS, in the same format as previously shown.

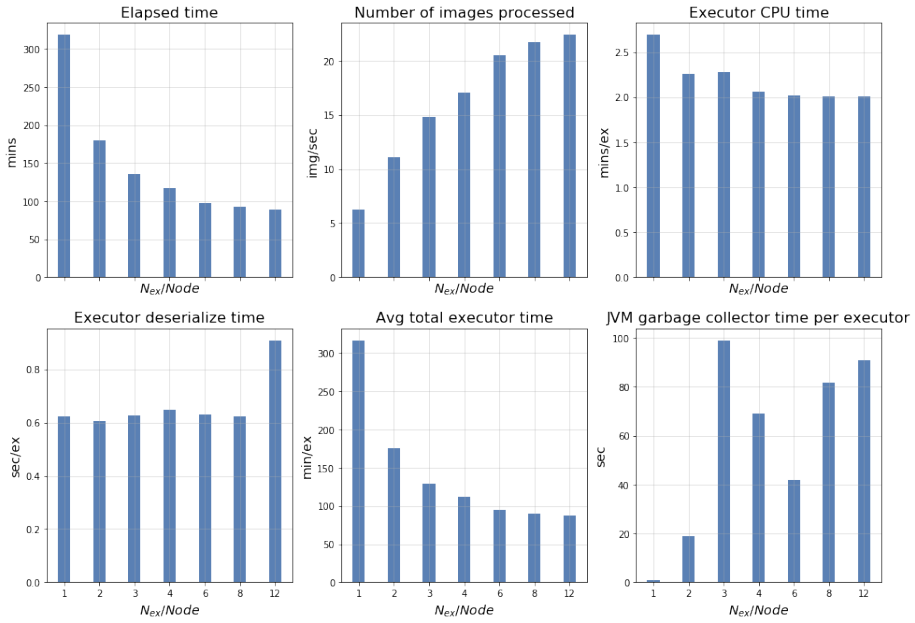


Figure 1.5: Strong scalability - 1 - Lustre

From this result we can extrapolate the speed-up, measured as relative gain in total elapsed time as the number of executors per node is increased, as reported in Fig.1.6

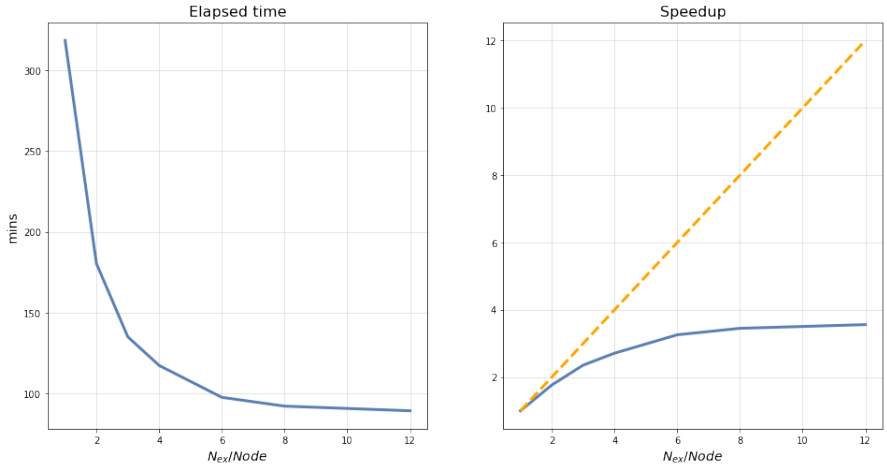


Figure 1.6: Strong scalability - 2 - Lustre

The speed-up gain is always less then 4, in every condition, and far from

the optimal case denoted by the dashed orange line. Since the problem is feed-forward and embarrassingly data-parallel this behaviour is reasonably related to communication between nodes and, more importantly, by the time probably used by I/O since low values of CPU time are recorded. The system reaches it maximum gain the gain at the value of six executors per node, that is at half of the total capacity.

In [12] the authors report a similar effect porting Spark to HPC facilities and running the BigData Benchmark [13] over Lustre file system. Their analysis shows that this kind or workload overwhelmes the file system by calling a large number of metadata operations, which are usually characterized by high latency. To overcome this issue they work both on the software and hardware side, suggesting that working with local disk improves significantly the performance. For this reason and for consistency with the work in [7], we replicate the same analysis placing the images on the local disks of the machines. Figg.1.7 and 1.8 show the outcome of this measurement.

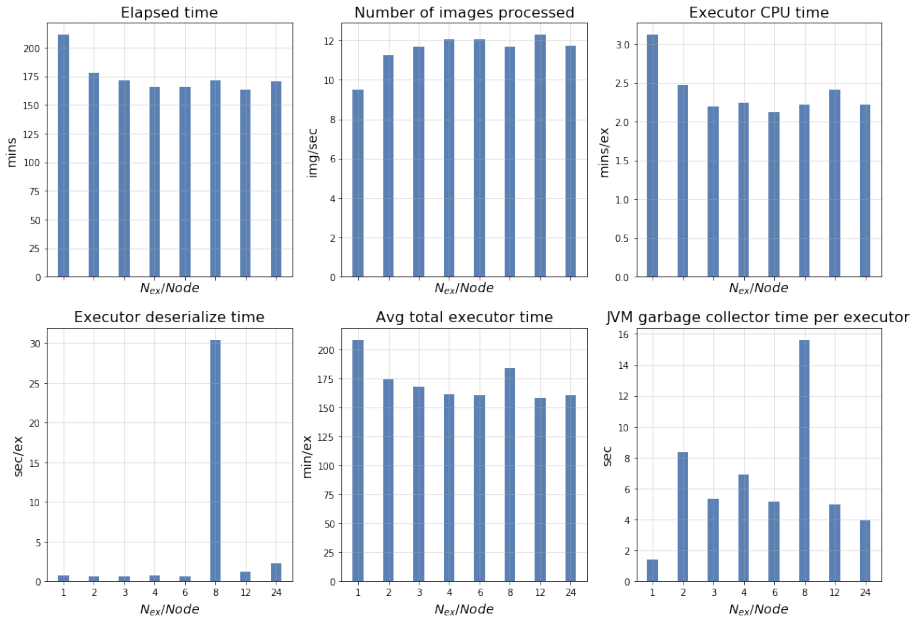


Figure 1.7: Strong scalability - 1 - Local FS

Unlike the case of Lustre, where the system shows a very weak ability to scale, in this case the speed-up almost completely disappears. In terms of absolute execution time the case of the system at half capacity is the best one, but the gain in performance from adding computational power is one order of magnitude less then the ideal case ($\sim$ 1.2 against 12). This is in

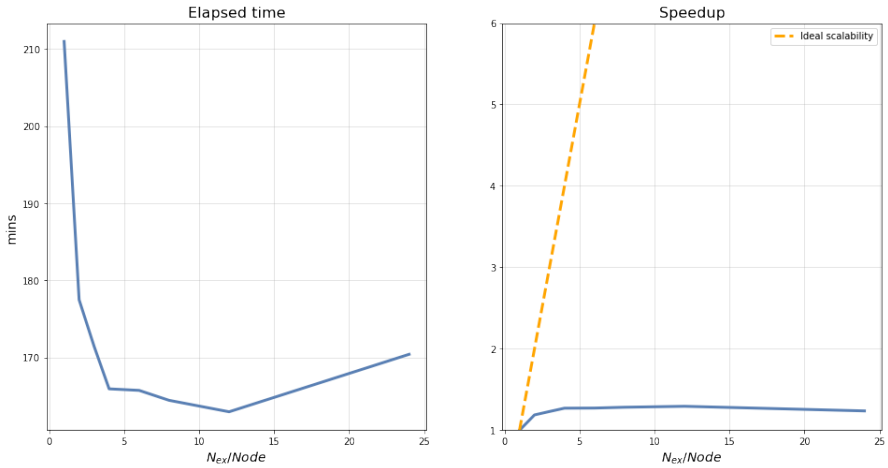


Figure 1.8: Strong scalability - 2 - Local FS

agreement with the single machine measurements in [7].

We can conclude that the choice of the file system is crucial for the performances of Spark; on the other hand even the best case scenario is not performing as we hoped. It is possible that different tuning configuration might lead to better performance but one should keep in mind that Spark is not designed to run on classical HPC facilities but rather on data-centric facilities. As discussed in [12] optimizations of Spark in a HPC context, like the one employed for this project, requires hardware adaptation and heavy modifications of the Spark core software.

The development of Spark code is nonetheless interesting as more and more attention is devoted to developing distributed software systems for machine learning services that might on a common standard framework to make the code portable on multiple architectures.

Chapter 2

Distributed Training

In this chapter we illustrate a strategy to distribute a neural network training process over multiple machines using the TensorFlow native distributed framework. In particular we discuss a simple application for a two-layer classifier trained over the MNIST dataset and illustrate how to extend the code for other models.

The first section is devoted to the introduction of the two main concepts of a TensorFlow code: the computational graph and the session. These translate in the distributed framework so we introduce them here, together with the concept of device placing, discussing two very basic examples. The second section instead translates the concepts of the first section in the language of Distributed TensorFlow, discussing the implementation of the simple model of neural network used here as a test case. Finally we discuss the results of this test and illustrate a possible strategy to generalize the code.

2.1 TensorFlow

TensorFlow (TF) [24] is an open-source computational software developed by the Google Brain team for machine learning applications; APIs are released for many languages such as python, C, C++, Java and Go. The computational paradigm of TensorFlow forces the user to decompose explicitly his or her own problem into a *computational graph*, a directed graph where nodes represent computations or variables and links represent data and their transforms. Once the problem has been cast in the form of a computational graph, TensorFlow needs a *session* to execute it; consider for instance the following *Hello world!* example:

```
import tensorflow as tf

# Graph creation
hello = tf.constant('Hello world!')

# Define and run tf session
with tf.Session() as sess:
        # Run the op
        print(sess.run(hello))
```

Listing 4: `Hello world` example in TF.

The first part of the code is devoted to the creation of the computational graph: in this case there is only one node, the instruction

```
hello = tf.constant('Hello world!')
```

The operation (*op*) `tf.constant` defines a TF variable that stores the string `'Hello world!'`. In order to execute the action TF needs a `tf.Session` to be started: the session defines the environment in which the execution takes place and takes care of setting operations and variables on the available devices. The method `sess.run` then calls for the execution of the node `hello` in the graph.

Starting from here complex computational graph can be built, but this flow is essentially the same for any TF application. A useful property crucial to distribute any calculation is the possibility of manually placing any operation on different devices: if there are no dependencies between nodes placed on different devices, TF is able to run them in parallel. As an example, consider the following piece of code[1]

```
c = []
for d in ['/device:gpu:0', '/device:gpu:1']:
        with tf.device(d):
            a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0],\
                            shape=[2, 3])
            b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0],\
                            shape=[3, 2])
            c.append(tf.matmul(a, b))

with tf.device('/cpu:0'):
```

[1]From the official guide: `https://www.tensorflow.org/guide/using_gpu`

```
11              sum = tf.add_n(c)
12
13  with tf.Session() as sess:
14          print(sess.run(sum))
```

Listing 5: Placement of variables on different devices.

Here copies of the variable `a` and `b` are placed on both `'/device:gpu:0'` and `'/device:gpu:1'`; the first op node is the `tf.matmul`, here placed on both GPUs that execute the computation independently. Each appends its result in a list `c`. The final node of the graph is the `sum` node, which will be executed on the CPU `'/device:cpu:0'` and should return the sum of the singular computations of each GPU.

A typical approach to the parallelization of a training process it to create copies of the model (defined in the graph) across multiple devices which will operate on different subsets of the data, in a fully data parallel fashion; the update of the model parameters is usually placed on the CPU and can either be *synchronous* or *asynchronous*. The latter case may be faster but usually results in a worst accuracy performance of the network.

In more detail, a training step can be realized in the following way (Fig.2.1): a central unit (typically the CPU) sends batches of images to the devices (the two GPUS in Fig.2.1) which compute the loss function and the gradients; the gradients are then sent back to the central unit to update the parameters. The procedure goes on like this as long as a stopping condition is not met (e.g. maximum number of epochs).

The detailed implementation of these steps strongly depends on the version of TF used and the format in which the input is formatted, so we omit details here that will be discussed with our code later on.

2.2 Distributed TensorFlow

Ordinary TF code is expected to be executed locally on a single machine, with possibly many devices, but as models and data get more and more complex in everyday applications, it is necessary to think about solutions to scale the code to a large number of hosts. In order to do so, Google developed its own system of remote procedure call (RPC), called gRPC, to connect services on different machines.

The component of TF that relies on this kind of communication is called *Distributed* TF and is the main tool used for this part of the project.

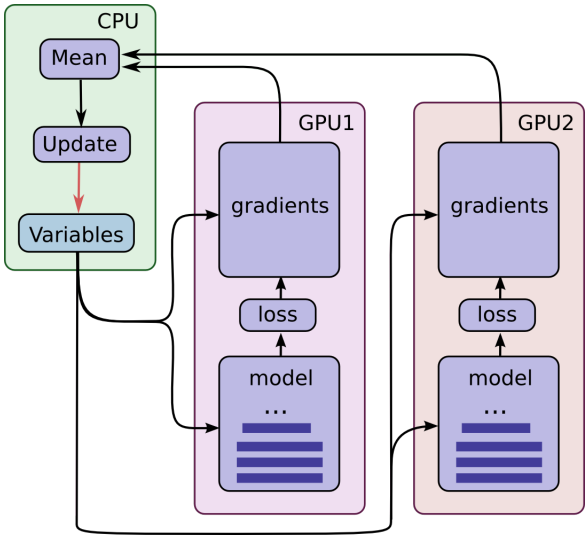


Figure 2.1: Training scheme with two GPUs.

In order to test the system we train a two layer fully connected classifier over the MNIST database, a set of 70000 (28×28 pixels) images of hand-written digits, historically used as a benchmark dataset. The example developed is simple enough to capture the main features of the distributed framework. The training is asynchronous and we employ a *between-graph* replication approach, meaning that every node on the cluster starts its own process (or *server*) creating a copy of the computational components of the graph locally, while pinning variables on a *parameter server*. Distributed TF allows for two classes (also called *jobs*) of nodes in the cluster: a node can either be a *worker node*, i.e. a node that takes care of executing (some) operations of the graph, or a *parameter server* (ps), i.e. a node that hosts shared variables, in our case the parameters of the model to be updated.

The first step in the construction of the code is to instruct every host of the overall structure, the *cluster*, instantiating a `tf.train.ClusterSpec` structure, defined by the two lists `ps_hosts` and `worker_hosts`, and defining a `tf.train.Server` which will specify the process to be executed inside the cluster,

```
cluster=tf.train.ClusterSpec({"ps": ps_hosts,\
                              "worker": worker_hosts})
server=tf.train.Server(cluster,\
                       job_name=FLAGS.job_name,\
                       task_index=FLAGS.task_index)
```

Listing 6: Definition of the cluster.

The `ps_hosts` and `worker_hosts` lists are nothing but two lists containing the IPs of the ps/worker nodes, and the ports for communication. These are passed as argument to the main code with flags, together with the specification, for each node, of its job (`ps` or `worker`) and its *task* index, an integer index starting from 0 that must be assigned to the servers within the cluster.

Once the cluster is defined, the main application can be started,

```
if FLAGS.job_name == "ps":
        server.join()
elif FLAGS.job_name == "worker":
    is_chief = (FLAGS.task_index == 0)
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

Listing 7: Starting the servers.

The ps servers call the `join` method to initialize the communication channels towards the workers and start listening to the worker servers on the assigned ports.

Among the workers, the one with task index 0 is given the *chief* status: this process will take care of starting and managing the session, coordinating the workers, and logging the results to disk.

The main section of the code can be split in two parts, just as any TF code: graph definition and session execution. Distributed TF introduces some changes in the objects definitions, but conceptually everything is as in the examples discussed before.

The first step in logical order of execution is naturally the definition of the inputs of the network; so far TF APIs suggested the usage of *queues* to feed data to the devices, but the recently developed `tf.data` API introduces two main abstractions: a `tf.data.Dataset` class which is the collection of data as `tf.Tensors`; acting on the dataset is possible with the use of the `tf.data.Iterator` class. An interesting side effect is that the interaction of the `tf.data.Dataset` with the session simplifies considerably the input section of the code.

The `tf.data` API is also endowed with the method `tf.data.TFRecordDataset` which allows reading the images from disk (in the binary TF record format) and save them in a `tf.data.Dataset` instance, which can be acted upon with user-defined transformation.

In our case the input batch is extracted by the `inputs` function, defined as:

```
def inputs(train=True, batch_size, num_epochs):

    filename = [os.path.join(INPUT,'train.tfrecords')]
    if not train:
        filename=[os.path.join(INPUT,'validation.tfrecords')]

    with tf.name_scope('input'):
        # Deserialize and transform the dataset
        dataset = dataset.map(decode)
        dataset = dataset.map(normalize)
        dataset = dataset.shuffle(1000 + 3 * batch_size)
        dataset = dataset.repeat(num_epochs)
        if not train:
            dataset = dataset.repeat()
        dataset = dataset.batch(batch_size)

        iterator = dataset.make_one_shot_iterator()

    return iterator.get_next()
```

Listing 8: Input function from `distributed_mnist.py`

The output is a pair of tensors `[batch_images,batch_labels]` which can be directly fed to the model; more importantly the iterator is *moved* along by the session, so that different batches are extracted at different training steps.

The definition of the graph is replicated with the help of the function `tf.train.replica_device_setter` that allows to place, automatically, any variable on the parameter servers and any operation on the local worker device,

```
 with tf.device(tf.train.replica_device_setter(worker_device=\
                "/job:%s/task:%d/gpu:0" % (FLAGS.job_name,\
                FLAGS.task_index),cluster=cluster)):

        # Get input
        next_example, next_label = inputs(batch_size=\
                                   batch_size,\
                                   num_epochs=num_epochs)
```

```
9
10          # Define model
11          with tf.name_scope('hidden1'):
12              weights = tf.Variable(tf.truncated_normal([784,\
13                        128],stddev=1.0/math.sqrt(float(784))),\
14                        name='weights')
15              biases = tf.Variable(tf.zeros([128]),name='biases')
16          with tf.name_scope('hidden2'):
17              weights = tf.Variable(tf.truncated_normal([128,\
18                        64],stddev=1.0 \
19                        math.sqrt(float(128))),name='weights')
20              biases = tf.Variable(tf.zeros([64]),name='biases')
21          with tf.name_scope('softmax_linear'):
22              weights = tf.Variable(tf.truncated_normal([64,\
23                        10],stddev=1.0 /\
24                        math.sqrt(float(64))),name='weights')
25              biases = tf.Variable(tf.zeros([10]),name='biases')
26              logits = tf.matmul(hidden2, weights) + biases
27
28          # Define optimization
29          global_step = tf.train.create_global_step()
30          optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.07)
31          loss = tf.losses.sparse_softmax_cross_entropy(labels=next_label,\
32          logits=logits)
33          train_op = optimizer.minimize(loss=loss,global_step=global_step)
```

Listing 9: Graph definition from `distributed_mnist.py`

As illustrated above, after the graph definition, in a regular TF code it would suffice to start a `tf.Session` to run and execute any operation; for the distributed case, the `tf.data` API suggests the use of a specific class, `tf.train.MonitoredTrainingSession`. This class offers a number of advantages over the standard (low level) `tf.Session`, even if used locally: for instance, the automatic initialization of the variables, the possibility to define *hooks* to automatically save the model and create statistics during learning. More importantly it handles the communication between workers.

In the test we fix the total number of epochs for the training phase by defining a *hook* for the session:

```
1  hooks=[tf.train.StopAtStepHook(last_step=num_epochs)]
2
```

```
with tf.train.MonitoredTrainingSession(master=server.target,\
     checkpoint_dir=OUTDIR,is_chief=is_chief,\
     hooks=hooks) as sess:

     while not sess.should_stop():
             _, global_step_value, loss_value =\
             sess.run([train_op, global_step, loss])
```

Listing 10: Session execution from `distributed_mnist.py`

The session `sess` runs in this example the three op nodes `[train_op, global_step, loss]` and collects their outputs on the left hand side; during execution it will also save checkpoints of the model parameters and summary statistics in the folder `checkpoint_dir`.

We briefly sum up the important concepts of this section:

(i) The main blocks of a TF program are present also in the distributed framework: *graph* creation and *session* execution;

(ii) Three key concepts are specific of Distributed TF:

- definition of a cluster, `tf.train.ClusterSpec`;
- distribution of variables and ops given by `tf.train.replica_device_setter`;
- definition of a `tf.train.MonitoredTrainingSession` session.

2.3 Results

In this section we report the results of the performance analysis of the MNIST classifier described above.

The metrics for the analysis are collected by TensorFlow in files that can be read by the TensorBoard application [25]; here summary statistics and the computational graph (Fig.2.2) are visualized.

The model has been run over three different physical hosts, for a total of six GPUs; each server is started in the following way:

```
python distributed_mnist.py --ps_hosts <PS_IP>:2222 \
--worker_hosts <WORKER1_IP>:2222,...,<PS_IP>:2223 \
--job_name <WORKER/PS> --task_index ...
```

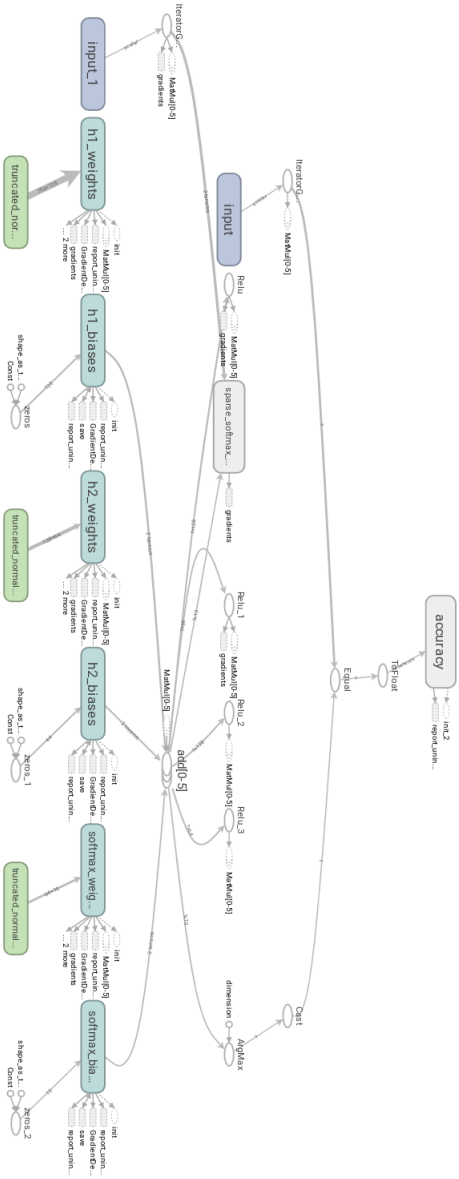


Figure 2.2: Graph of the model as shown by `TensorBoard`.

In order to start a worker on the ps server, and then using two additional GPUs as workers it is necessary to *hide* the devices to the ps server. This is achieved by setting to null the `CUDA_VISIBLE_DEVICES` environmental variable for the ps process,

```
CUDA_VISIBLE_DEVICES= python distributed_mnist.py \
--ps_hosts <PS_IP>:2222 --worker_hosts <WORKER1_IP>:2222,... \
...,<PS_IP>:2223 --job_name ps --task_index 0
```

With these settings the system has been tested to train the network, spanning the number of GPUs from one to six. Fig.2.3, 2.4, 2.5 show the main statistics computed. The accuracy is computed at each training step over a validation set of $N = 1000$ images by the ps CPU, and represents the fraction of corrected classifications.

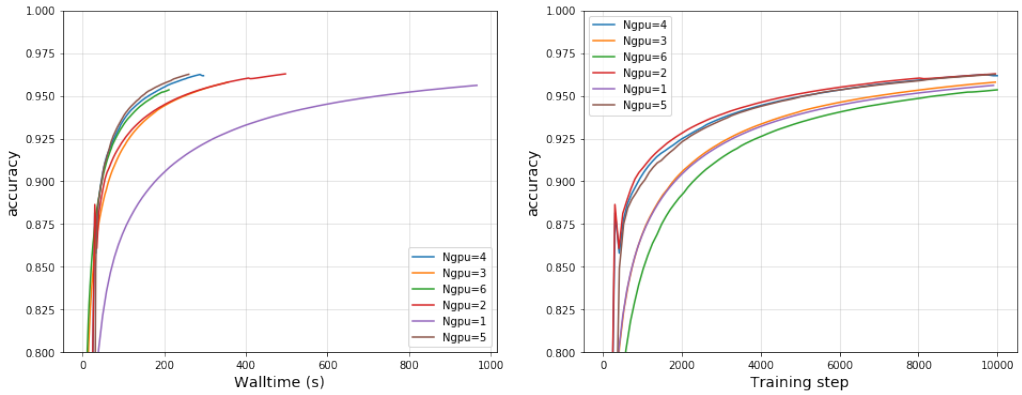


Figure 2.3: Left side: accuracy vs. relative wall time. Right side: accuracy vs. global training step.

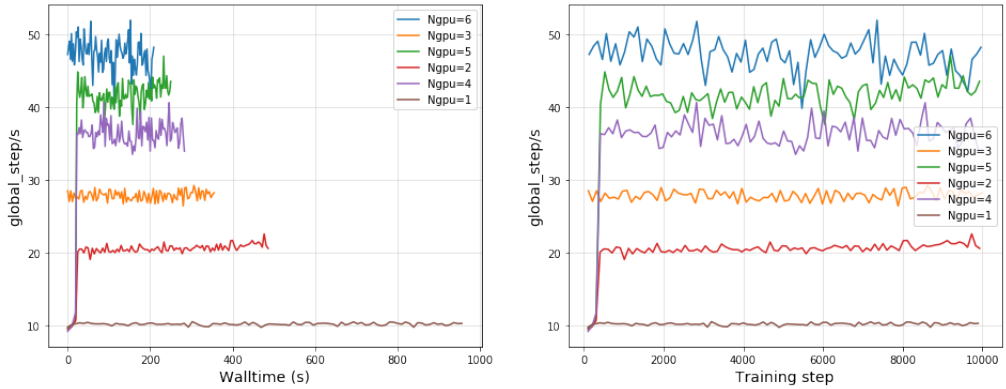


Figure 2.4: Left side: global steps per second vs. relative wall time. Right side: global steps per second vs. global training step.

It is easy to note that as the number of GPUs increases the time for convergence is reduced. In Fig.2.4 the number of global training step per second is plotted: for any number of devices there is a delay period at the beginning of the execution where the master node starts working before all the workers join. This latency effect is small compared to the total execution time and so no significant delay is present at the end of the job between the workers. Unfortunately we verified cases in which the communication to and/or from some specific machines is not optimal, and this finally results in a consistent delay that breaks down the speed-up effect of adding machines to the pool. The issue has been reported to the system administrator of the cluster who has been able to identify the issue as related to the specific version of the operating system. We thus filtered the nodes in the cluster by imposing compatibility requirements in order to overcome the delay issue. In this way the results will not be affected by node-specific properties.

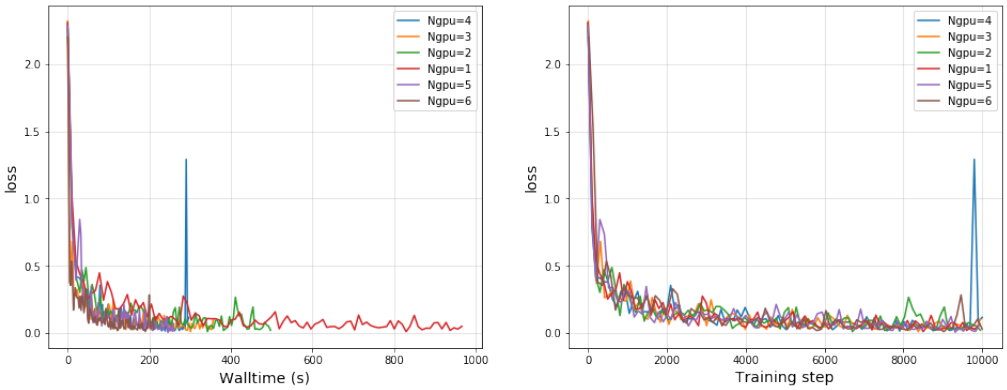


Figure 2.5: Left side: loss function vs. relative wall time. Right side: loss function vs. global training step.

To quantify the performance of the system we looked at the execution time up to $N = 10000$ epochs; the results are shown in Fig.2.6.

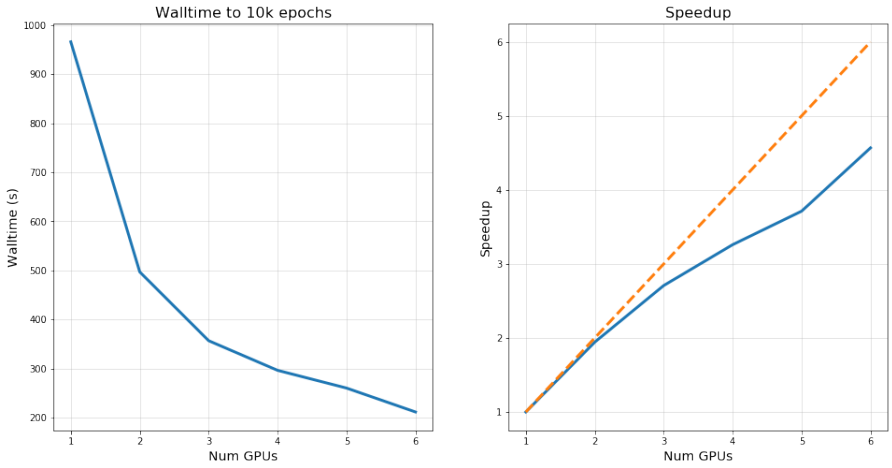


Figure 2.6: Speedup results.

The system shows a good ability of scaling up with multiple machines. This might be due to the low complexity of either the model and the data, which are then not overloading memory and devices. It must be noted here that it is possible that adding computational complexity could result in even better results as long as the memory is not overloaded during execution.

These results open up for the possibility of generalizing to more complex models; it is in fact planned to port the architecture studied and analysed in [8], at the time of writing the most accurate model on the NFFA dataset, to the distributed environment.

Conclusions

In this thesis we have been able accomplish the goal of defining a distributed computational environment for the neural network models within the NFFA-Europe project. In particular we realized:

(i) a fully distributed infrastructure for inference, based upon two main layers:

- a container level, managed with Docker Swarm. This was achieved by initializing and adding nodes to the Docker swarm cluster, and defining an overlay network in order to start a correct communication among containers and overcoming the isolation issue;
- a computational environment with Spark. The initialization of the Spark cluster, a basic configuration and the start u of the correct daemons has been automatized by means of a python script, `autoconfigure.py`.

On top of these layers a TensorFlow application, `measure_inference.py`, has been interfaced with the cluster with the python-Spark APIs and the sparkMeasure package to collect metrics about the system duryng execution. After tuning the Spark cluster parameters, a speed-up test has been performed over both Lustre file system and local file system: the outcome of these tests is compatible with results in [7] and clearly show that the infrastructure does not scale as expected.

(ii) a strategy for distributing the training phase of a neural network model over a multi-gpu and multi-hosts system with the use of Distributed TensorFlow. We measured the time to $N = 10000$ epochs for a simple classifier model trained on the MNIST dataset, with the number of GPUs ranging from one to six: the speed-up gain is close to the ideal case (~ 4.6 against 6 for the case of six GPUs), suggesting that the system is performing well under these conditions.

Altogether these result will allow:

(i) to port a Spark cluster on different systems and machines thanks to the isolation property granted Docker containers to perform inference on new datasets. Along this path a number of direction can be explored: studying the interaction of Spark with different container orchestrators, e.g. Kubernetes, or running a cluster with GPUs. Docker Swarm is not natively able to work with GPUs, but efforts could be spent on the implementation of a cluster based on the wrapper *NVIDIA Docker*, [26]. Starting from version 2.0, NVIDIA Docker allows the use of GPUs from Swarm services in a natural way [2].

(ii) to speed-up the training of complex models. Building the correct interface, with minor modification to the code, any model can be trained in the same way as the simple MNIST classifier discussed in the previous chapter. This means that much lower number of hours of CPU/GPU time can be spent on training the model and more computational power can be devoted to the analysis of different models. In particular we are now looking at the possibility to have the model *DenseNet* in [8] scaling over multiple hosts. An interface for any new model should face two main challenges:

   - Modification of the input pipeline: the main issue here is to decode the tfRecords files in the exact format as they were originally created. The `decode` function must then be adapted to the specific dataset, but the use of the `tf.data` API guarantees that the overall dataset definition is not going to change. This implementation is thus really flexible and should be easily modifiable for future applications;
   - Redefinition of the computational graph: as explained above, the TF graph is basically the definition of the neural network. This means that, obviously, in order to change the model, a redefinition of the graph is necessary. A possible generalization should be *modular*: it should be possible to import the model with a module containing the network definition and a function to instantiate the graph. In the definition of the model one could also add different metrics to the summary statistics created by the session.

   Another possible advance could be to exploit an NVIDIA Docker cluster to have a *resource agnostic* system for large networks to train on other machines, like the ones at Cineca, endowed with more advanced

[2] See `http://cowlet.org/2018/05/21/accessing-gpus-from-a-docker-swarm-service.html`

hardware. On the software side, other frameworks can be explored: for instance the Microsoft open-source project CNTK [27] uses a communication paradigm based on OpenMPI for its distributed version, and as show it shows better and more promising scalability properties.

Overall this work sets the first step towards the realization of a fully distributed environment for the classification problem within the NFFA-Europe project but it is expected to be generalizable enough to realize alternative strategies for other projects.

Bibliography

[1] *NFFA-EUROPE project*, `https://www.nffa.eu/`

[2] *CNR-IOM*, `https://www.iom.cnr.it/`

[3] *TASC Institute*, `http://tasc.iom.cnr.it/`

[4] *Carnia Industrial Park*, `www.carniaindustrialpark.it`

[5] Modarres M.H., Aversa R., Cozzini S., Ciancio R., Leto A. & Brandino P.B., *Neural Network for Nanoscience Scanning Electron Microscope Image Recognition*, Scientific Reports, volume 7, Article number: 13282 (2017)

[6] Aversa R., Modarres M.H., Cozzini S., Ciancio R. & Chiusole A., *The first annotated set of scanning electron microscopy images for nanoscience*, Scientific Data volume 5, Article number: 180172 (2018)

[7] Aversa R., *Scientific image processing within the NFFA-EUROPE Data Repository*, MHPC thesis 2015/2016

[8] De Nobili C., *Deep Learning for Nanoscience Scanning Electron Microscope Image Recognition*, MHPC thesis 2016/2017

[9] Coronica P., *Feature Learning and Clustering Analysis for Images Classification*, MHPC thesis 2017/2018

[10] Dean J., Ghemawat S., *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150

[11] Ousterhout K., Rasti R., Ratnasamy S., Shenker S., Chun B., *Making Sense of Performance in Data Analytics Frameworks*, USENIX NSDI, 2015

[12] Chaimov N., Malony A., Canon S., Iancu C., Ibrahim K., Srinivasan J., *Scaling Spark on HPC Systems*, High Performance and Distributed Computing (HPDC), February 5, 2016,

[13] *BigData benchmark*, `https://amplab.cs.berkeley.edu/benchmark/`

[14] *eXact lab*, `http://www.exact-lab.it/`

[15] *Docker Swarm*, `https://github.com/docker/swarm/`

[16] *Kubernetes*, `https://github.com/kubernetes/kubernetes`

[17] *Apache Spark*, `https://spark.apache.org/`

[18] *Apache Storm*, `https://storm.apache.org/`

[19] *Apache Tez*, `https://tez.apache.org/`

[20] *Neo4j*, `https://neo4j.com/`

[21] *Yarn*, `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

[22] *Mesos*, `http://mesos.apache.org/`

[23] *sparkMeasure*, `https://github.com/LucaCanali/sparkMeasure`

[24] *TensorFlow*, `https://github.com/tensorflow/tensorflow`

[25] *TensorBoard*, `https://github.com/tensorflow/tensorboard`

[26] *Nvidia-Docker*, `https://github.com/NVIDIA/nvidia-docker`

[27] *Miscrosoft CNTK*, `https://github.com/Microsoft/CNTK`

[28] Rosenblatt F., *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, pp. 386–408, (1958)

[29] Marr, D., *A theory of cerebellar cortex*, The Journal of Physiology, 202(2), 437–470, (1969)

[30] Fukushima, K., *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*, Biological Cybernetics, 36[4], pp. 193-202 (April 1980)

[31] Weng J., Ahuja N. & Huang T., *Object recognition by self-organizing neural network which grows adaptively*, Parallel Image Analysis. ICPIA 1992. Lecture Notes in Computer Science, vol 654. Springer, Berlin, Heidelberg. (1992)

[32] Hinton G.E., Salakhutdinov R.R., *Reducing the Dimensionality of Data with Neural Networks*, Science: Vol. 313, Issue 5786, pp. 504-507 (2006)