Master in High Performance Computing

Scientific image processing within the NFFA-EUROPE Data Repository

Supervisor:
Stefano Cozzini

Candidate:
Rossella Aversa

2nd edition
2015–2016

Acknowledgements

First of all, I definitely want to thank my supervisor Dr. Stefano Cozzini, one of the most enlightened persons I've ever met.
Moreover, a special thanks goes to Stefanino, always ready to help me, to make me laugh, and to play the right music for the most dramatic moments.
Thanks also to Pino, Francesco, Moreno, Juan, Marlon, and Owais for having shared with me part of this trip. In particular, I want to thank Hadi for having collaborated with me to this work.
Thanks to the people close to me: Enzino, Marco, Gabri, Ambra, and most of all Mirko. I'm so luky to have you in my life.
Finally, thanks to my family, always supporting me when I don't feel able to handle things alone. I love you all.
This work has been done within the NFFA-EUROPE project. I thank the project and the CNR-IOM for supporting my participation to the HPC master.

Abstract[1]

This thesis is embedded within the NFFA-EUROPE project, that aims to setup the first overarching Information and Data management Repository Platform (IDRP) for the nanoscience community.

The main goal of the IDRP developement is to semiautomatize the harvesting of scientific data coming from many different instruments among NFFA European facilities, and identify the correct metadata to allow them to be searchable accordingly to the FAIR Guiding Principles [1].

For our analysis, we selected data from a specific instrument, the Scanning Electron Microscope (SEM). The reasons for this choice are many:

- the instrument is available at the CNR-IOM in Trieste, so we could discuss with scientists about their needs and have fast feedback;
- a significant amount of SEM images (roughly 150,000) have been provided us for testing purposes;
- the plugin for the SEM has already been tested and is available for this instrument;
- ten out of the twenty NFFA European partners have a SEM facility, so our work can be exported to a sizeable part of the community.

The specific goals of this thesis are the following:

- explore machine learning algorithms to classify scientific images coming from the SEM instrument;
- once the images have been categorized, enrich the plugin with the ingestion of new metadata. In this way, a search engine can be used to find the relevant images through a semantic search on the database;
- setup a computational and storage infrastructure to process a massive amount of such images efficiently;

The achievements of our work are the following:

- a supervised machine learning algorithm has been successfully implemented and tested;

[1]This chapter might have been called "Introduction", but nobody reads the introduction, and we want you to read this. We feel safe admitting this here, in the footnote, because nodoby reads the footnotes, either.

- algorithms and tools for massive data processing have been identified, set up, optimized, and finally benchmarked;
- many tools for simplifying the setting up of complex computational and storage infrastructures have been extensively used. In particular, we employed Docker and OpenStack;
- some estimates of the time needed to process the image data on different environments have been provided.

The thesis is organized as follows: in Chapter 1 we will present the NFFA-EUROPE project and we will frame our work within it; in Chapter 2 we will offer an overview of machine learning techniques and tools used in thoughout our work; in Chapter 3 we will explain the main Spark concepts and how we set the environment up to employ it; scientific and technical results are discusses in Chapter 4 and Chapter 5, respectively; finally, in Chapter 6 we will present our conclusions and future perspectives.

Contents

List of Figures

Chapter 1

The NFFA-EUROPE Project

This first Chapter aims to frame the activities of this thesis within the NFFA-EUROPE [2] . The project is an integrating activity carried out in the Horizon 2020 - Work Programme for European research infrastructures. The overall objective of NFFA-EUROPE is to implement the first integrated, distributed research infrastructure as a platform supporting comprehensive user projects for multidisciplinary research at the nanoscale.

One of the key aims of NFFA-EUROPE is to create the first overarching Information and Data management Repository Platform (IDRP) for nanoscience, defining a metadata standard for data sharing as an open collaborative initiative within the framework of the Research Data Alliance (RDA) [3]. The CNR-IOM [5] coordinates the Joint Research Activity 3 (JRA3) proposed by NFFA-EUROPE, which will develop and deploy the IDRP. This repository, through a suitable open-data access policy, will be a novel and unique instrument for knowledge transfer. It is therefore within the IDRP activities of the NFFA-Project that our thesis was developed. In the following, we will present the basic design of IDRP, the past activities already performed and which was our starting point, and finally a description of the scientific instrument we are targetting in this work: the SEM.

1.1 IDRP

The goal of the IDRP is to provide its users not only with high performance data access, but also with data sharing to promote collaboration and interoperability. Moreover, its semantic search engine will allow users to look for scentific related data among different resources. In this respect, it is of great importante to have a clear identification and organization of metadata associated with scientific data coming from different experimental and theoretical sources.

The design of the repository architecture is based on the recommendations of the European Commission [4] and the data fabric of the RDA, and employs the results provided by the European projects EUDAT [6] and PaNdata [7]. The IDRP will include all the information on a given user project, from proposal to data analysis, relevant calibrations, and references. The core of the IDRP architecture will be based on the KIT Data Manager (DM) of Karlsruhe Institute of Technology [8], a multi-layered service architecture integrating software and technologies to build up repository systems able to manage Big Data. The distributed repository architecture is based on the data infrastructures that are already used in the nanoscience facilities, and will extend them to enable long-term preservation, data sharing, discovery, publication, access and exportation, and will foster their interoperability to enable Transnational Access Activities.

A dataset in the repository consists of raw data, analysis data and metadata satisfying the general requirements for data reuse. The Single Entry Point (SEP) portal acts as a central access point for users to access the IDRP based on single-sign-on and for data discovery based on search with the describing metadata.

The IDRP consists of two repositories:

- a repository for published data, providing a persistent preservation of data and metadata;
- data discovery, publicly accessible. Registration is necessary to download data in order to restrict the load of the infrastructure.

The IDRP Management serves as a dataset registry for the facilities and handles the data organization. Metadata already available at the facilities are extended to offer additional data services, e.g., for discovery and policy enforcement. Multiple measurements of a probe in various facilities may be shared and merged to one dataset for common analysis and cross calibration. Locally available services and data infrastructures (based on the KIT DM) at the facilities may be integrated seamlessly, whenever possible. Also some architecture components (for example, the content metadata management and interfaces to the NFFA Distibuted Repository Management) may be deployed at the facilities to extend their local infrastructure.

It is worth noticing that nanoscience poses new challeges related to the complexity of metadata, which are not uniquely defined among such a wide multidisciplinary area. Further challenges includes the extreme diversity and heterogeneity of the infrastructures and of the data, which can be collected as binary files (in some cases even with proprietary format), images, plain text, videos, and so on. Taking these aspects into account, the overall architecture of the IDRP will facilitate:

- seamless integration of existing technologies;
- interoperability for internal technology changes by standardized interfaces, for example OAI-PMH for metadata harvesting;
- dataset registration and discovery;
- integration of data access policies as defined by the project:
  - registration of datasets by one responsible machine/user;
  - mandatory registration for all users to discover and access data;
  - granting discovery and access for selected datasets;
  - published datasets visible to public; access and download only for registered users.

1.1.1 The prototype

As already mentioned in Sec. 1.1, the KIT DM has been chosen as the initial core of the IDRP architecture thanks to its multi-layered service architecture and its several services available. A beta release of the KIT tool has been evaluated and tested, and a first NFFA prototype of its global data repository has been built upon it. The prototype has been connected to three different experimental facilities, representative of the NFFA Centre: a Scanning Electron Microscope (SEM) instrument, an Advanced Photoelectric Effect (APE) beamline for spectroscopy, and an open code for first principle simulation (Quantum Espresso). The IDRP acquires data from the above facilities by means of a set of plugins which interact through the restful interface of the KIT DM. The overall prototype has been build on top of a cloud infrastructure with a set of virtual machines playing the role of the computer devices located in the experimental facilities. For each experimental device a virtual machine was installed, and the injection plugin was tested on a set of preloaded data provided as representative by selected users. The plugins are able to allow an automatic harvest of metadata associated to data produced by NFFA scientific users. They are ready to be installed on the computing devices available at the experimental facilities (APE and SEM) while the plugin for Quantum Espresso will be installed on the CNR-IOM HPC facility, where the majority of the computations are performed. All the plugins are then available to the final users, who can inject the data on the experimental/computational facility.

1.1.2 The SEM plugin

The SEM plugin is fully integrated with the Python interface provided by KIT DM. Images generated from SEM are saved as `tif` in the Tagged Image File Format (TIFF), a file format able to save some metadata about the image in a header of the file. The plugin collects in an automated way the `tif` images and their associated metadata (information about the user, the sample and the instrument), and integrates them in a unique `HDF5` file [11] that is then automatically loaded into the KIT DM. The plugin works in an automatic way, harvesting all the data stored in a specific location (directory). Several tests were performed on SEM images, including data injection into the KIT DM. In all the cases, the `HDF5` files have been automatically submitted to the KIT DM using the webdav protocol. At the same time, the collected metadata associated to the files loaded into the KIT DM are sent to an Elasticsearch [9] server, to guarantee a search engine for the users. In the prototype, the server is external to the KIT DM for testing purposes, but is fully integrated in the virtual environment. The SEM plugin represents the starting point of our activities.

1.2 SEM images

Scanning electron microscopy (SEM) is a common characterisation technique used by nanoscientists to view their samples at higher magnifications than possible with traditional optical microscopes. The range of magnification on a SEM is 250-500,000 times. Whilst visible light has a wavelength range of 400-700 nm, an accelerated electron can have a wavelenth $\lambda \lesssim 1$ nm, allowing nanometre resolution to be achieved when using this technique. SEM works by scanning a focused beam of electrons over the surface of a sample. The interaction of the electron beam and the sample results in the release of secondary electrons, which are collected by a detector. The number of secondary electrons detected depends on a number of factors, such as the atomic radius of the sample atoms and the topography of the surface. By raster scanning over an area, the intensity of the detected signal can be used to build a grayscale image of the topography of a sample. Typically, SEM samples should be electrically conductive and grounded to prevent the build-up of electrostatic charge. Non-conductive samples are often imaged by coating them with a very thin conductive layer of gold or platinum.

SEM is a popular technique amongst nanoscientists, since it is a versatile tool that can be used to view samples at high magnifications and to check whether the desired stucture has been obtained without much prior sample preparation required. For material scientists, this could be checking the morphology and shape of nanostructures. Some scientists and engineers who fabricate devices, such as transistors, microfluidics or other micro-/nano-scale devices on a chip, use SEM to verify the correct fabrication of their devices before or after they are used. Each sample is often view at different magnifications. For example, a patterned surface might be viewed at a low magnification to check whether the patterns have been developed correctly over large areas, and at high magnification to check the smoothness of the edges of individual patterns. As will be discussed in Sec. 4.3, this can provide significant challenges for image recognition, since the same object can look very different depending on the magnification used.

Training a neural network on SEM images can offer the following benefits for nanoscience researchers:

- automatic classification of images, avoiding the need for the users having to manually classify each image they produce;
- providing a searchable database of nanoscience images which can allow scientists to find specific category of SEM images.

Chapter 2

Machine Learning with TensorFlow

In this Chapter we will present a brief overview of machine learning techniques and tools used in our work. We will describe the basic concepts of machine learning in Sec. 2.1, then we will give an overview, mainly based on [15], of deep feedforward networks in Sec. 2.2 and of convolutional neural networks in Sec. 2.3. TensorFlow and Inception-v3 are presented in Sec. 2.4, while Sec. 2.5 is dedicated to a description of how we set our work environment up.

2.1 Machine Learning Basics

2.1.1 Learning Algorithms

According to the definition of [16] "a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". In other words, a machine learning algorithm is an algorithm that is able to learn from data.

The Task T

Machine learning tasks are usually described in terms of how the system should process an *example*, which is defined as a collection of *features* that have been quantitatively measured from some object or event that we want the machine learning system to process. An example is typically represented as a vector $\vec{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature. In our specific case, the features of an image are the pixel brightness values.

Many kinds of tasks can be solved with machine learning; we are interested in *classification*. In this type of task, the program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is asked to produce a function $f\colon \mathbb{R}^n \to \{1, \ldots, k\}$. When $y = f(\vec{x})$, the model assigns an input described by $\vec{x}$ to a category identified by the numeric code y. In our specific case of object recognition, the input is an image, described as a set of pixel brightness values, and the output is a numeric code identifying the object in the image, converted into a human readable label.

The Performance Measure P

In order to evaluate the abilities of a machine learning algorithm, a quantitative measure of its performance is needed. Usually this performance measure P is specific to the task T being carried out by the system. For tasks such as our classification, we measure the *accuracy* of the model, which is defined as the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the *error rate*, defined as the proportion of examples for which the model produces an incorrect output. The error rate is often referred to as the expected 0-1, which is 0 if a particular example is correctly classified and 1 if it is not.

Usually, we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in a real case. We therefore evaluated P using a *test set* of data that had been kept separate from the data used for training.

The Experience E

Most machine learning algorithms can be broadly categorised as *supervised* or *unsupervised* according to what kind of experience on a *dataset* (defined as a collection of many examples) they are allowed to have during the learning process. Unsupervised learning involves observing several examples of a random vector $\vec{x}$, and attempting to implicitly or explicitly learn the probability distribution $p(\vec{x})$ of that distribution. Supervised learning instead involves observing several examples of a random vector $\vec{x}$ and an associated value or vector $\vec{y}$, and learning to predict $\vec{y}$ from $\vec{x}$, usually by estimating the conditional probability $p(\vec{y}|\vec{x})$. Roughly speaking, unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of its structure, while supervised learning algorithms experience a dataset containing features, but each example is also associated with a label. This is our case with SEM image recognition.

Example: Linear Regression

We present here an example of a simple machine learning algorithm: the *linear regression.* The goal is to build a system that takes a vector $\vec{x} \in \mathbb{R}^n$ as input and predicts the value of a scalar $y \in \mathbb{R}$ as output. In the case of linear regression, the output is a linear function on the input. Let $\hat{y}$ be the value that our model predicts y should thake on. We define the output to be:

$$\hat{y} = \vec{w}^T \vec{x} \ , \tag{2.1}$$

where $\vec{w} \in \mathbb{R}^n$ is the vector of weights.

Suppose we have a matrix of m example inputs $X^{(\text{test})}$, used only as a test set for evaluating the performance of the model, and a vector of regression targets $\vec{y}^{(\text{test})}$, providing the correct value of y for each of these examples.

The task T is to predict y from $\vec{x}$ by outputting $\hat{y}$ given by Eq. 2.1. The performance measure P of the model can be obtained by computinig the mean squared error on the test set. If $\hat{\vec{y}}^{(\text{test})}$ gives the predictions of the model on the test set, then the mean squared error is given by:

$$\text{MSE}_{\text{test}} = \frac{1}{m}\sum_i (\hat{\vec{y}}^{(\text{test})} - \vec{y}^{(\text{test})})_i^2 \ . \tag{2.2}$$

To make a machine learning algorithm, we need to improve the weights $\vec{w}$ in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(X^{(\text{train})}, \vec{y}^{(\text{train})})$. For example, we can minimize the mean squared error on the training set by solving:

$$\nabla_w \text{MSE}_{\text{train}} = 0 \tag{2.3}$$

which gives:

$$\vec{w} = (X^{(\text{train})T} X^{(\text{train})})^{-1} X^{(\text{train})T} \vec{y}^{(\text{train})} \tag{2.4}$$

The system of equations whose solution is given by Eq. 2.4 is known as *normal equations*. Evaluating Eq. 2.4 constitutes a simple learning algorithm.

2.1.2 Generalization

The central challenge in machine learning is the *generalization*, defined as the ability to perform well on new, previously unobserved inputs. When training a machine learning model, a training set is used: an error measure called the *training error* can be computed and minimized. This is simply an optimization problem. What distinguishes machine learning from optimization is the additional requirement of the *generalization error*, defined as the expected value of the error on a new input, to be low as well. The generalization error of a model is usually estimated by measuring its performance on a test set, which is why it is also called *test error*. When using a machine learning algorithm, the training set is sampled and used to choose the parameters in order to reduce the training error, then the test set is sampled. Under this process, the expected test error is greater or equal to the expected training error. The factors determining how well an algorithm will perform are:

1. its ability to make the training error small;
2. its ability to make the gap between training and test error small.

These two factors correspond to *underfitting* and *overfitting*, respectively. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and the test error is too large.

The way to control whether a model is more likely to overfit or underfit is by altering its *capacity*, which is its ability to fit a wide variety of functions. Machine learning algorithms generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with low capacity are unable to solve complex tasks because they may struggle to fit the training set; models with high capacity may overfit by memorizing properties belonging to the training data it has seen and is unable to generalize to unseen test data. We show an example of this principle in Fig. 2.1.

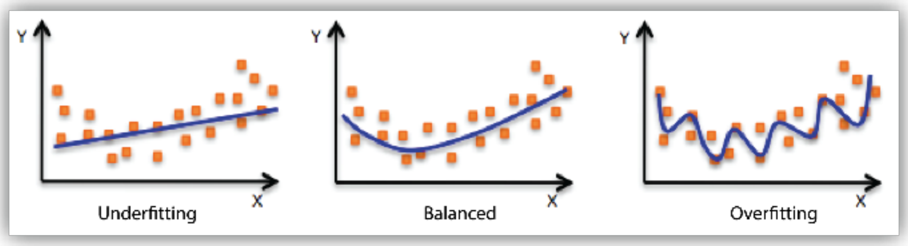


Figure 2.1: Simple example of a function which underfits (left panel), correctly fits (central panel) and overfits (right panel) the same data sample. Figure has been taken from [33].

2.1.3 Parameters and Hyperparameters

Parameters are values that control the behaviour of the system. In our case, we can think of a vector of parameters $\vec{w} \in \mathbb{R}^n$ as a set of *weights* that determine how each feature affects the prediction. If a feature x_i receives a positive weight w_i, then increasing the value of that feature increases the value of the prediction. If a feature receives a negative weight, then increasing the value of that feature decreases the value of the prediction. If a feature's weight is large in magnitude, than it has a large effect on the prediction, while if its weight is zero, it has no effect on the prediction.

Most machine learning algorithms have several settings that can be used to control their behaviour, called *hyperparameters*. Unlike parameters, the values of hyperparameters are not adapted by the learning algorithm itself. Frequently, the settings must be hyperparameters because it is not appropriate to learn them on the training set. This applies to all hyperparameters controlling the model capacity. If learned on the training set, they would always choose the maximum possible model capacity, resulting in overfitting. To solve this problem, the subset of data used to guide the selection of hyperparameters, called *validation set*, is needed. Since the test examples do not have to be used in any way to make choices about the model, the validation set is always constructed from the training data. Specifically, the training data is split into two disjoint subsets: one of them is used to learn the weights, the other is used to estimate the generalization error during or after training, allowing the hyperparameters to be updated accordingly.

2.2 Deep Feedforward Networks

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MPLs), are the most typical deep learning models. The goal of a feedforward network is to approximate some function f^*. As an example, a classifier $y = f^*(\vec{x})$ maps an input $\vec{x}$ to a category y. A feedforward network defines a mapping $\vec{y} = f(\vec{x}; \vec{w})$ and learns the value of the weights $\vec{w}$ that result in the best function approximation. These models are called *feedforward* because information flows through the function being evaluated from $\vec{x}$, through the intermediate computations used to define f, and finally the output

$\vec{y}$. There are no feedback connections in which an output of the model is fed back into itself.

They are called *networks* because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\vec{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\vec{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the *first layer* of the network, $f^{(2)}$ is called the *second layer*, and so on. The overall length of the chain gives the *depth* of the model. It is from this terminology that the name "deep learning" arises. The final layer of a feedforward network is called *output layer*. During neural network training, we drive $f(\vec{x})$ to match $f^*(\vec{x})$. The training data provides us with noisy, approximated examples of $f^*(\vec{x})$ evaluated at different training points. Each example $\vec{x}$ is accompained by a label $y \approx f^*(\vec{x})$. The training examples specify directly that the output layer must produce a value close to y at each point $\vec{x}$. The behaviour of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, which means to best implement an approximation of f^*, but the training data does not show the desired output for each of these layers: for this reason, they are called *hidden layers*.

Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the *width* of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of it as consisting of many *units* that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it rececives input from many other units and computes its own activation value.

2.2.1 Hidden Layers

The role of hidden units can be described as applying to an input vector $\vec{x}$ a nonlinear function $\phi(\vec{x}; \vec{w})$ which provides a set of features describing $\vec{x}$ (or simply a new representation for $\vec{x}$). This function is usually built by computing an affine transformation controlled by learned weights, followed by a fixed, element-wise, nonlinear function called *activation function*. Most hidden units are distinguished from each other only by the choice of the form of the activation function. In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU [19, 20, 21], defined by the the activation function $R(z) = \max\{0, z\}$, where z is the individual output of the affine transformation on x. The ReLU activation function is shown in Fig. 2.2. We will see in Sec. 2.4.2 that all the hidden layers of the Inception-v3 architecture [13] use this function.

2.2.2 Stochastic Gradient Descent

A recurring problem in machine learning is that large training sets are necessary for a good generalization, but they are also computationally expensive. For

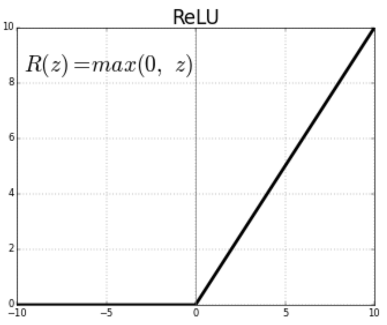


Figure 2.2: The rectified linear unit (ReLU) activation function. The Figure has been adapted from [15].

example, the computational cost of a gradient descent operation is $O(m)$, where m is the number of training data. This means that as the training set size grows, the time to take a single gradient step may become prohibitively long. The insight of *stochastic gradient descent*, an approximation of the gradient descent algorithm, is that the gradient is an expectation, thus it may be approximately estimated using a small set: on each step of the algorithm, a *minibatch* $\mathbb{B} = \{\vec{x}^{(1)}, \ldots, \vec{x}^{(m')}\}$ may be drawn uniformly from the training set. The minibatch size m' is usually chosen to be a relatively small number of examples, ranging from 1 to a few hundred. The crucial aspect is that m' is held fixed as the training set size m increases. The estimate of the gradient for the minibatch $\mathbb{B}$ becomes:

$$\vec{g} = \frac{1}{m'} \nabla_{\vec{w}} \sum_{i=1}^{m'} L(\vec{x}^{(i)}, y^{(i)}, \vec{w}) \ , \tag{2.5}$$

where L is the per-example loss $L(\vec{x}, y, \vec{w}) = -\log p(y|\vec{x}; \vec{w})$. The stochastic gradient descent algorithm then follows the estimated gradient downwards:

$$\vec{w} \leftarrow \vec{w} - \epsilon \vec{g} \ , \tag{2.6}$$

where ϵ is the learning rate, a positive scalar determining the size of the step.

The cost per update does not depend on the training set size, so the asymptotic cost of training a model is $O(1)$ as a function on m. This is the reason why stochastic gradient descent is used in nearly all of deep learning algorithms.

2.3 Convolutional Neural Networks

Convolutional neural networks [22] are a specialyzed type of neural networks suited for processing data that has a grid-like topology, for example image data, which can be thought of as a 2D grid of pixels. As the name indicates, the network employs a mathematical operation called *convolution*, which is a particular kind of linear operation. We will briefly describe it in Sec. 2.3.1. Almost all convolutional networks make use of an operation called *pooling*, described in Sec. 2.3.2. It is also worth mentioning in Sec. 2.3.4 the softmax function, which will be useful for our classification purposes.

2.3.1 Convolution

Given a function $x(t)$, usually referred to as the *input*, and a weighting function $w(t)$, referred to as the *kernel*, both dependent on an index variable t, the convolution operation is defined as:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{2.7}$$

In general, convolution is defined for any function for which the above integral is defined. Intuitively, it performs a weighted average operation on the input, giving as output a so-called *feature map*. Of course, when working with data on a computer, the index variable will be discretized, so we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_a x(a)w(t-a) \tag{2.8}$$

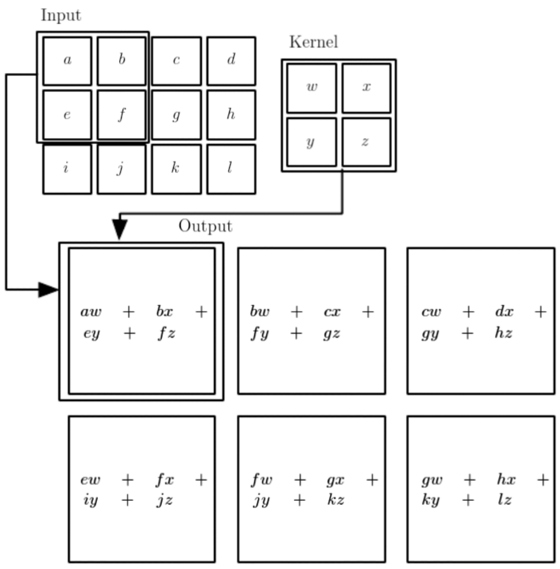


Figure 2.3: Example of 2D convolution. For simplicity, the output is restricted only to positions where the kernel lies entirely within the image, called "valid" convolution. Boxes with arrows indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor. The Figure has been taken from [15].

In machine learning applications, the input is usually a multidimensional array of data and the kernel is a multidimensional array of parameters that are adapted by the learning algorithm, namely the weights. These multidimensional arrays are referred to as *tensors*.

In our case, we will use convolutions over two axes, schematically represented in Fig. 2.3. For a 2D image I as our input, and a 2D kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{2.9}$$

2.3.2 Pooling

A typical layer of a convolutional network consists of three stages (see Fig. 2.4). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, called detector stage, each linear activation is run through a nonlinear activation function, such as the ReLU. In the third stage, a pooling function is used to replace the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the *max pooling* operation [23] used in the Inception-v3 architecture reports the maximum output within a rectangular neighbourhood.

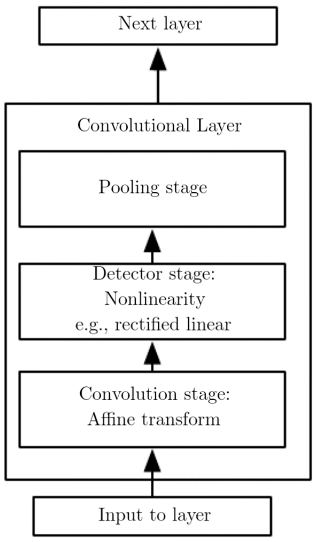


Figure 2.4: The components of a typical convolutional neural network layer. The Figure has been adapted from [15].

The role of pooling is to make the representation become approximately invariant to small translations of the input, which means that the values of most of the pooled output will not change if the input is shifted by a small amount. This is a very useful property in our case, where we are interested in detecting some features rather than in specifying their exact location. Moreover, another very important issue in our case is to classify images which may have different sizes, while the input to the classification layer must have a fixed size. Pooling is essential for this task, which can be accomplished by varying the size of an offset between multiple pooling regions, so that the classification layer always receives the same number of summary statistics regardless of the input size.

2.3.3 Dropout

Dropout is a technique for addressing the problem of overfitting (see Sec. 2.1.2) in deep neural networks with a large number of parameters. The term “dropout” refers to dropping out units (hidden and visible) in a neural network, which means temporarily removing them from the network, along with all their in-coming and outgoing connections, as shown in Fig. 2.5. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can be set to 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5, as reported by [10]; a typical choice is 0.8.

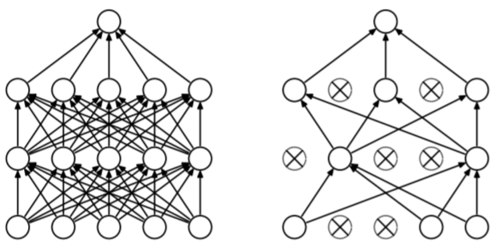


Figure 2.5: Dropout model. Left: a standard neural network with two hidden layers. Right: an example of a thinned network produced by applying dropout to the network on the left; crossed units have been dropped.

Applying dropout to a neural network amounts to sampling a "thinned" network from it. The thinned network consists of all the units that survived the dropout, as shown in the right panel of Fig. 2.5. Training a neural network with n units can be seen as training a collection of 2^n possible thinned neural networks, which all share weights so that the total number of parameters is still $O(n^2)$ (or less), where each thinned network gets trained very rarely, if at all. At test time, it is not feasibile to explicitly average the prediction from so many thinned models. However, a very simple approximate averaging method works well in practice. The idea is to use a single neural network at test time without dropout. The weights of this network are scaled-down versions of the trained weights. If a unit is retrained with probability p during training, the outgoing weights of that unit are multiplied by p at test time. This ensures that for a hidden unit the *expected* output (under the distribution used to drop units at training time) is the same as the actual output at test time. By doing this scaling, 2^n networks with shared weights can be combined into a single neural network to be used at test time. Training a network with dropout and using this approximate averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods.

2.3.4 Softmax

The softmax function is used any time there is the need to represent a probability distribution over a discrete variable with n possible values. In our case, it is used as the output of the classifier in the final layer, to represent the probability distribution over the nanoscience categories, producing a vector $\vec{y}$, with $y_i = P(y = i|\vec{x})$. To represent a valid probability distribution, the vector $\vec{y}$ is required to be such that $0 \leq y_i \leq 1$, and $\sum_i y_i = 1$. Formally, the softmax function is given by:

$$\text{softmax}(\vec{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \ , \tag{2.10}$$

where $\vec{z}$ is the unnormalized log probability $z_i = \log \tilde{P}(y = i|\vec{x})$, such that $P(y = i|\vec{x}) = \text{softmax}(\vec{z})_i$.

The use of the exponential function works very well when training the softmax to output a target value y using the maximum log-likelihood, as is done in most modern neural networks. In this case, the function that is requested to be maximized is:

$$\log P(y = i; \vec{x}) = \log \text{softmax}(\vec{z})_i = z_i - \log \sum_j \exp(z_j). \tag{2.11}$$

The second term in Eq. 2.11 can be roughly approximated by $\max_j z_j$ because $\exp(z_j)$ rapidly becomes negligible for any $z_j \ll \max_j z_j$. Thus, the negative log-likelihood cost function (used as a measure of how well the neural network performed in mapping training examples to the correct output) will always be dominated by the most incorrect predictions, while the correct answer will give a contribution close to zero.

2.4 TensorFlow and Inception-v3

TensorFlow [14] is an software library for numerical computation originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license in 2015. It is currently used for both research and production by tens of different teams in dozens of commercial Google products, such as speech recognition, Gmail, Google Photos, and Google Images.

In TensorFlow computations are represented as data flow graphs. Nodes in the graph are called *op* (short for operations), and represent mathematical operations, while the graph edges represent the tensors (see Sec. 2.3.1). An op takes zero or more tensors, performs some computations, and produces zero or more tensors. A TensorFlow graph is a description of computations. To compute anything, a graph must be launched in a *Session*. A Session places the graph ops onto devices, such as CPUs or GPUs, and provides methods to execute them. These methods return tensors produced by ops as numpy `ndarray` objects in Python, and as `tensorflow::Tensor` instances in C and C++.

2.4.1 The Computational Graph

TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph. For example, our code creates a graph to represent and train a neural network in the construction phase, and then repeatedly executes a set of training ops in the graph in the execution phase. The TensorFlow Python library has a default graph to which ops constructors add nodes.

To build a graph, it is necessary to start with ops that do not need any input (*source* ops), and pass their output to other ops that do computation. The ops constructors in the Python library return objects that represent the output of the constructed ops. The TensorFlow implementation translates the graph definition into executable operations distributed across available compute resources, such as the CPU or GPU cards. As a default, TensorFlow uses the first GPU available, for as many operations as possible. To modify this behaviour, it is necessary to add a `with tf.Device` statements to specify which CPU or GPU to use for operations. On the infrastructure we used through this work, described in Sec. 2.5, two GPUs are present, so we explicitly set the usage for both of them, whenever appropriate.

2.4.2 Inception-v3

Inception-v3 [13] is the 2015 iteration of Googles Inception architecture for image recognition. It has 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 for [26, 27].

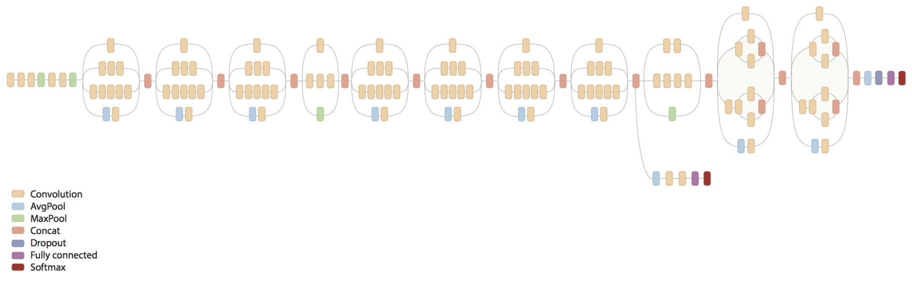


Figure 2.6: Inception-v3 architecture. The Figure has been taken from [24].

A schematic picture of such an architecture is shown in Fig. 2.6. The model is a multi-layer architecture consisting of alternating convolutions and nonlinearities, using ReLU activation functions in all the stages. These layers are followed by a fully connected layer leading to a softmax classifier.

The implementation of it was written by the same people who wrote TensorFlow, and so it employs many TensorFlow techniques. The official TensorFlow repository [24] has a working implementation of the Inception-v3 architecture. The Inception code uses TF-Slim [25], which is an abstraction library over TensorFlow that makes writing convolutional nets easier and more compact.

The training procedure employs synchronous stochastic gradient descent across multiple GPUs: replica of the model (training on a subset of data) is placed on each GPU, and model weights are updated synchronously by waiting for all GPUs to finish processing a batch of data. The diagram of this model is shown in Fig. 2.7. Each GPU computes inference and gradients for a unique batch of data. This setup requires that all GPUs share the model weights. Transferring data to and from GPUs is known to be quite slow. For this reason, all model weights are stored and updated on the CPU; a fresh set of model weights is transferred to the GPU when a new batch of data is processed by all GPUs. The GPUs are synchronized in operations: all gradients are accumulated from them and averaged. The model weights are updated with the gradients averaged across all model replicas.

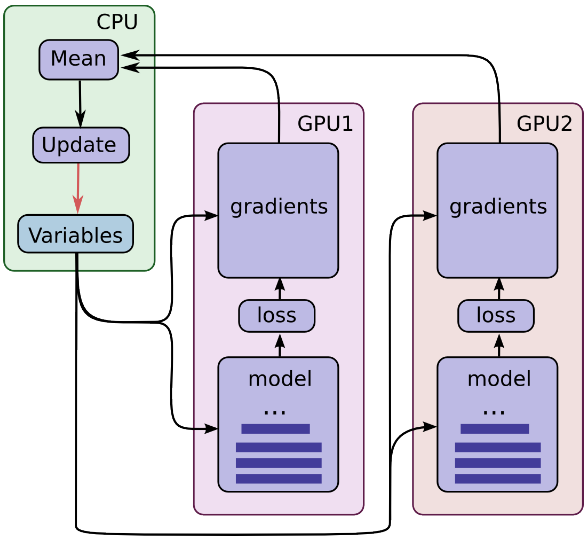


Figure 2.7: Diagram showing the Inception-v3 training procedure in TensorFlow. The Figure has been taken from [14].

The Inception-v3 implementation makes heavy use of the various scoping mechanisms available in TensorFlow. The entire Inception model is wrapped into a new TensorFlow op (Sec. 2.4): first, it is wrapped in an `op_scope` named `inception_v3`, then various `arg_scope`s are used to set the default arguments for ops inside the model. TensorFlows `arg_scopes` are a simple way of setting the default arguments for a lot of ops in a model at the same time, without having to repeatedly enter them each time an op is called. The list `end_points` in the model contains all intermediate tensors: for instance, `end_points['conv0']` contains the output of the first convolution, which is then fed into another convolution, the result of which is saved as `end_points['conv1']` and so on. Convolution, max-pooling, and dropout layers are repeatedly applied to the tensors, and the result is the `logits` variable, which gives the predictions (a vector of length equal to the number of categories for each image in the batch) and the `end_points` list which stores all intermediate results. To reduce over-fitting, dropout (see Sec. 2.3.3) is applied before the softmax layer. TensorFlow `tf.nn.dropout` op automatically handles scaling neuron outputs in addition to masking them (i.e., performing a pooling).

2.5 Environmental Setup

We have been provided with root access on a computational node, on the C3E Cloud Computing Environment of COSILT, located in Tolmezzo and managed by eXact-lab srl [30]. The cluster node is equipped with two Intel Xeon CPUs E5-2697 at 2.70 GHz (12 cores each, for a total of 24 cores) and two K20s Nvidia GPUs. The operative system installed on the node is CentOS 7, but the only Linux distribution TensorFlow runs on is 64-bit Ubuntu. Since Tensorflow provides a Docker image on the website, the best option has been to install a Docker container on the node and work inside it.

2.5.1 Docker

Docker [32] is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow wrapping a piece of software in a complete filesystem that contains everything needed to run, such as code, runtime, system tools, system libraries and other dependencies, and ship it all out as one package, becoming part of a base working image. This guarantees that the software will always run the same, regardless of its environment.

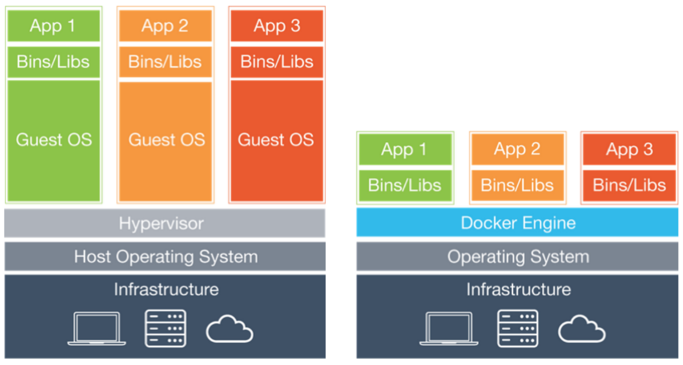


Figure 2.8: A comparison between the virtual machine (on the left) and the Docker container (on the right) model. The Figure has been taken from [32].

Docker Containers have much more potential than virtual machines. The virtual machine model blends an application, a full guest operative system (OS), and disk emulation. In contrast, the container model uses just the application's dependencies and runs them directly on a host OS. Containers do not launch a separate OS for each application, but share the host kernel while maintaining the isolation of resources and processes where required. The fact that a container does not run its own OS instance reduces dramatically the overhead associated with starting and running instances. Startup time can typically be reduced from 30 seconds (or more) to 0.1 seconds. For the same reason, performance of an application inside a container is generally better compared to the same application running within a virtual machine. The number of containers running on a typical server can reach dozens or even hundreds. The same server, in contrast, might support 10 to 15 virtual machines. A schematic illustration of virtual machine and Docker container models is shown in Fig. 2.8.

2.5.2 Docker installation

Installing CUDA

A preliminary step was to install DKMS (Dynamic Kernel Module Support), a program that enables generating Linux kernel modules whose sources generally reside outside the kernel source tree. To install CUDA, we downloaded the GPU drivers from the Nvidia website [31] and installed them. We generated some of the devices (`/dev/nvidia0`, `/dev/nvidia1`, and `/dev/nvidiactl`) using the `nvidia-smi` software, while `/dev/nvidia-uvm` has been generated by running for a few seconds the command `nvidia-cuda-mps-server`.

Installing Docker

The Docker installation required the configuration of a repository for yum with the RPM (originally Red Hat Package Manager, now a recursive acronym RPM Package Manager, a package management system). We created a file called `/etc/yum.repos.d/docker.repo` containing the following:

```
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
```

Finally, `yum install docker-engine` and `service docker start` allowed us to have a working Docker service, and we run our Docker container using the TensorFlow image provided on the Tensorflow website [14], with the additional flag `--device` to export the Nvidia devices.

Chapter 3

Distributed image processing with Spark

The SEM instrument provided a huge amount of images, so the obvious way to proceed is processing them in parallel. We decided to use Apache Spark [34], a new framework which is increasingly being used in the world of big data, mainly for faster processing. Spark utilizes in-memory capabilities to deliver fast processing (almost 100 times faster than Hadoop [35]). Moreover, in Spark it is possible to write applications that use machine learning to classify data in real time as it is ingested from streaming sources. We are very much interested in this aspect, since in the future perspective this will be the usecase for the processing of SEM images.

In this Chapter, we will first present the MapReduce model which Spark is based on (Sec. 3.1), then we will give an overview of Spark (Sec. 3.2), explaining why it is preferred to MapReduce (Sec. 3.2.4), and finally we will describe how we set the environment up for our work (Sec. 3.3).

3.1 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. As stated in the paper in which the MapReduce programming abstraction is presented [37], this approach was created to solve the three main aspects of parallel computations:

- distribution and balancing of the computing tasks;
- distribution of the data;
- fault tolerance

Programs written in this functional style are automatically parallelized and executed on a cluster while the runtime system takes care of the details of partitioning the input data, scheduling the program execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

3.1.1 The programming model

The MapReduce model consists of a *Map* function and a *Reduce* function, schematically illustrated in Fig. 3.1. The Map function takes a set of data and convert it into another set of data, by applying element-wise a given function, written by the user; individual elements are broken down into tuples (key/value pairs). The produced output is set of intermediate key/value pairs. The Reduce takes the output from a Map as input and combines those data tuples into a smaller set of tuples.

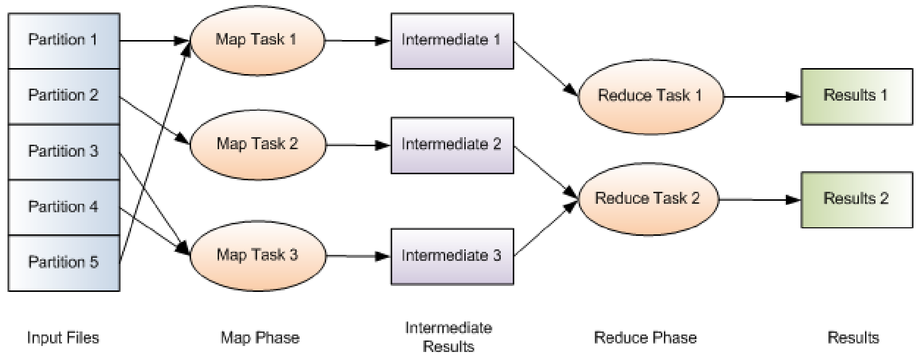


Figure 3.1: Schematic overview of a MapReduce job execution. The figure has been taken from [38].

The MapReduce library groups together all intermediate values associated with the same intermediate key, and passes them to the Reduce function, which is also written by the user. This function takes an intermediate key and a set of values for that key, and merges the values together to form a possibly smaller set of values. Typically, just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the Reduce function via an iterator; this allows handling lists of values that are too large to fit in memory.

3.1.2 Execution Overview

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits, which can be processes in parallel by different machines. The Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function, specified by the user.

Fig. 3.2 shows the overall flow of a MapReduce operation in the Google implementation [37]. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Fig. 3.2 correspond to the numbers in the list):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 MB per piece (controllable by the user through an optional parameter). It then starts up many copies of the program on a cluster of machines;

2. One of the copies of the program is the master, all the others are workers that are assigned work by the master. There are M Map tasks and R

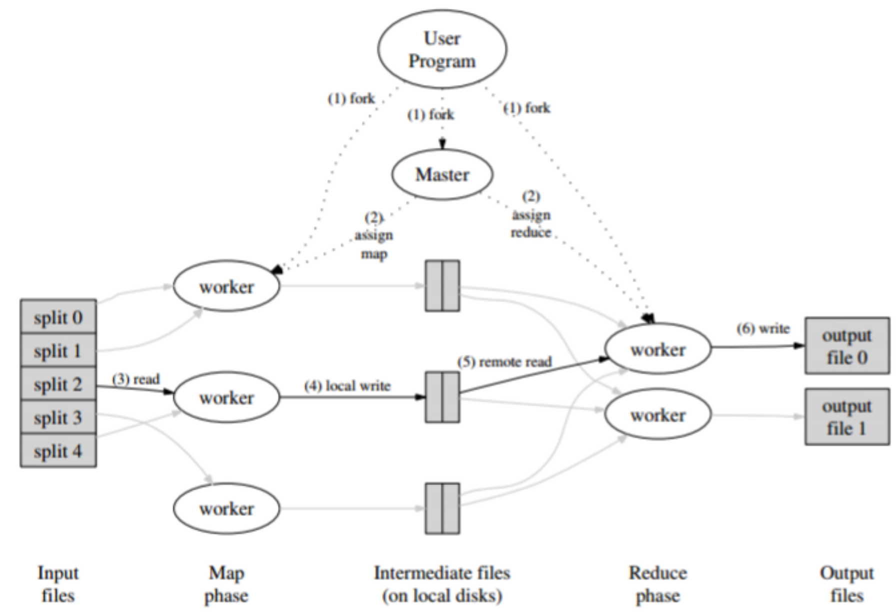


Figure 3.2: Execution overview of the Google MapReduce implementation. The figure has been taken from [37].

Reduce tasks to assign. The master picks idle workers and assigns each one a Map task or a Reduce task;

3. A worker who is assigned a Map task reads the content of the corresponding input split. It parses key/value pairs out of the input data, and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory;

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding their locations to the Reduce workers;

5. when a Reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the Map workers. When a Reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same Reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used;

6. the Reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user Reduce function. The output of the Reduce function is appended to a final output file for this Reduce partition;

7. when all Map tasks and Reduce tasks have been completed, the `MapReduce` call in the user program returns back to the user code.

3.2 Apache Spark

Apache Spark [34] is a cluster computing platform designed to be fast and general purpose. One of the main features Spark offers is the ability to run computations in memory. Moreover, the system extends the popular MapReduce model described is Sec. 3.1, and is more efficient than it for complex applications running on disk.

The Spark project contains multiple closely integrated components. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster*. Because the core engine of Spark is both fast and general-purpose, it powers multiple higher-level components specialized for various workloads, which are designed to interoperate closely. One of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models.

3.2.1 Spark's components

The Spark's components are shown in Fig. 3.3. Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interaction with storage systems, and more. Spark Core is also the location of the API that defines the *Resilient Distributed Datasets* (RDDs), which are the Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes, that can be manipulated in parallel. Spark SQL is a package for working with structured data. Spark Streaming is a Spark component that enables processing of live streams of data. MLib is a library that comes with Spark, containing common machine learning functionalities. GraphX is a library for manipulating graphs and performing graph-parallel computations. Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and Spark's own built-in Standalone cluster manager, called the Standalone Scheduler, which is the one adopted by us.

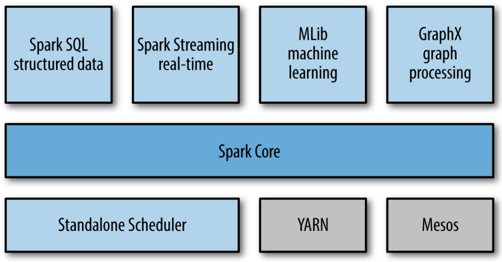


Figure 3.3: Schematic illustration of the Spark's components. The figure has been taken from [36].

3.2.2 RDD operations

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, `map` is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, `reduce` is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.

All transformations in Spark are *lazily evaluated*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.

By default, each transformed RDD may be recomputed each time an action is run on it. However, an RDD can be allowed to persist in memory by using the `persist` (or `cache`) method, in which case Spark will keep the elements on the cluster for much faster access the next time the RDD will be used. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

3.2.3 Spark runtime architecture

Spark provides a simple way to parallelize applications across clusters, and hides the complexity of distributed system programming, network communication, and fault tolerance. The system gives however enough control to monitor, inspect, and tune applications while allowing implementation of tasks. As mentioned in Sec. 3.2.2, computations in Spark are expressed through operations on distributed collections that are automatically parallelized across the cluster, namely the RDDs.

In distributed mode, Spark uses a master/slave architecture with one central coordinator and many distributed workers. The central coordinator is called the *driver*, which communicates with a potentially large number of distributed workers called *executors*. The driver runs in its own Java process and each executor is a separate Java process. A driver and its executors are together termed a Spark *application*. A Spark application is launched on a set of machines using an external service called *cluster manager*. As already mentioned, Spark is packaged with a built-in cluster manager called Standalone cluster manager. A schematic view of the components and how they are connected during a distributed execution of an application in Spark is shown in Fig. 3.4, and a brief description follows.

The Driver

The driver is the process where the `main` method of the program runs. The driver program accesses Spark through a `SparkContext` object, which represents a connection to the computing cluster. Once a `SparkContext` has been initialized, it can be used to build RDDs and to perform calculations on them, either *transformations* to derive new RDDs or *actions* to collect or save data. A Spark program implicitly creates a logical *Directed Acyclic Graph* (DAG) of operations. When the driver runs, it converts this logical graph into a set of

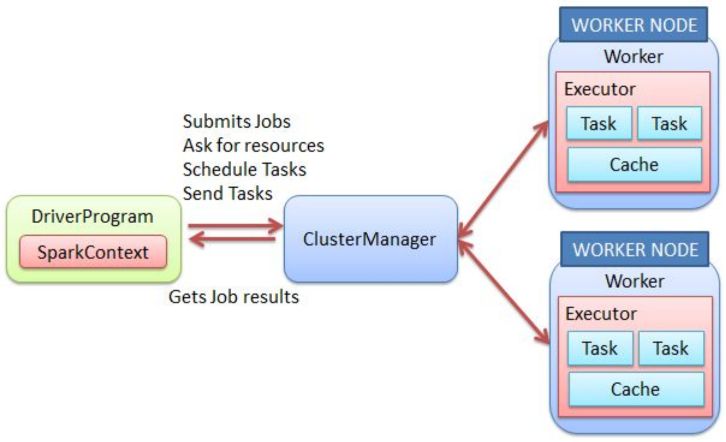


Figure 3.4: Schematic illustration of the components for a distributed execution of an application in Spark. The figure has been adapted from [36]

stages. Each stage, in turn, consists of units of physical execution called *tasks*, which are bundled up and prepared to be sent to the cluster.

The Executors

Spark executors are worker processes responsibile for running the individual tasks in a given Spark job. They are launched once at the beginning of an application and typically run for the entire lifetime of an application, though Spark applications can continue if executors fail. Executors have two roles:

- they run the tasks that make up the application and return results to the driver;
- they provide in-memory storage for RDDs that are cached by user programs, through a service called the Block Manager, which lives within each executor.

The Cluster Manager

Spark depends on a cluster manager to launch executors and, in certain cases, to launch the driver (*local mode*). Its main roles are:

- scheduling information, such as the amount of resources requested by the application for the job;
- keeping information about the runtime dependencies of the application, such as libraries of files that must be deployed to all worker machines.

As already mentioned, the cluster manager is a pluggable component; this allows Spark to run on top of different external managers, such as YARN and Mesos, as well as its build-in Standalone cluster manager.

3.2.4 Why Spark is faster than Hadoop MapReduce

Apache Hadoop [35] is a software framework written in Java for distributed data storage and computing, offering an open source implementation of MapReduce. One of the main limitations of Hadoop MapReduce is that it persists the full dataset to the Hadoop File Sistem (HDFS) after running each job. This is very expensive because it incurs three times replication of the size of the dataset in disk I/O and a similar amount of network I/O. Spark takes a more holistic view of a pipeline of operations. When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage. This is an innovation over Hadoop MapReduce that came from Microsoft's Dryad paper [39], and is not original to Spark.

The main innovation of Spark was to introduce an in-memory caching abstraction. This makes Spark ideal for workloads where multiple operations access the same input data. Users can instruct Spark to cache input data sets in memory, so they don't need to be read from disk for each operation.

Another advantage Spark has is that it can launch tasks much faster than Hadoop MapReduce. The latter starts a new Java Virtual Machine (JVM) for each task, which can take seconds with loading JARs, parsing configuration XML, and so on. Spark instead keeps an executor JVM running on each node, so launching a task is simply a matter of making a remote procedure call to it and passing a runnable to a thread pool, which takes milliseconds.

3.3 Environmental setup

Our approach to Spark has been the following:

1. in order to get acquainted with infrastructure and methodology, we deployed and tested a toy program on an already set up Spark implementation, namely Databricks [40], described in Sec. 3.3.1;

2. we deployed our own virtual Spark cluster on an OpenStack cloud provided by CNR-IOM [5], described in Sec. 3.3.2. This allowed us to check the complexity of the installation, the configuration, and the full deployment of a real program. Such a solution revealed several weak points: no shared filesystem, no control on real performance, and executor instabilities. These points will be discussed in Sec. 5.2.

3. our final step was the installation of Spark on a bare metal system, namely a node on the COSILT cluster, through a Docker container. The setting up procedure is described in Sec. 3.3.3. We evaluated the performance on this architecture and compared it with the cloud environment's one. We also reported estimates on the usage of different file systems, in order to understand which is the most suited one to deal with large amounts of image data (see Sec. 5.3).

3.3.1 Databricks community edition

Our first approach with Spark has been done employing the Databricks Community Edition, which is the free version of Databricks cloud-based big data platform [40], hosted on Amazon Web Services, with an already set up Spark implementation. With the Databricks Community Edition, we have been provided access to a 6GB micro-cluster as well as a cluster manager acting itself as a worker, and the notebook environment to prototype simple applications. We used the Python notebook environment to develop the parallel version of our serial code for image recognition and to familiarize with the Spark metodology.

3.3.2 Instances on Openstack

OpenStack [41] is an open source cloud management infrastructure, that controls large pools of computing, storage, and networking resources throughout a datacenter. It aims to be simple to implement and massively scalable. OpenStack consists of several independent parts, named the OpenStack services. All services authenticate through a common Identity service. Individual services interact with each other through public APIs, except where privileged administrator commands are necessary. Internally, OpenStack services are composed of several processes. All services have at least one API process, which listens for API requests, preprocesses them and passes them on to other parts of the service. With the exception of the Identity service, the actual work is done by distinct processes. Users can access OpenStack in many ways, for example via the web-based user interface implemented by *Dashboard* or via command-line clients. The Dashboard is a modular Django web application providing a graphical interface to OpenStack services, such as launching an instance, assigning IP addresses and configuring access controls. It allows managing of OpenStack resources and services and interaction with the OpenStack Compute cloud controller using the OpenStack APIs. *Instances* are the individual virtual machines that run on physical compute nodes inside the cloud. Each launched instance runs from a copy of the base *image*. When an instance is launched, a *flavor* must be chosen, which represents a set of virtual resources. Flavors define virtual CPU number, RAM amount available, and ephemeral disks size.

We launched 9 `m1.medium` instances, each of them characterized by 2 virtual CPUs, a 40 GB disk and a 4 GB RAM. To setup our cluster, we wrote an installation bash script to be executed as superuser on each instance. The script first installs Java:

```
add-apt-repository ppa:webupd8team/java
apt-get update
apt-get install oracle-java8-installer
```

Then downloads Spark from the archive page [42] and creates a soft link:

```
ln -s spark-2.0.1-bin-hadoop2.7 spark
```

Change the owner (which is root since the script is running as superuser):

```
chown ubuntu.ubuntu spark
```

Adds the following lines in the `.bashrc` file:

```
SPARK_HOME=/home/ubuntu/spark
export PATH=$SPARK_HOME/bin:$PATH
```

Then downloads TensorFlow from the webpage [14] and install it and other dependencies, in particular Pillow and h5py, through `pip`.

One of the instances has been used as driver. On it we started the standalone master server by executing:

```
./sbin/start-master.sh
```

Once started, the master prints out a `spark://HOST:7077` URL for itself, which is used to connect workers to it.

The remaining 8 instances have been used as workers. To allow them to connect to the master, its IP has to be put into the `/etc/hosts` file of each worker. After having verified ability to ping the master, we started the standalone slave server on each instance and connected them to the master by executing:
`./spark/sbin/start-slave.sh spark://HOST:7077`

We finally ended up with a 16 core cluster on which perform the image processing. We did not set a distributed file system up, so we equipped each instance with a local copy of the data, consisting of a sample of 1000 `tif` images and a sample of 2000 `tif` images.

3.3.3 Dockers on the COSILT node

To compare the performance obtained on a virtual environment with the ones on bare metal, we used a computational node of the COSILT cluster, equipped with two Intel Xeon CPUs E5-2697 at 2.70 GHz (12 cores each, for a total of 24 cores).

We installed Spark and all the needed dependencies on a Docker container built from a TensorFlow image available on the TensorFlow website [14], and we exported it as our Spark-TensorFlow image, from which we built 3 Dockers. Dockers containers can be configurated in order to provide them a well defined amount of resources. We run a driver Docker with 1 CPU, and two worker Dockers with 12 CPUs each. This has been possible adding at running time the flag `--cpuset-cpus=""`, putting the IDs of the CPUs which allow the execution.

On the node architecture, three different ways are available to store our own dataset: a local file system to the node itself, a shared Network File System (NFS), and a Lustre [12] parallel file system. To make our implemented Spark cluster able to read and write on these file systems, we mounted on each Docker three folders, and we put the image samples in all of them.

Using data stored on the local file system offers the advantage of consuming no network bandwidth. On the other hand, some of the most notable benefits that the NFS can provide are:

- Local workstations use less disk space because commonly used data can be stored on a single machine and still remain accessible to others over the network.
- There is no need for users to have separate home directories on every network machine. Home directories could be set up on the NFS server and made available throughout the network.

Lustre instead runs on commodity hardware and uses object based disks for storage and metadata servers for storing file system metadata. This design provides a substantially more efficient division of the workload between computing and storage resources. Distributed Object Storage Targets (OSTs) are responsible for actual file system I/O. This leads to a scalable file system and more reliable recoverability from failure conditions. Lustre also supports strong file and metadata locking semantics to maintain total coherency of the file systems even in the presence of concurrent access.

Chapter 4

Scientific Results

In this Chapter, we will present the scientific results achieved through the machine learning/neural network approach. These results address the initial part of the ambitious work we are doing to provide a searchable database to the nanoscience community. The incremental steps to reach our purposes are the following:

1. agree upon the most suitable criteria to classify the SEM images and provide a label for each image in the training set.(Sec. 4.1);

2. retrain the last layer of a pre-trained neural network based on Inception-v3 architecture using supervised learning (Sec. 4.2);

3. employ the TensorFlow image recognition to label the SEM images with the category it is supposed to belong to (together with the probability score), and collect statistics on the obtained results (Sec. 4.3). This has been done to provide directions and hints on how to use the system we have designed to the final scientific users.

4.1 Classifying SEM images

As already mentioned, our work is based on supervised learning. Thus, in order for the neural network to be trained, labelled training data must be provided, from which the neural network is able to generalise to unseen images. This section explains the process by which SEM images were classified into a labelled training set.

For the training to be effective, the main general rules for selecting the training images are the following:

- at least 100 images for each category should be provided;

- the images should be a good representation of what the application will be asked to recognize once trained;

- particular attention should be paid to anything that the labeled images have in common, to avoid the learning process acquiring irrelevant or wrong characteristics.

Due to the highly diverse nature of nanoscience research and the wide range of scientists and engineers who use the SEM, the images produced through this technique also encompass a broad range. Often the images produced can look very different to what we see in our macroscopic world. Nanoscience research is sometimes categorised by its function, or its method of production. For example, "top-down" approaches to making nanostructures often involve creating smaller structures by adding and removing material from a bulk, similar to a sculptor carving a statue. These methods have been extensively explored, and are used to produce devices such as transistors in computer chips, and micro-electromechanical (MEMS) devices such as accelerometers which are used in mobile phones. On the other hand, "bottom-up" approaches involve assembling and arranging molecular and atomic starting material to form larger structures, as is done for molecular motors, which are capable of rotating when they receive input energy. There are commonalities between SEM images of some structures formed in the same way: for example, those structures formed by top-down methods, such as lithography, are often similar in that they have repeated patterns and smooth edges. However, this is not always the case. Structures formed from chemical synthesis can have a variety of different final shapes, structures, and morphologies.

Since the neural network can only be trained by the pixels within the images, it was important to develop a classification system which was dependent on the shape of the nanostructure, rather than the more abstract and subjective notion of its function. However, where possible, the classification was chosen such that the categories whould be relevant to the work of the scientists, and close to the way how they might tag or label the image themselves.

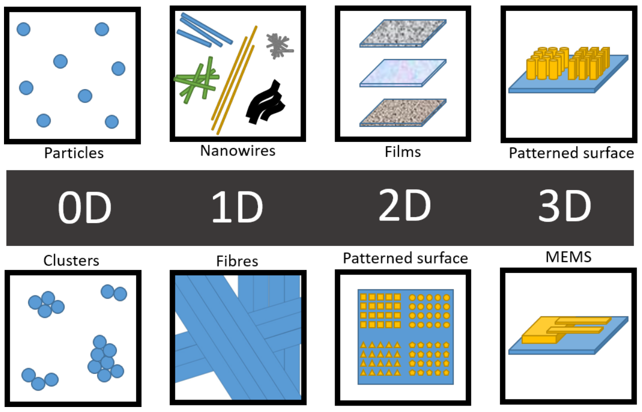


Figure 4.1: Schematic representation of a classification for nanostructures based on their dimensionality.

One way of classifying nanostructures based on their shape and structure is their dimensionality. We can classify nanostructures as 0D, 1D, 2D, and 3D objects, as schematically illustrated in Fig. 4.1. In this classification, 0D objects refer to particles which can be dispersed and isolated over an area of the sample, or clustered together. 1D objects are often referred to as nanowires, rods, or fibres. These structures are often packed together, either in an unorganized entangled way, or aligned parallel to one another. 2D structures refer to films and coatings on a surface, which can be formed from a variety of different materials with a range of surface topologies. Some surfaces can seem smooth and flat on a SEM, whilst others are made of small particles packed together, covering the entire surface. 3D structures can refer to pillars, or other devices like MEMS, typically fabricated using lithographic processes.

The dimensionality is both visually distinct, allowing it to be classified by the neural network, as well as being meaningful, since many of the properties of nanostructures depends on their dimensionality. Nevertheless, after further investigation of SEM images, additional categories such as biological samples and cantilever tips were added in order to fully cover the spectrum. The first set of categories which were chosen is shown in Fig. 4.2.

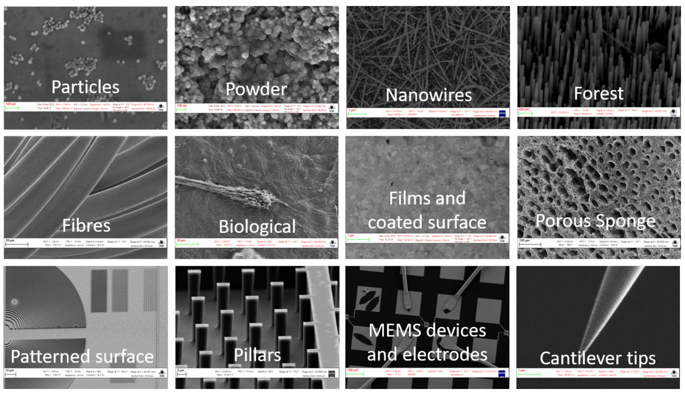


Figure 4.2: The dimensionality of nanoscience objects provided the basis for the categories chosen. Ther categories, such as Biological and Cantilever Tips were added as these were common images found in the SEM images database.

These twelve categories cover a broad range of SEM images, and they are applicable to scientists working in different areas of nanoscience. They were chosen to be as distinct from one another as possible; however, as will be discussed in Sec. 4.2, depending on the magnification chosen, different categories can look very similar to one another, creating challenges for the image recognition.

4.2 Retraining the neural network

Training an Inception-v3 network from scratch is a computationally intensive task and may take several days or even weeks. *Transfer learning* is a technique that dramatically reduces the time by allowing use of a fully-trained model for a set of categories, and to retrain from the existing weights for new classes. In fact, it has been demostrated [28] that lower layers which have been trained to distinguish between some objects can be reused for many recognition tasks without any alteration. In our case, we employed an Inception-v3 model fully trained on 1000 categories in ImageNet [29], a common accademic dataset for training image recognition systems, and retrained the final layer from scratch for SEM images, while leaving all the others untouched. Though transfer learning is not as good as a full training, it is noticeably effective and has the great advantage that it can be run in relatively short times, even without the need of any GPU accelerator. The execution of the code described in Sec. 4.2.1 took roughly 5 minutes (Fig. 4.3) when employing one GPU on the COSILT cluster (Sec. 2.5).

4.2.1 The code

We adapted the `retraining.py` Python script available on the TensorFlow GitHub page [24]. The images should be provided as `jpg` or `png` files located in a directory structure in which each subdirectory corresponds to a unique label for the images that resides within that subdirectory (i.e., all the images of a given category must be in a subfolder named as that category). The script performs the following steps:

- set up the pre-trained Inception graph;
- create a list of all the images and split them into different sets;
- calculate the bottlenecks image summaries and cache them on disk;
- add the new layer to be trained;
- set up all the weights to their initial default value;
- create the operations needed to evaluate the accuracy of the new layer;
- run the training for number of training steps as specified;
- run the final test evaluation;
- write out the trained graph and labels with the weights stored as constants.

Training, Validation and Test set

The provided images are divided into three different sets. The general rule is to put 80% of the images into the main training set, keep 10% aside to run as validation frequently during training to avoid overfitting, and then have a final 10% used as a test set to predict the performance of the classifier.

The `create_image_list()` function uses the filename of the image in the given `image_dir` to determine which set it should belong, according to the `testing_percentage` and `validation_percentage`, and returns a dictionary containing an entry for each label subfolder, with images split into training, validation and testing sets within each label. This is designed to ensure that images do not get moved between sets on different runs, since it could generate overfitting problems if images that had been used for training a model were subsequently used in a validation set.

To decide the appropriate set and to keep existing image file in the same set even if more images are subsequently added, a `hash` of the file is used to generate a probability value:

```
percentage_hash=(int(hash_name_hashed,16)%max_n_images_per_class)*
                 (n_class/max_num_images_per_class)
if percentage_hash<validation_percentage:
   validation_images.append(image_name)
elif percentage_hash<(testing_percentage+validation_percentage):
   testing_images.append(image_name)
```

```
else:
   training_images.append(image_name)
```

Bottlenecks

The first phase of the script analyzes all the provided images and calculates the *bottleneck* values for each of them. Bottleneck is an informal term often used for the layer just before the final output layer that actually does the classification. This penultimate layer has been trained to output a set of values used by the classifier to distinguish between all the categories it has been asked to recognize. It is a meaningful and compact summary of the images, containing enough information for the classifier to make a good choice in a very small set of values. The reason why the final layer retraining can work on new categories is that the kind of information needed to distinguish among the ImageNet categories is also useful to distinguish among new types of features. Calculating each bottleneck takes a significant amount of time. Since every image is reused multiple times during training, bottlenecks values are cached on disk so they do not have to be repeatedly recalculated.

Training

Once the bottlecks are completed, the actual training of the top layer of the network begins. Each step randomly takes 10 images from the training set, finds their bottlenecks from the cache, and feeds them into the final layer to get predictions, which are then compared against the actual labels to update the final layer's weights though the back-propagation process. Each step outputs the training accuracy, the validation accuracy and the cross-entropy (see Sec. 2.1). The final test accuracy is based on the percentage of images in the test set that have been correctly classified after the model has been fully trained.

We explored the performance of the learning process varying the training steps from 2000 to 8000. Doubling the number of steps, doubled the time, as expected, but the final accuracy did not increase significantly, despite the network having been trained for twice as long, because the model converged very rapidly. Thus, we fixed the number of training steps to 4000.

Moreover, we verified how the retrain execution time could be affected by the total number of images in the training sample, as shown in Fig. 4.3. Four repetitions have been performed and the average time has been used. The error bars represent the range of recorded results. It can be seen that for the range studied (450 to 25,500 images) the number of images does not seem to strongly influence the average retrain time.

The script writes out a version of the Inception-v3 network with a final layer, called `final_result`, retrained on SEM images, to a `GraphDef` file called `output_graph.pb` and the labels to a text file called `output_labels.txt`. These two files are the only two elements needed for the image recognition.

4.2.2 Achievements on Retraining

The pre-trained Inception model was retrained using SEM images in the categories shown in Fig 4.2. In order to investigate the retraining, and to understand

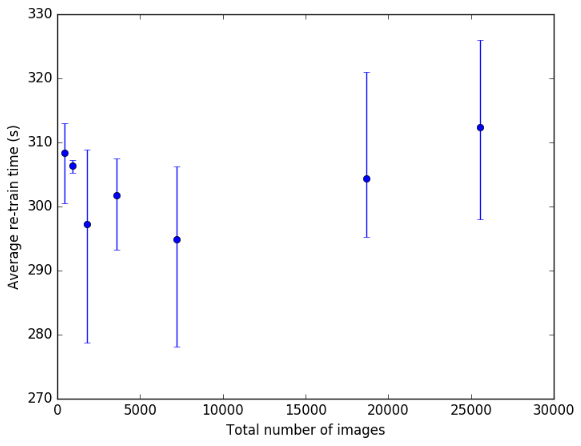


Figure 4.3: The average retrain time as a function of total number of images in the training sample. Four repetitions have been performed and the average time has been used. The error bars represent the range of recorded results.

how the overall accuracy of the neural network could be improved, the retraining was repeated with some categories being merged, and with different numbers of images in each category. In the first retrain in total 18,693 images were used, with an average final train accuracy of 86.2%. The breakdown of the number of images in each category is shown in Table 4.1.

Table 4.1: Initial training set

Category	Number of images
`Particles`	3419
`Fibres`	153
`Biological`	953
`Patterned_surface`	1999
`Pillars`	1364
`Cantilever_Tips`	1561
`Nanowires`	2439
`Powder`	895
`Forest`	1253
`MEMS_devices_electrodes`	4178
`Porous_Sponge`	171
`Films_Coated_Surface`	308

Some SEM images are annotated by scientists with lines, circles and written measurements written directly on the image file. An example is shown in Fig. 4.4. These images were withheld from the initial retrain, however when they were included (increasing the total number of images to 21,106), the final train accuracy remained the same, suggesting that the neural network is able to ignore these features. This may be because these annotations are found in almost all the categories, so they do not become features used by the neural network to identify a specific category.

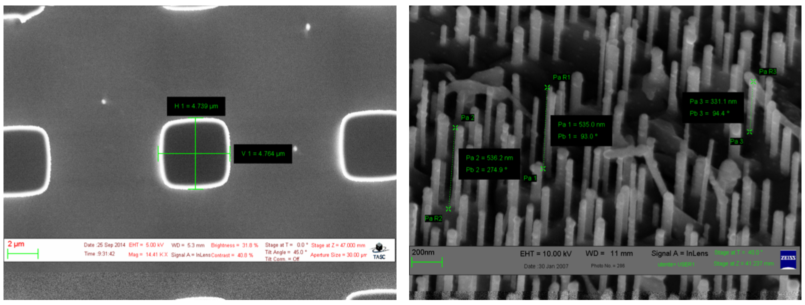


Figure 4.4: Examples of SEM images which are annotated with lines and writing, which are directly written on the image file. The neural network was able to effectively ignore these features when re-trained.

In order to determine the effect of the number of images in each category on the final train accuracy, only the 9 categories which had up to 800 images available were chosen. The results are reported in Fig. 4.5. Four repetitions have been performed and the average accuracy has been used. The error bars represent the range of recorded results. The clear trend is that the the final train accuracy increases as the number of images increases. Whilst the accuracy is only 81.8% with 450 images (50 for each category), this rises to 86.1% where there are 7200 images (800 for each category). However, it is interesting to note that even with just 450 images, the neural network can achieve an accuracy of more than 80%.

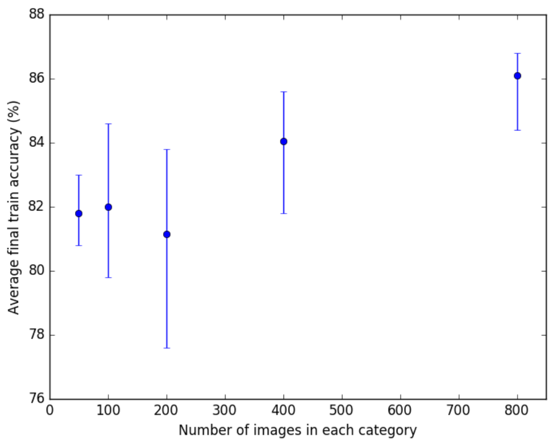


Figure 4.5: Average final train accuracy as a function of the number of images in each category. Four repetitions have been performed and the average accuracy has been used. The error bars represent the range of recorded results.

In order to further increase the accuracy, some categories were combined with one another. The categories `Pillars` and `Patterned_surfaces` share many similarities, as shown in Fig. 4.6. They both have regular arrays of objects, and if pillars are imaged directly from above, rather than from a tilted angle, they look almost indistinguishable from some patterned surfaces. When the neural network was retrained with these categories combined, the final accuracy increased to 89.5%, compared to 86.2% when they were separate.

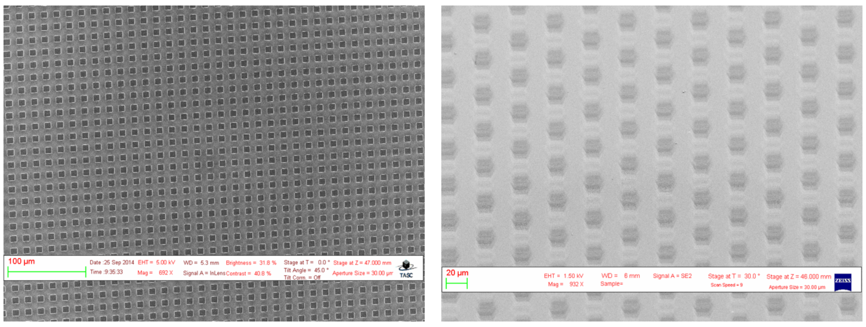


Figure 4.6: Example of how a top down view of `Pillars` (left) looks almost indistinguishable from some `Patterned_surface` (right).

To verify that this improvement in accuracy was due to the fact that these categories are similar to one another, and not simply because there was one less category, two dissimilar categories (`Pillars` and `Biological`) were combined. The final accuracy dropped to 85.7%, which is lower than when they were separated. This confirmed that combining categories which are very similar to one another can help to significantly increase the retrain accuracy, at the expense of broadened categories which offer lower distinction. Following this idea, we also combined `Forest` and `Nanowires`, resulting in a final set of 10 categories, with an average final accuracy of 90.1%. Further merging of categories, such as by combining `MEMS_devices_electrodes` within `Patterned_Surfaces` actually led to a decrease in the average final train accuracy.

The final categories and the number of images in each of them is shown in Table 4.2.

Table 4.2: Final training set

Category	Number of images
`Particles`	3419
`Fibres`	153
`Biological`	953
`Patterned_surface`	3363
`Cantilever_Tips`	1561
`Nanowires`	3692
`Powder`	895
`MEMS_devices_electrodes`	4178
`Porous_Sponge`	171
`Films_Coated_Surface`	308

4.3 Image recognition

When retraining the neural network, a portion of the images are automatically kept aside for validation and testing. The final train accuracy gives an overall score for the "average" performance, however it does not shed any light on which types of image the neural network is able to categorise consistently well, and which it finds more challenging.

In order to manually test the performance of the neural network, categories were split into sub categories which represented the types of image found in each category. For example, the main category `Nanowires` has images which are:

- a mesh of entangled nanowires
- a few nanowires
- a focused image of an individual nanowire
- parallel aligned forests
- forests where the top of the forest (the crust) is also seen

A summary of the results is presented in Sec. 4.3.2.

Some of these subcategories may look more similar to images in other categories, and so the accuracy of the neural network in recognising images could be different for each subcategory. For each category, all the subcategories have been tested by running the image recognition script on a representative sample of images. Results are presented in Sec. 4.3.3.

There were a number of images which had elements of two categories within the single image. These were separated from the train sample, and investigated by passing the images through the image recognition of the trained network. We present the results in Sec. 4.3.4.

4.3.1 The code

We adapted the `classify_image.py` Python script available on the TensorFlow GitHub page [24]. The code creates a graph from the GraphDef protocol buffer we obtained from the retraining (see Sec. 4.5):

```
with tf.gfile.FastGFile(os.path.join(model_dir,
        'output_graph.pb'), 'rb') as f:
   graph_def = tf.GraphDef()
   graph_def.ParseFromString(f.read())
   _ = tf.import_graph_def(graph_def, name='')
```

Then it runs inference on an input `jpg` or `png` image using the last layer of the retrained network, namely `final_result`, and the list of labels in `output_labels.txt` (see Sec. 4.5):

```
with tf.Session() as sess:
   softmax_tensor = sess.graph.get_tensor_by_name('final_result:0')
   predictions = sess.run(softmax_tensor,
            'DecodeJpeg/contents:0':image_data)
   predictions = np.squeeze(predictions)
   fname = os.path.join(model_dir, 'output_labels.txt')
```

```
with open(fname) as f:
   content = f.readlines()
top_k = predictions.argsort()[-num_top_predictions:][::-1]
```

Finally, it outputs human readable strings of the top 5 predictions along with their probabilities, as metadata ready to be added to the image file, once converted to HDF5 format.

4.3.2 Summary of Image Recognition

All of the results from the image recognition of the test images was collated. The top ranked score was split into three groups:

- Top ranked score is for the correct category
- Top ranked score is for an incorrect category, however the second ranked score is for the correct category
- Both of the top two categories are incorrect

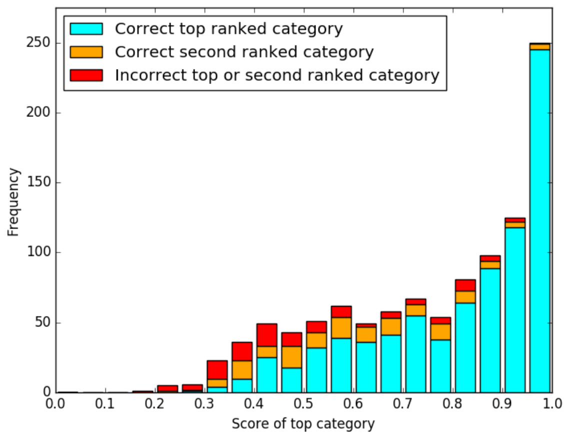


Figure 4.7: A histogram of the score of the top ranked category. Either the top ranked category was correct (cyan), or it was incorrect with the second category being correct (orange), or incorrect with neither of the top two categories being correct (red).

A histogram of all of the test results is shown in Fig. 4.7. It can be noted that when the score of the top ranked category is high (e.g. above 0.8), there is a high chance that the top ranked category will be the correct category of the image. However, when the score is low (e.g. below 0.5), it is more likely that the top ranked category is not the correct one. This matches with our understanding of the neural network. If the trained neural network is confident that an image belongs to a certain category, it will assign that category a high score. A low score for the first category suggests that the neural network has struggled to identify the image.

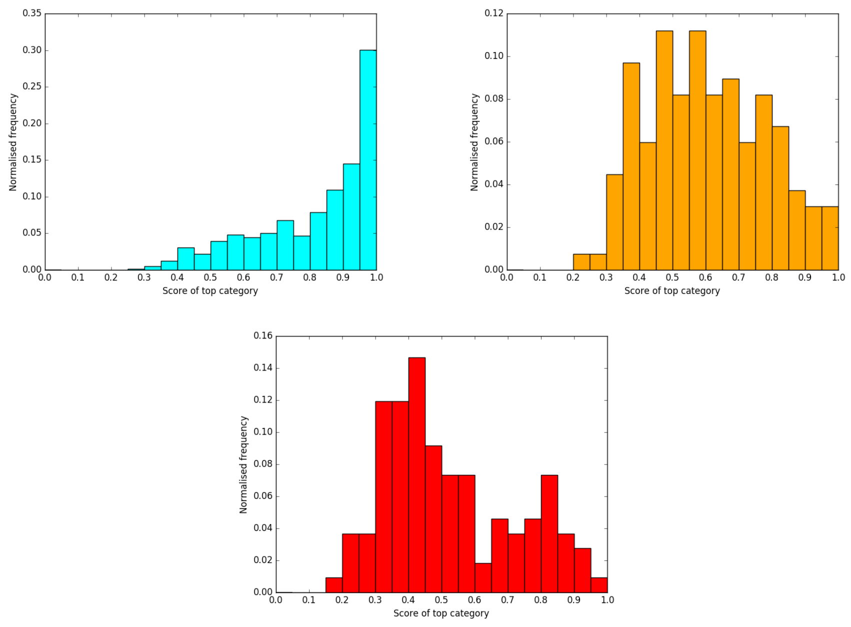


Figure 4.8: The normalised distribution of scores for each of the groups. Cyan: Top category score when the top category was correct. Orange: Top category score when the second ranked category was correct. Red: Top category score when neither of the top two categories was correct.

The distribution of normalised frequencies in each group is shown in Fig. 4.8. It can be seen that those top ranked scores which were for the correct category (cyan bars) mostly occur at very high scores, with almost a third being in the range of scores 0.95-1. In contrast, the distribution for those top scores which were incorrect, but the second score was correct (orange bars) has a peak at around 0.5-0.6. In this case the neural network is often conflicted between two categories because the image may be an edge case and similar to trained images found in two categories (see also Sec. 4.3.4). Finally, the normalised frequency distribution of top ranked scores, when neither of the top two scores were correct (red bars) shows a peak at around 0.4. As mentioned before, when the top ranked category has such as low score, it suggests that the network is unable to categorise the image.

The relative frequencies of the top ranked scores was used to generate plots of the probability that a given top ranked score is for the correct category. The left plot in Fig. 4.9 shows that the probability that a top ranked score is for a correct category increases for higher scores. The crossover of lines indicates the point at which the two groups are equally likely (i.e. probability 0.5). When the top ranked category has a score of 0.5, there is a 50% chance that the top ranked category is correct. The right plot in Fig. 4.9 shows that when we consider the top or second ranked category being correct, the crossover point occurs at a lower score of 0.35.

This data can be used to help design the interaction of the neural network with users. It shows that even if the top ranked score is low (in the region of

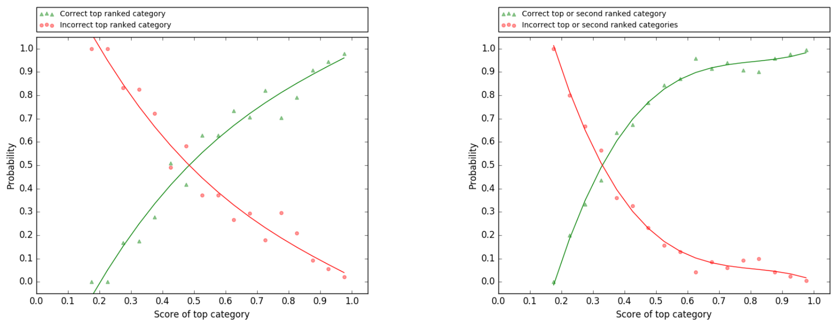


Figure 4.9: For a given top ranked score for an image, probability that it is the correct category (left) and that either it is the correct category or the second category is correct (right). In each case the inverse is also plotted, with the cross over point showing the score at which the two scenarios are equally likely.

0.35) the top two ranked categories could be suggested to the user, whilst for very high scores ($\gtrsim 0.9$) it may be sufficient to automatically assign the top ranked category with a high confidence of it being the correct one. Moreover, when the top ranked category is not correct, in 55% of cases the second top ranked category is correct.

4.3.3 Image recognition of subcategories

The neural network with ten categories and a final train accuracy of 90.1% was used for testing. For each image, the output of the image recognition is that each category is given a probability score of being the category to which the image belongs. If the neural network is confident that an image belongs to a category, it will assign that category a score close to 1. Images which are more difficult for the neural network to identify will have many categories with approximately similar scores. For example, one image of an electrode scored the following:

```
MEMS_devices_and_electrodes: 0.88 (top ranked category)
Patterned_surface: 0.035 (second ranked category)
Biological: 0.028
Cantilever_tips: 0.016
Nanowires: 0.011
```

In the following, for some representative test folders, the highest scoring categories are shown. An explanation is provided for why an “incorrect” category has scored highly for a set of test images which should all belong to another category.

`Nanowires`

The “crust” of forests bares a strong resemblance to neurons in biological images, as shown in Fig. 4.10. This explains why 12% of the images in the crust sub folder had `Biological` as their top category, instead of the expected result of `Nanowires`.

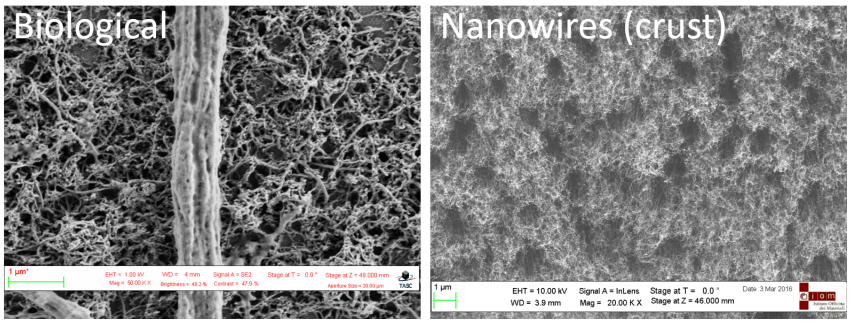


Figure 4.10: Comparison showing the resemblance between the crust of forest (a subcategory of `Nanowires`) and `Biological`, explaining why 12% of the times the neural network failed in identifying the correct category.

`Powder`

The boundary between particles, powders, and certain types of films and coated surface could be down just to the magnification used, as shown by the results in Fig. 4.11. Whilst a zoomed out image of a powdery material is correctly identified 80% of the time, when an image at a higher magnification is taken, at which point individual circular particles which form the powder can be seen, the top ranked category changes to `particles`. There is very little that can be done to correct this, because the neural network is correctly judging what it is seeing in the image, whilst the human observer has the benefit of seeing the image in the context of the previous images, and thus s/he knows that this is in fact a zoomed in image of a powder.

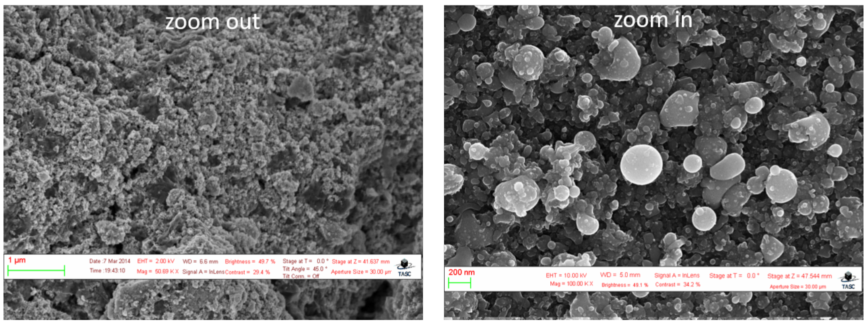


Figure 4.11: Comparison between zoomed in and zoomed out `powder`, explaining why 20% of the times the neural network failed in identifying the correct category.

4.3.4 Images with two categories

Images which had elements of two categories within the single image were investigated as a separate case. As expected, for these images the neural network was able to identify both categories within the images, with often the more dominant category receiving a high score, as the following examples demonstrate.

`Nanowires` on `MEMS_electrodes_and_devices`

Files in this test folder comprised images of electrodes and MEMS structures which had nanowires dispersed over them. The dominant feature in the image is the electrode like structure, since the nanowires were mostly dispersed and isolated from one another. These observations are reflected in the scores, with `MEMS_devices_and_electrodes` being the top ranked category in 69.2% of cases, whilst for 23% of images `Nanowires` were identified as the top ranked category.

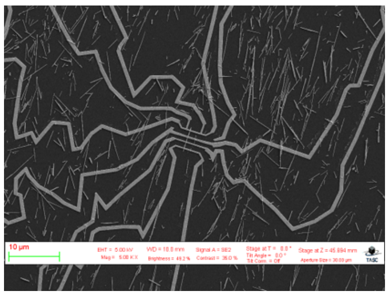


Figure 4.12: A representative image from a test folder of SEMs which are MEMS electrodes covered with nanowires.

`Nanowires` on `Cantilever_tips`

This test folder contained images of nanowires which were present at the very end of tips. In this case the triangular shape of the tip is evident, as well as the web of nanowires which are present on the surface. The neural network identified both of these categories, with 56% of images being classified as `Nanowires`, and 28% of images classified as `Cantilever_tips`.

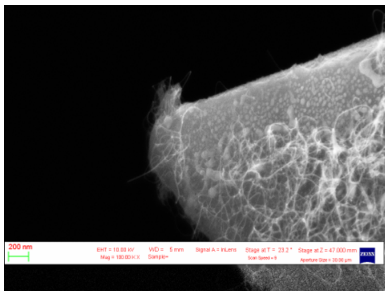


Figure 4.13: A representative image from a test folder of SEMs which are of cantilever tips covered by nanowires.

`Particles` on `Nanowires`

These are images of nanowires which have small particles on their surface along their length. If these images had to be classified into one category, they would be classed as nanowires, since the nanowire is the dominant feature, and the particles are decorating their surface. This is represented by the behaviour of the neural network, which selected `Nanowires` as the correct category in over 85% of cases, and `Particles` as the correct category in $\sim 9\%$ of cases.

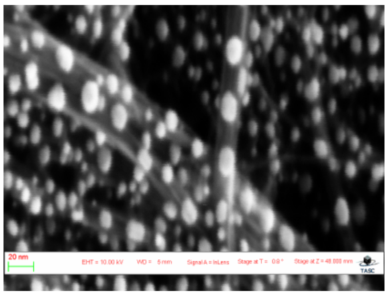


Figure 4.14: A representative image from a test folder of SEMs which are of nanowires decorated with particles along their length.

Chapter 5

Technical Results

This Chapter reports the work and the results obtained in setting up the computational infrastructure to perform a massive image processing. As a matter of fact the image classification through TensorFlow used to obtain the results presented in Sec. 4.3 should be run on hundreds of thousand of images. Our aim is to survey the available computational tools and storage services allowing us to process the huge amount of data provided by the SEM instrument.

We therefore ported our Python code on an Apache Spark cluster implementation (Sec. 5.1) and we studied the performance behaviour of the whole program on large image samples obtained in two different environments, the first one created on an OpenStack infrastructure (Sec. 5.2), and the latter created by means of Docker infrastructure on a real bare metal platform on a COSILT cluster node (Sec. 5.3). Due to the fact that the Spark cluster is embedded within a real HPC cluster we were also able, in this latter case, to evaluate different file systems as storage for all the images to be processed. On COSILT cluster three different options are available: we will first present individually the results obtained with the shared NFS (Sec. 5.3.1), the parallel Lustre file system (Sec. 5.3.2), and the local file system to the node itself (Sec. 5.3.3); finally, a comparison is presented in Sec. 5.3.4.

5.1 The Code

In this section we briefly discuss the strategy we adopted for porting and parallelising the serial code described in Sec. 4.3.1 on a Spark (See Chapter 3) cluster.

We remark that the classification part is only one section of the whole processing, which also includes the conversion of the image from `tif` to `jpg` format and the conversion from `tif` to a `HDF5` file in which the top 5 label outputs of the image recognition are added as new metadata, along with their rates. We therefore included these two additional steps in the code we discuss below, and we instrumented our parallel version for all these functions. We will report and comment the different times and roles of the three sections mentioned above for all the cases we studied on the bare metal infrastructure.

The final code is still written as a Python script that will be launched within the Spark cluster with some scheduling mechanism. In this work we used the

Standalone mechanism through the `bin/spark-submit` script, which includes the Spark dependencies for us in Python, and sets up the environment for the Spark's Python API to work properly.

The first thing the Spark program does is to create a `SparkContext` object, which tells Spark how to access the cluster:

```
from pyspark import SparkContext
sc = SparkContext()
```

After this initialization, all the Spark methods to create and manipulate RDDs can be used. We created the first RDD by parallelizing an existing collection of image paths. Unless explicitly indicated, Spark automatically splits the RDD into multiple partitions, which may be computed on different nodes of the cluster. To take over control of this aspect, we manually forced the distribution of the collection by setting the optional `numSlices` parameter:

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

where the `tif_list_batched` has been obtained by splitting the whole list according to the number of cores and managing the rest:

```
def_split_list(image_list, n_core):
   dim_batch = [len(image_list) // n_core for _ in range(n_core)]
   for i in range(len(image_list)%n_core):
      dim_batch[i] = dim_batch[i]+1
   splitted_list = []
   k=0
   for d in dim_batch:
      splitted_list.append(image_list[k:k+d])
      k = k + d
   return splitted_list
```

To obtain the new RDDs, we called either the `map` or the `flatMap` transformations, which apply the given function on each element of the RDD and return a RDD of the contents of the iterator returned. For example:

```
jpg_RDD = tif_RDD.map(tif2jpg_batch)
```

Due to the lazy evaluation of Spark (see Sec. 3.2.2), to obtain the final values we had to perform a `collect()` action:

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

It is worth noting that each time we call a new action, the entire RDD must be computed from scratch. To avoid this inefficiency, we explicitly asked Spark to `persist()` the data for the RDD we used more than once, saving them in the cache.

The TensorFlow graph reads the GraphDeph and the list of labels, which are outputs of the neural network training (see Sec. 4.3.1). These two variables must be available on each node, so the obvious option was to broadcast them:

```
model_data_bc = sc.broadcast(model_data)
content_bc = sc.broadcast(content)
```

We note that broadcast variables keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. Spark attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

5.2 Openstack

As described in Sec. 3.3.2, we built a 16 core Spark cluster, where the images to be processed were locally stored. In the first version of the parallel code, the images had been previously converted to the `jpg` format, so they were ready to be processed in parallel by the TensorFlow image recognition, using a `map` approach. In the second version, we took images in the original `tif` format generated by the SEM, and included the `tif2jpg` conversion using the same parallel approach. In the third version, we finally improved our initial implementation, by reading the input pipeline at the beginning of the TensorFlow graph and thus noticeably reducing the execution time.

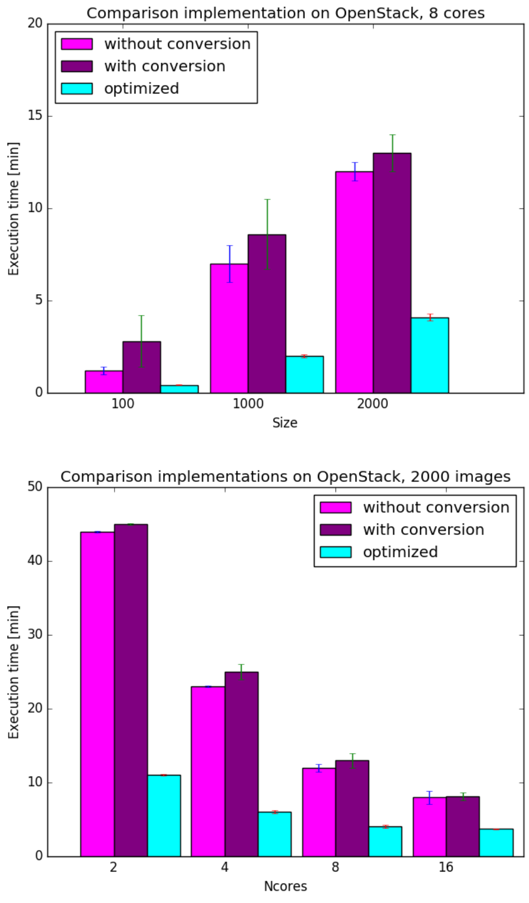


Figure 5.1: Comparison of the strong (upper panel) and weak (lower panel) scaling of different implementations on Openstack, respectively as a function of the number of cores using a sample of 2000 images, and as a function of the sample size using 8 cores.

The performance of the different implementations have been compared in Fig. 5.1, where the execution time is shown as a function of the sample size at a fixed number of cores (top panel), and as a function of the core number for a fixed sample size (bottom panel). The plotted values are the average over five runs, with the error bars representing the standard deviation of the measures. It is worth noticing the large fluctuations among the five runs that at a first glance seem quite surprising: they reach up to 40%-50% in some cases.

After a careful analysis, we discovered that the task completed correctly despite the fact that many jobs failed on some nodes of the Spark cluster. The overall fault tolerant Spark architecture was stable enough to cope with the failing jobs in a completely transparent way: the lost RDD partitions were recomputed from the original dataset by the other worker nodes. However, it is clear that execution times allocated to those jobs are not meaningful and reliable in those cases. Many repetitions have been necessary to obtain the complete overview shown in Fig. 5.1.

In the Figure, the comparison between the magenta and the purple bars in both the panels gives an estimate of the time needed to execute the `tif2jpg` conversion, which is a relatively small fraction of the total execution time for a minimum sample size of roughly 1000 images. We will further investigate this aspect, and will offer a more quantitative estimate of the relative durations of the individual functions in Sec. 5.3. The cyan bars in both the panels show the optimized version of the code represented by the purple bars. Given the remarkable improvement obtained, from now on we will always refer only to the optimized version. However, it must be noted that even with the optimized version the time needed to process the images seems to be longer than expected.

Thus, our results suggest that the virtual environment is neither stable nor performing well enough for our purposes. This is likely due to the limited amount of resources available on the OpenStack infrastructure we used.

5.3 The computational node on COSILT

In this section we present and discuss results for the code which encompasses all the three functions on each parallel batch of images: the `tif2jpg` conversion, the TensorFlow image recognition, and the `tif2hdf5` conversion with the metadata writing.

We run this code on the computational node of COSILT described in 3.3.3, measuring the time of each of the three functions.

It is also worth studying the role of multithreading and how it could affect the performance, due to the fact that TensorFlow is natively multithreaded. In general, multithreaded code will detect the number of cores available on a machine and make use of all of them. The number of threads available to the TensorFlow process can however be explicitly configured by passing a `tf.ConfigProto` argument when constructing the `tf.Session` in which the image recognition takes place:

```
tf.Session(config=tf.ConfigProto(intra_op_parallelism_threads=1))
```

In this way, we were able to compare the behaviour of both the single thread and the multithread implementations.

Moreover, as already mentioned, we investigated with care the impact of the file system on the overall performance by running the code on data located in the NFS (Sec. 5.3.1), in the Lustre file system (Sec. 5.3.2) and in the local file system (Sec. 5.3.3).

5.3.1 Network File System

The NFS allows a system to share directories and files with others over a network: in case of COSILT cluster, the NFS is mounted over infiniband high speed network; the NFS server provides a RAID5 disk.

The execution times for both single thread and multithread algorithms are compared in Fig. 5.2, where different panels are shown for different sample sizes. Scalability for both the algorithms is rather good on a relatively small number of cores ($N \lesssim 8$), but with a further increase of the core number it seems that no real gain is obtained.

We note that the executors are distributed by Spark in a balanced way: if 8 cores are requested for a job, and two workers are available, 4 cores will be allocated to each worker. Our cluster consists of 2 workers with 12 cores each. Thus, a single process running on a worker has 12 physical threads available. When using 2 cores, 12 threads are launched on each worker, when using 4 cores, 24 threads are launched on each worker, and so on.

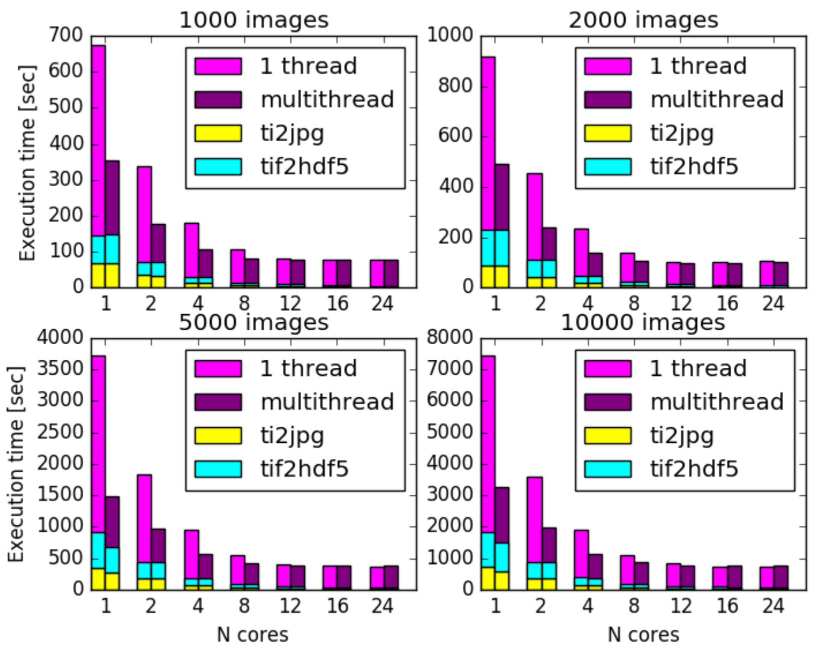


Figure 5.2: Execution time of single thread (magenta) and multithread (purple) TensorFlow image recognition, `tif2jpg` conversion (yellow) and `tif2hdf5` (cyan) as a function of the number of cores on the COSILT node, using different sample sizes. Data images are located in the NFS.

In the range in which scalability is good, multithreading is of course much more efficient than single thread algorithm and the codes run faster. On the other hand, comparing the plots in Fig. 5.3 we can note that the speedup of the single thread algorithm is higher than the multithread one's. This is due in part to the higher baseline for the single thread algorithm (the execution time is nearly double the execution time for the multithread one), and in part to the mutual interference of multiple threads when the number of cores is $N \gtrsim 4$.

Thus, it is not surprising that for a larger number of Spark executors the multithread algorithm is not working optimally, due to the limited number of physical cores: i.e., threads are mutually interfering negatively due to the lack of physical resources.

In Fig. 5.2 the execution times of `tif2jpg` conversion (yellow) and `tif2hdf5` (cyan) are also plotted, in addition to the time of the TensorFlow image recognition. As we have estimated in Sec. 5.2, it is now evident that the execution time is dominated by the TensorFlow image recognition function. As expected, the `tif2jpg` and the `tif2hdf5` conversions take approximatively the same amount of time, being two equivalent functions. Moreover, their scalability is good, confirming the image recognition to be the non-optimally scaling function, especially in the multithread case.

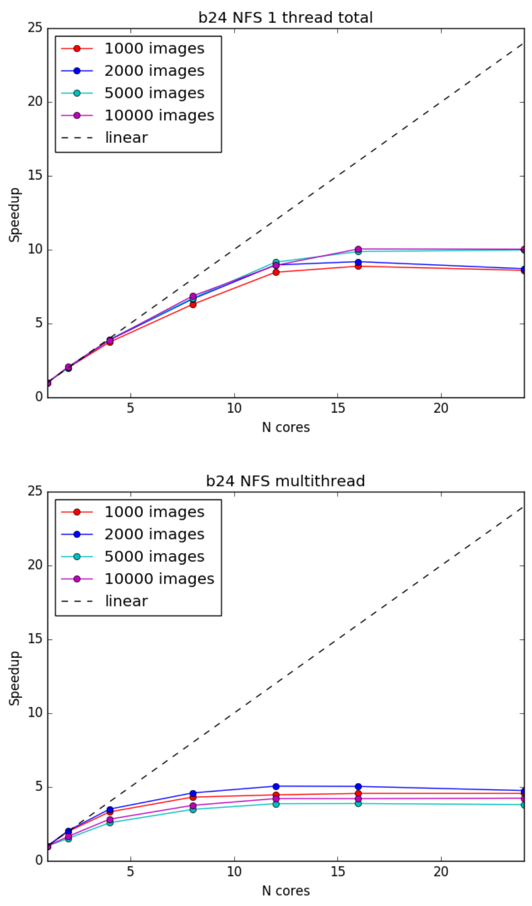


Figure 5.3: Speedup of single thread and multithread implementations as a function of the number of cores on the COSILT node, using different sample sizes. Data images are located in the NFS. Errorbars, of the order of 10%, are not reported to avoid confusion.

5.3.2 Lustre file system

The Lustre File System is a high-performance distributed file system, providing high I/O throughput in clusters and shared-data environments, and also independence from the location of data on the physical storage. On the COSILT infrastructure it spawns two I/O servers with two OSTs each. The network connectivity is provided by Infiniband.

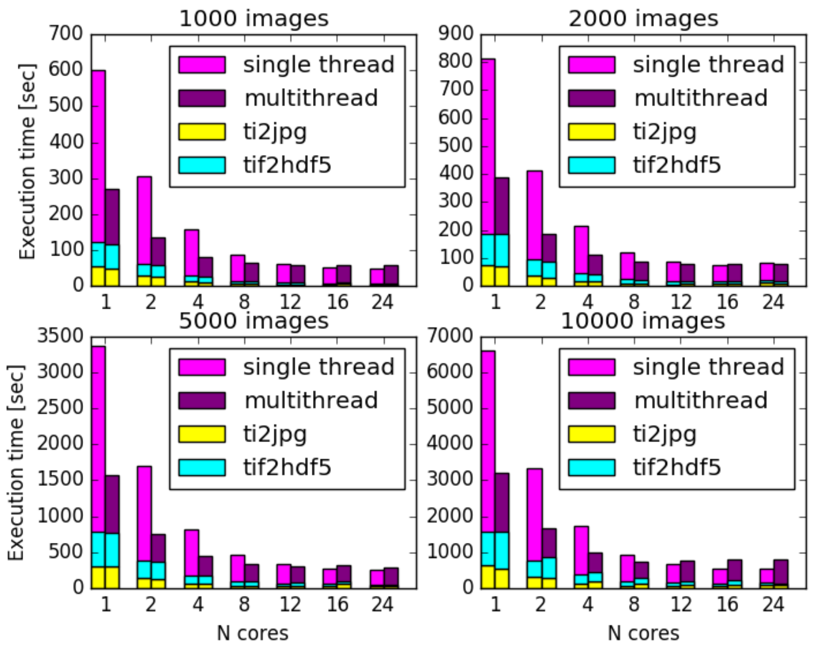
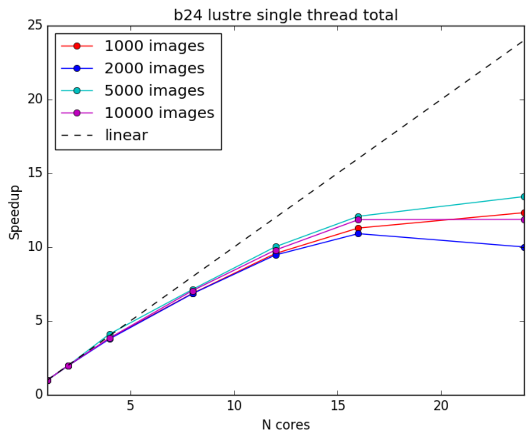


Figure 5.4: Execution time of single thread (magenta) and multithread (purple) TensorFlow image recognition, `tif2jpg` conversion (yellow) and `tif2hdf5` (cyan) as a function of the number of cores on the COSILT node, using different sample sizes. Data images are located in the Lustre file system.

The execution times for both single thread (magenta) and multithread (purple) TensorFlow image recognition, `tif2jpg` conversion (yellow) and `tif2hdf5` (cyan) functions are compared in Fig. 5.4, where different panels are shown for different sample sizes. The scalability for both the algorithms is slightly better than for the NFS case, in particular for a large number of images ($\sim 10,000$). This is even more evident in Fig. 5.5, where the speedup of single thread (top panel) and multithread (bottom panel) algorithms is shown as a function of the number of cores for different sizes.

The same considerations concerning the multithreading made in Sec. 5.3.1 apply here. It can be noted how the execution times of `tif2jpg` and the `tif2hdf5` conversions are fluctuating, especially for a large number of cores ($N \gtrsim 16$). This is not a serious problem, which can be explained with the statistical fluctuations of the runs, of the order of 20%.



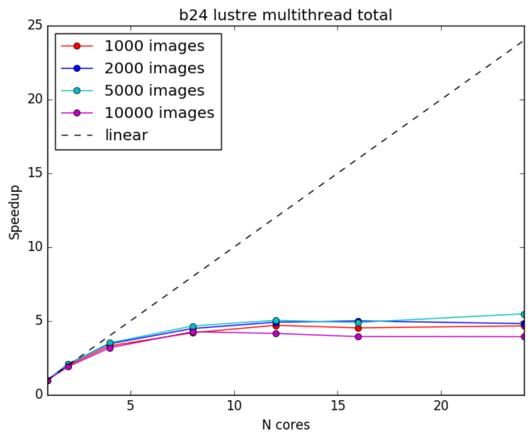


Figure 5.5: Speedup of single thread and multithread implementations as a function of the number of cores on the COSILT node, using different sample sizes. Data images are located in the Lustre filesystem. Errorbars, of the order of 20%, are not reported to avoid confusion.

5.3.3 Local file system

Finally, we explored the performance of the parallel implementation when employing the local file system to the node on the COSILT cluster. Such a storage area is provided by a single SATA disk of about 1 TB size.

Fig. 5.4 shows the execution times for both single thread (magenta) and multithread (purple) TensorFlow image recognition, `tif2jpg` conversion (yellow) and `tif2hdf5` (cyan) functions, where different panels are for different sample sizes. The scalability for both the algorithms is the worst among the three file systems, in particular for a relatively small number of images ($\lesssim 5,000$). This is even more evident in Fig. 5.5, where the speedup of single thread (top panel) and multithread (bottom panel) algorithms is shown as a function of the number of cores for different sizes. Fluctuations in the data points are considerable, and may be partially explained by statistical errors, which in this case amount to 25-30%.

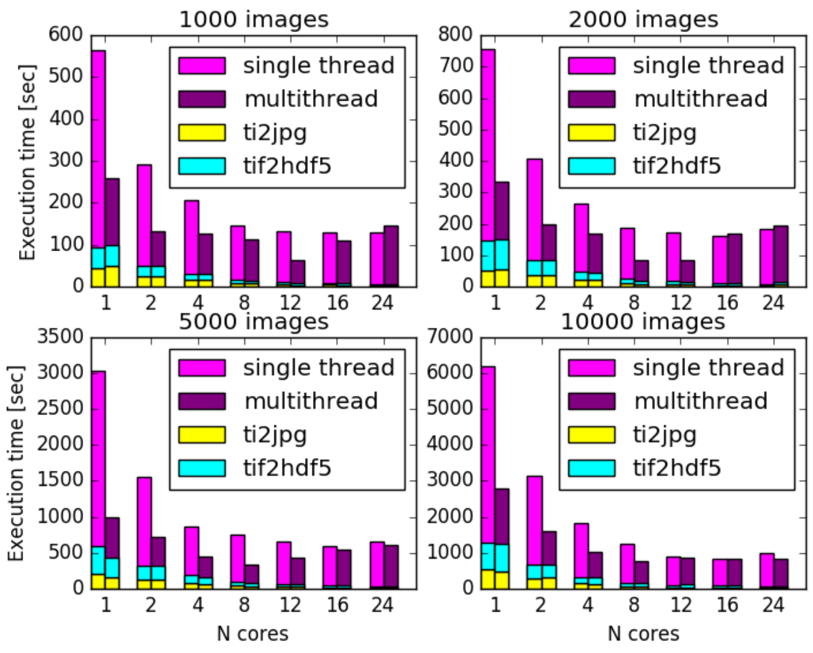


Figure 5.6: Execution time of single thread (magenta) and multithread (purple) TensorFlow image recognition, `tif2jpg` conversion (yellow) and `tif2hdf5` (cyan) as a function of the number of cores on the COSILT node, using different sample sizes. Data images are located in the local files ystem.

The execution time of the multithread algorithm increases when more than 8 cores are used. This is partially due to the mutual interference of multiple threads as the number of involved processes increases. However, this clearly indicates that the local file system is not able to manage the I/O overhead in an optimal way.

On the other hand, the scaling of `tif2jpg` and the `tif2hdf5` parts is good with respect to the TensorFlow image recognition for both the multithread and the single thread implementations, which seems to represent the bottleneck in the process. Further and more dedicated investigations are needed to fully understand this issue.

5.3.4 Comparison of different file systems

The behaviour of our application is generally similar for all the file systems, i.e. limited scalability, especially for the multithreaded version, but some pattern performance related to the different file systems clearly emerge. In this section, we will briefly compare the impact on performance of the studied file systems for some significant cases, highlighting some differences. The top panel of Fig. 5.8 shows the total execution time of the multithread algorithm running on 2 cores, when all the physical threads are still employed in an efficient way, for the three file systems. Computational times are similar, and we observe that the local file system seems to perform slightly better than the Lustre file system: this should

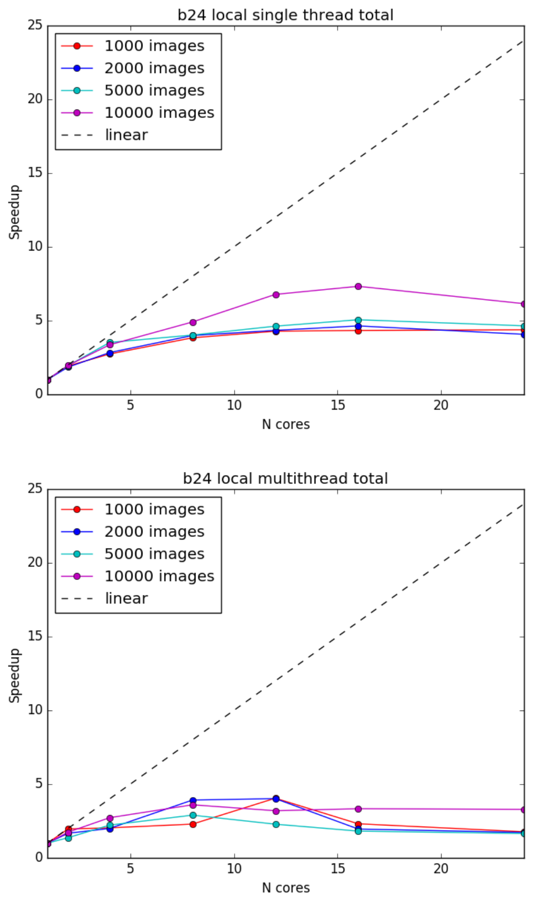
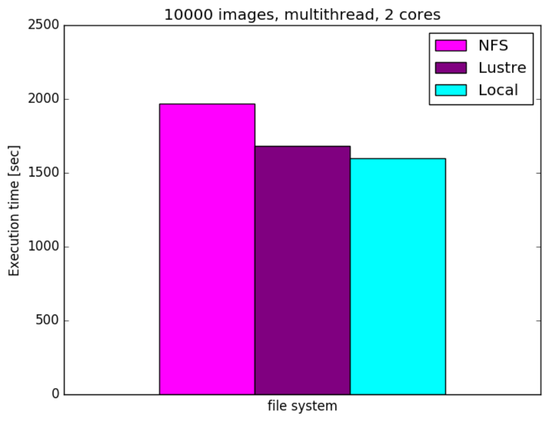


Figure 5.7: Speedup of single thread and multithread implementations as a function of the number of cores on the COSILT node, using different sample sizes. Data images are located in the local filesystem. Errorbars, of the order of 25-30%, are not reported to avoid confusion.

be due to the favourable data proximity.

The top panel of Fig. 5.8 shows the total execution time of the single thread algorithm when all the available 24 cores are used, for the three file systems. Here the situation is quite different with respect to the previous case: the local file system here gives the worst performance, while Lustre file system is performing better. Our explanation is that 24 independent processes are actually overwhelming the capacity on the single local disk, while the Lustre FS thanks to its characteristic high I/O throughput is able to deliver much better performance. At first sight, NFS is delivering relatively good performance with respect to the local disk: our guess here is that the combined action of RAID and the infiniband network is able to reduce the congestion of the I/O operations on the same storage area.



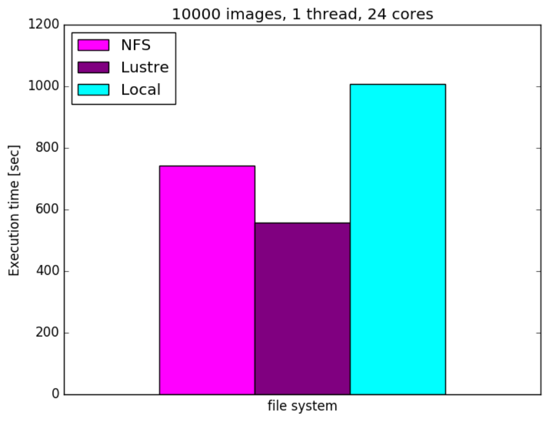


Figure 5.8: File system comparison with a fixed size of 10000 images. Top panel: multithread execution time comparison at 4 cores. Bottom panel: single thread execution time comparison at 24 cores.

Chapter 6

Conclusions

In this thesis, we explored different methods and tools to implement a semi-automatic classification of images coming from the SEM instrument, with the final aim of providing a searchable database to the nanoscience community.

In the first part of our work, we retrained the last layer of a pre-trained neural network based on the Inception-v3 architecture using supervised learning, and employed the TensorFlow library to perform image recognition on SEM images.

The results of the image recognition show that the neural network has a high accuracy at detecting distinct and representative images of each category, however certain subcategories of different objects can look very similar to one another. This effect is primarily due to the nature of the SEM which allows magnification over a very wide range (250-500,000 times). In fact, a zoomed out image of a structure can look completely different to its surface topology when zoomed in. Therefore, while the zoomed out image may be easily classified, the zoomed in image could be confused with another category.

The summary of the test results shows that when the score for the top ranked category is high, there is a high probability that the net has identified the correct category. However, when the score of the top ranked category is low, it becomes more likely that either the second category is correct, or both of the top two categories are incorrect. This information can be used in deciding whether to automatically assign labels to SEM images taken by scientists, or, based on the score of the top ranked category, to suggest the most likely categories to the user to choose from.

As a matter of fact, the image classification through TensorFlow used to obtain the results presented above should be run on hundreds of thousand of images. Thus, in the second part of our work, we surveyed the available computational tools and storage services allowing us to perform a massive data processing.

Our results shows that the Spark architecture is optimally suited for our purposes, and has many advantages: it is simple to use, efficient, flexible, and fault tolerant.

We compared the implementation on two different environments: the first one created on an OpenStack infrastructure, and the second created by means of Docker infrastructure on a bare metal platform on a real HPC cluster node. In this latter case, we were also able to evaluate different file systems as storage for all the images to be processed: a shared NFS, a parallel Lustre file system, and the local file system to the node itself.

Our results suggest that the virtual environment is neither stable nor performing well enough for our purposes. This is likely due to the limited amount of resources available on the OpenStack infrastructure we used. As for the HPC cluster node, the scalabilty of our application is generally limited for all the files systems, especially for the multithreaded version. However, some differences arise: when few cores are allocated and all the physical threads are employed in an efficient way, the local file system seems to perform slightly better than the others, probably due to the favourable data proximity. On the other hand, when all the available resources of the cluster are employed the Lustre file system is performing better than the others, thanks to its characteristic high I/O throughput.

Further and more dedicated investigations are needed to obtain a detailed and focused analysis. With a complete view of all these aspects, we are then planning to employ our parallel image recognition engine, eventually using GPUs, to classify the remaining 125,000 SEM images in a semi-automatic way. We will also be able to provide the estimate of the time needed to process such amount of data.

Once classified and labelled, the images will be used to further enhance the accuracy of the neural network training. We will also explore the improvement that can be reached by training more layers in addition to the final one.

This work will be integrated into the NFFA-IDRP workflow, which will envisage the ingestion of data from SEM in real time. For this purpose, it will be crucial to employ the Spark stream.

The development of the complete workflow will be exported to the NFFA partners, since 10 SEMs belongs to the collaboration. Thus, our work will be very innovative and useful for the nanoscience community.

Bibliography

[1] M. D. Wilkinson et al., *The FAIR Guiding Principles for scientific data management and stewarship*, Scientific Data, 3:160018, 2016

[2] www.nffa.eu

[3] www.rd-alliance.org

[4] *COMMISSION RECOMMENDATION of 17.7.2012 on access to and preservation of scientific information*, http://ec.europa.eu/research/science-society/document_library/pdf_06/recommendation -access-and-preservation-scientific- information_en.pdf

[5] www.iom.cnr.it

[6] www.eudat.eu

[7] http://pan-data.eu

[8] kitdatamanager.net

[9] www.elastic.co/products/elasticsearch

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting*, JMLR, 2014

[11] https://support.hdfgroup.org/HDF5

[12] lustre.org

[13] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, *Rethinking the Inception Architecture for Computer Vision*, CoRR, 1512.00567, 2016

[14] www.tensorflow.org

[15] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, Book in preparation for MIT Press, 2016, www.deeplearningbook.org

[16] T. M. Mitchell, *Machine Learning*, McGraw-Hill, 1997

[17] A. Krizhevsky, I. Sutskever, G. Hinton, *ImageNet classication with deep convolutional neural networks*, NIPS, 2012

[18] S. Ioffe & C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, CoRR, 1502.03167, 2015

[19] K. Jarrett, K. Kavukcuoglu, M. Ranzato, Y. LeCun, *What is the best multi-stage architecture for object recognition?*, ICCV, 2009

[20] V. Nair & G. Hinton, *Rectied linear units improve restricted Boltzmann machines*, ICML, 2010 .

[21] X. Glorot, A. Bordes, Y. Bengio, *Deep sparse rectier neural networks*, AISTATS, 2011

[22] Y. LeCun, *Generalization and network design strategies*, Technical Report CRG-TR-89-4, University of Toronto, 1989

[23] Y. Zhou & R. Chellappa, *Computation of optical ow using a neural network*, IEEE, 1988

[24] https://github.com/tensorflow/tensorflow

[25] https://github.com/tensorflow/models/tree/master/slim

[26] K. He, X. Zhang, S. Ren, J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, arXiv:1502.01852, 2015

[27] A. Canziani, E. Culurciello, A. Paszke, *An Analysis of Deep Neural Network Models for Practical Applications*, arXiv:1605.07678, 2016

[28] J. Donahue, y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, T. Darrell, *DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition*, DBLP journals, arXiv:1310.1531, 2013

[29] www.image-net.org

[30] www.exact-lab.it

[31] www.nvidia.it

[32] www.docker.com

[33] aws.amazon.com/documentation/machine-learning

[34] spark.apache.org

[35] hadoop.apache.org

[36] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, *Learning Spark*, O'Reilly, 2015

[37] J. Dean & S. Ghemawat, *MapReduce: Simplified dta processing on large clusters*, Google Inc., OSDI, 2004

[38] C. Henderson, *Scalability with MapReduce*, published online, craighenderson.co.uk/papers/software_scalability_mapreduce/, 2010

[39] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, *Dryad: Distributed Data-parallel Programs from Sequential Building Blocks*, Proceedings of the 2007 Eurosys Conference, 2007

[40] www.databricks.com

[41] docs.openstack.org

[42] spark.apache.org/downloads.html