Master in High Performance Computing

Deep Learning for Nanoscience Scanning Electron Microscope Image Recognition

Supervisor:
Stefano Cozzini

Candidate:
Cristiano De Nobili

3rd edition
2016–2017

Abstract

In this thesis, part of the NFFA-Europe project, different deep learning techniques are used in order to train several neural networks on high performance computing facilities with the goal of classifying images of nanoscience structures captured by SEM (scanning electron microscope). Using TensorFlow and TF-Slim as deep learning frameworks, we train on multiple and different GPU cards several state-of-the-art convolutional neural network (CNN) architectures (i.e. AlexNet, Inception, ResNet, DenseNet) and test their performances in terms of training time and accuracy on our SEM dataset. Furthermore, we coded a DenseNet implementation in TF-Slim. Moreover, we apply and benchmark transfer learning, which consists of retraining some pre-trained models. We then present some preliminary results, obtained in collaboration with Intel and CINECA, about several tests on Neon, the deep learning framework by Intel and Nervana-Systems optimized on Intel CPUs. Lastly, Inception-v3 was ported from TF-Slim to Neon for future investigations.

Contents

Chapter 1

Introduction

This thesis corresponds with the final project of the Master in High Performance Computing. The specific goals of this project are the development and benchmarking of supervised *Deep Learning* architectures and techniques for image classification of nanoscience structures. Everything is done in a high performance computing environment. Image classification, the task of giving an input image a label from a set of categories, is an active and quickly evolving topic with plenty of applications in research and industry. This work has been done within the EU-funded NFFA-Europe project [1].

The NFFA-Europe project is an integrating activity carried out in the Horizon 2020 Work Programme for European research infrastructures. Its aim is the creation of a platform *to carry out comprehensive projects for multidisciplinary research at the nanoscale, extending from synthesis to nanocharacterization to theory and numerical simulation.* This platform is the first overarching Information and Data Repository Platform (IDRP) for the nanoscience community. It will define 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] (Istituto di Officina dei Materiali) coordinates the Joint Research Activity 3 (JRA3) proposed by NFFA-Europe, which will develop and deploy the IDRP.

Within the NFFA-Europe experimental facilities, the Scanning Electron Microscopy (SEM) is one of the core characterization tools. One of these instruments, located at CNR-IOM in Trieste, featured a dataset of 150.000 images, thus represents a relevant case to experiment many neural network models, deep learning techniques and frameworks, and to compare their performances on different HPC environments.

1.1 The Starting Point of this Thesis Project

The basis of the current thesis project is the work [6] in which the authors achieved the following results

- They defined the first specific training set for SEM images (see Sec. 1.1.1), which can be used as a reference set for future deep learning applications in the nanoscience domain.
- They applied *transfer learning* approach on the SEM dataset. In particular, they used *feature extraction* from an Inception-v3 model pre-trained on the ImageNet 2012 dataset, by retraining the last layers (fully connected layer + softmax) on the SEM dataset. Moreover, they compared the results obtained with the standard TensorFlow (TF) implementation of the Inception-v3 model with the ones provided by the TF-Slim new implementation and other recent models (Inception-v4 and Inception-ResNet-v2).
- They employed the image recognition algorithm to label the test SEM images with the categories they were supposed to belong to. The accuracy they obtained, i.e. the percentage of images correctly classified, is reported in the following Table 1.1.

Model/Architecture	Accuracy [%]
`Inception-v3`	89.8 ± 0.4
`Inception-v4`	89.2 ± 0.3
`Inception-ResNet-v2`	87.5 ± 1.1

Table 1.1: Results of [6] after 5000 steps of retraining the last layers (feature extraction) of the listed models pre-trained on ImageNet 2012.

1.1.1 The SEM Dataset

SEM [4] is a common characterization technique used by nanoscientists to view their samples at higher magnifications than possible with traditional optical microscopes. SEM works by scanning 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. By raster scanning over an area, the intensity of the detected signal can be used to build a 3-channel grayscale image of the topography of a sample.

Automatic image recognition of SEM images can be extremely useful for nanoscience researchers, mainly because

- it will avoid the manual classification of the images produced;
- it provides a searchable database of nanoscience images divided into specific category which will be of easy access for scientists.

Images generated from SEM have a size of 1024×768 pixels (3 channels) and they are saved in the Tagged Image File Format (TIFF), a file format able to save some metadata about the image in a header.

Out of the 150.000 SEM images provided by CNR-IOM, 18.577 were manually classified by users into 10 categories in order to form a labelled dataset for training (90%, 16724 images) and testing (10%, 1853 images) purposes. The categories chosen spanned the range of 0-Dimensional objects such as particles, 1D nanowires and fibres, 2D films and coated surfaces, and 3D patterned surfaces such as pillars.

After the investigations of this thesis, the resulting best deep learning classification model will be performed on the whole sample of 150.000 images, through a semi-automatic process which will include the outcoming labels as an additional metadata.

1.2 HPC Environments

We present here a short description of the HPC facilities used in order to accomplish most of the computations performed in this thesis project.

1.2.1 C3HPC eXact Lab Cluster

During the thesis project, we have been provided with access on the C3E *Cloud Computing Environment* of COSILT [8], from now on called C3HPC, located in Tolmezzo (Italy) and managed by eXact-lab srl [7]. The platform is composed by one master node, where users login using the secure shell (SSH) protocol and submit computational jobs to the PBS Pro queue system; the actual computation is carried on blade servers hosted on two chassis, composed by:

- chassis 1: 7 computing nodes equipped with 2 Intel Xeon E5 – 2697 v2 CPUs (for a total of 24 physical cores) and 64 GB RAM each;
- chassis 2: 4 computing nodes, equipped with 2 Intel Xeon E5 – 2697 v2 CPUs (for a total of 24 physical cores), 64 GB of RAM and 2 GPU accelerators Nvidia Tesla K20 each, with 2496 cores and a computation peak of 3.524 GFLOPs available on each board.

The distributed storage system is composed by two I/O servers which provide the LUSTRE parallel file-system, for a total of 24 TB available. The operative system installed on C3HPC nodes is CentOS7. The C3HPC master node is reachable by SSH using the registered name hpc.c3e.exact-lab.it.

1.2.2 GALILEO CINECA Cluster

GALILEO, the Italian Tier-1 cluster for industrial and public research, introduced in CINECA on January 2015, is an IBM NeXtScale supercomputer. It is equipped with 2 Intel Phi 7120p per node on 384 nodes, and up to 2 GPU accelerators NVIDIA

Tesla K80 per nodes on 40 nodes. GALILEO master node is reachable by SSH using the registered name login.galileo.cineca.it.

1.2.3 Marconi-A2 CINECA Cluster

MARCONI is the Tier-1 system, co-designed by CINECA and based on the Lenovo NeXtScale platform. We used the A2 update which emploies 2×24-cores Intel Xeon 8160 (SkyLake) at 2.10 GHz. MARCONI master node is reachable by SSH using the registered name login.marconi.cineca.it.

1.3 Tasks and Outline of this Thesis

The aim of this project is to extend the analysis presented in [6], especially focusing on

- extensive comparison among different known deep neural networks models (or architectures);
- exploration of recent convolutional neural network architectures;
- going beyond transfer learning feature extraction: application of training from scratch and fine tuning techniques;
- benchmark of our models on multiple and different GPU cards;
- investigate the performance on CPUs of the new Intel Neon deep learning framework with respect to TensorFlow.

This thesis is organized as follows: in Chapter 2 we briefly introduce deep learning techniques for image recognition, convolutional neural networks and the frameworks used, as well as an overview of the algorithms coded throughout our work; in Chapter 3, we show and discuss the training results obtained from multiple architectures; in Chapter 4, we present benchmark results on different hardwares; in Chapter 5, we summarize the work performed and the main results achived.

Chapter 2

Deep Learning

In this Chapter we first give an overview on the general deep learning concepts applied to image recognition, in particular convolutional neural networks. We then describe the deep learning frameworks we used (TensorFlow, TF-Slim, and Neon), including some extracts from our codes for clarity, together with the needed HPC tools.

2.1 Deep Learning for Image Recognition

Nowadays, machine learning methods enhance many aspects of modern technology: from marketing to finance, from military defense to industry and health care. Moreover, machine learning is also transforming science research for its ability to reveal hidden correlation in complex datasets, allowing new insights. For instance, many application of machine learning are spreading in particle accelerator detection, cosmology, drug discovery and genomics. Conventional machine learning techniques are limited in their ability to process natural data in their raw form, such as the pixel values of an image or the words of a text. Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representation needed for detection or classification.

Deep learning methods [33] are learning techniques with multiple levels of representation, obtained by composing simple but non-linear modules each of them transforming the representation at one level, starting from the raw data input, into a representation at higher and slightly more abstract level. Turning to classification tasks, which are the tasks we want to achieve in this thesis project, higher layers of representation amplify aspects of the input that are important for discrimination and suppress irrelevant variations. An image, for instance, comes in the form of an array of pixel values, and the learned features in the first layer of representation typically represent the presence or absence of edges at particular orientations and locations in the image. The second layer typically detects arrangements of edges, regardless of small variations in the edge positions. The third layer may assemble patterns

into larger combinations that correspond to parts of familiar objects, and subsequent layers would detect objects as combinations of these parts. The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure.

As already mentioned, in this thesis project we are going to heavily use supervised deep learning techniques in image classification, the task of giving an input image a label from a set of categories. The aim is to train deep learning algorithms until they become able to generalize and automatically classify nanoscience structures.

2.1.1 Deep Learning Basics

Following the definition of [19] *"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 task in T, as measured by P, improves with experience E".*

Deep learning *tasks* are described in terms of how the system should process a *sample*, which is a collection of *features* that have been quantitatively measured from some event that we want the learning algorithm to process. In our specific case, the features of an image are the pixel brightness values. Many kinds of task can be solved with machine learning, but we are interested in *classification*. In this context, the algorithm is asked to specify which of m categories some input image belongs to. In order to solve this task, the learning algorithm is asked to produce a function $f : \mathbb{R}^n \to \{1, \ldots, m\}$. 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 case, the input is an image described by 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.

In order to evaluate the abilities of a learning algorithm, a quantitative measure P of its performance is needed. This in general is specific to the task T, but 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. Usually, with the purpose to measure how able the algorithm is to generalize what it has learned, we evaluate P using a *test set* of data that had been kept separate from the data used for training, i.e the *training set*. In machine learning, a *dataset* is defined as a collection of many examples.

Machine learning algorithms can be categorized as *supervised* or *unsupervised*

- Unsupervised learning involves observing several examples of a random vector $\vec{x}$ (think an image), and attempting to implicitly or explicitly learn the probability $p(\vec{x})$ of that distribution. Roughly speaking, unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of its structure;

- 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 $p(\vec{y} \mid \vec{x})$. Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label.

As already stressed, we are going to apply supervised learning for SEM image recognition.

2.1.2 Generalization Abilities

The central challenge in machine learning is the generalization, defined as the ability to perform well on new, previously unobserved inputs. When training a 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 or test error*, defined as the expected value of the error on a new input, to be low as well. The test error of a model is usually estimated by measuring its performance on a test set. When using a machine learning algorithm, the training set is sampled and used to choose the model parameters in order to reduce the training error, then the test set is sampled. An algorithm will perform well if

- the training error is small
- the gap between the training and test errors is small

That been said, we can express that a machine learning model is

- *underfitting* when it is not able to obtain a sufficiently low training error;
- *overfitting* when the gap between training error and test error is too large. Considering that test error is always greater or equal to training error, overfitting appears when test error is much greater than test error.

A way to predict how a model is more likely to overfit or underfit is its *capacity*, which describes how complex relationship it can model. Models with higher capacity are expected to be able to understand more relationships between more variables than models with a lower capacity. A very rough and easy way to estimate the capacity of a model is to count the number of parameters and compare it with the task the model needs to perform and the amount of training data it is provided with.

2.1.3 Parameters, Hyperparameters and Validation Set

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 determines 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 the weight of a feature 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. The values of the parameters are learned during the training process.

Hyperparameters instead have a different nature. Unlike parameters, their values are not adapted by the learning algorithm itself. They can be view as settings that can be used to control the behaviour of the algorithm. For instance, when dealing with a deep neural network (DNN), the learning rate, the number of layers, the dropout probability could be some of them. Unlike parameters, the values of hyperparameters are not adapted by the learning algorithm itself. Sometimes, to find out an optimal setting of hyperparameters, a *validation set* can be useful. The idea is to train on the training set different models each of them with slightly different hyperparameters. The validation set is then used to have an unbiased understanding of which of the different models perform better. Once this is understood, the final test error must be computer evaluating the model with the best hyperparameters on the test dataset, which contains samples that the model has never seen before. In few words, split the whole dataset in training, validation and test set. First use the training set to optimize the parameters, then use the validation set to tune the hyperparameters. Finally, when your final model is built, assess its accuracy on the test set.

2.2 Deep Feedforward Artificial Neural Networks

Deep feedforward artificial neural networks are a special class of artificial neural networks (ANNs) or multilayer perceptrons (MLPs) and form a great fraction of deep learning models. The task of a DNN is to approximate some function f, which for a classifier could be a map between an input $\vec{x}$ and a category y. An ANN defines a mapping $\vec{y} = L(\vec{x} \,|\, \vec{w})$, where $\vec{w}$ are the values of the weights that need to be learned in order to find the best function approximations. The word *feedforward* means that the information flows through the network just in one direction and therefore connections between the units do not form a cycle. An example of non feedforward ANNs, are recurrent neural networks (RNNs). From now on, we are going to use the popular term deep neural network (DNN) to indicate a feedforward neural network.

They are called *networks* since they are typically built by a composition of many different functions, called *layers*. For instance, we might have three layers l_1, l_2, l_3 connected in a chain, such that the input of a layer is the output of its previous one, as expressed by the formula

$$L(\vec{x}) = l_3(l_2(l_1(\vec{x}))). \tag{2.1}$$

The length of the chain gives the depth of the model, from which the word *deep learning* arises. The final layer of the a DNN is called *output layer*. Training a DNN means driving $L(\vec{x})$ to match as much as possible $f(\vec{x})$. The training examples

contained in the training dataset specify directly that the output layer must produce a value close to y at each input 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*.

2.2.1 Hidden Layers

The role of hidden layers can be described as applying to an input vector $\vec{x}$ a nonlinear function $h(\vec{x}\,|\,\vec{w})$ which provides a set of features describing $\vec{x}$ (or simply a new representation for it). 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 [29, 30, 31], defined by $R(z) = \max\{0, z\}$, where z is the individual output of the affine transformation on x. This solves a common problem that other activation functions exhibit: the *vanishing of gradients*. All the DNNs hidden layers we are going to use have ReLU activation functions. Now is the proper moment to introduce one of the most common type of layer, the *fully connected layer* (FCL), in which each neuron is connected to every neuron in the previous layer, and each connection has its own weight. This is a totally general purpose connection pattern and makes no assumptions about the features in the data. The high number of connections which characterize a FCL, makes them computationally demanding. These kind of layers are also very expensive in terms of memory, since the number of weight corresponds to

$$\text{weights} = \text{neurons\_previous\_layer} \times \text{neurons\_layer}. \tag{2.2}$$

However, despite their disadvantages, FCLs are essential especially at the end of a CNN. The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of a FCL is to use these features for classifying the input image into various classes based on the training dataset.

2.2.2 Stochastic Gradient Descend

Deep learning algorithms consider the problem of minimizing an *objective or loss function* that has the form of a sum

$$Q(\vec{x}, \vec{y}; \vec{w}) = \frac{1}{m} \sum_{i=1}^{m} Q_i(\vec{x}_i, \vec{y}_i; \vec{w}), \tag{2.3}$$

where m is the size of the training set and the parameters $\vec{w}$ which minimize Q need to be estimated. Each summand function Q_i is the value of the loss function associated with the i-th observation in the training dataset. When used to minimize

the above function, a standard gradient descent method would perform the following iterations on the full training set in order to compute the true gradient

$$\vec{w}_{new} = \vec{w}_{old} - \eta \nabla_{\vec{w}} Q(\vec{x}, \vec{y}; \vec{w}) = \vec{w}_{old} - \frac{\eta}{m} \sum_{i=1}^{m} \nabla_{\vec{w}} Q_i(\vec{x}_i, \vec{y}_i; \vec{w}), \tag{2.4}$$

where η is the *learning rate*. Evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions; thus, standard gradient descend operation has $O(m)$ complexity. In order to achieve good generalization performances, large dataset are necessary to train a DNN and standard gradient descend becomes easily intractable. The insight of *stochastic gradient descent* (SGD), an approximation of the gradient descent algorithm, is that the gradient is an expectation, thus it may be approximately estimated using a small set. At each step, the algorithm takes a *mini-batch* $\mathbb{B} = \{\vec{x}_1, \cdots, \vec{x}_{m'}\}$ drawn uniformly from the training set. The mini-bacth, that we will simply call *batch*, can have a size $m' < m$ ranging from one to a few hundred. If the batch size is one, the SGD is called *on-line training*, since the parameters update is done for each training example. The estimate of the gradient for the batch $\mathbb{B}$ becomes

$$\vec{w}_{new} = \vec{w}_{old} - \frac{\eta}{m'} \nabla_{\vec{w}} \sum_{i=1}^{m'} Q_i(\vec{x}_i, \vec{y}_i; \vec{w}), \tag{2.5}$$

where the loss function Q is the cross entropy in our specific trainings. This last approach can perform significantly better than on-line SGD since

- the code can make use of vectorization libraries rather than computing each step separately;
- the variance is reduced in the parameter update and can lead to more stable convergence.

With SGD, the computational 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 of m.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) [20] are a type of neural networks particularly suited for processing data that have 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 special kind of linear operation.

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* or *filter*, both dependent on an index variable t, the

convolution operation is defined as

$$c(t) = (x * w)(t) = \int x(a)w(t-a)\, da \tag{2.6}$$

Intuitively, it performs a weighted average operation on the input, giving as output a so-called *feature map* or *channel*. Of course, when working with data on a computer, the index variable will be discretized, so we can define the discrete convolution

$$c(t) = (x * w)(t) = \sum_i x(i)w(t-i) \tag{2.7}$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is a multidimensional array of parameters (namely the weights) that are adapted during the learning process. These multidimensional arrays are referred to as *tensors*. In our case, we will use convolutions over two axes. For a $2D$ image I as our input, and a $2D$ kernel K

$$C(i,j) = (x * w)(i,j) = \sum_{m,n} I(i+m, j+n)K(m,n) \tag{2.8}$$

The power of CNNs is that each convolutional filter moves across the entire input image with a step called *stride*, which will be precisely explained in Sec. 2.4.1 . These replicated units share the same parameters (weight and bias) and form a feature map. This means that all the neurons in a given convolutional layer respond to the same feature. Replicating units in this way allows for features to be detected regardless of their position in the visual field, thus constituting the property of *translation invariance*.

2.3.2 Convolutional Layer, Pooling, Dropout and Softmax

A typical convolutional layer in a network consists of three stages

- the layer performs several convolutions in parallel, each separated by a stride, to produce a set of linear activations;
- each linear activation is run through a nonlinear activation function, such as the ReLU;
- a *pooling function* is used to replace the output of the net at a certain location with a summary statistic of the nearby outputs.

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. For example, the *max pooling* operation [21] reports the maximum output within a rectangular neighborhood. Moreover, pooling is a way of sub-sampling, i.e. reducing the dimension of the input.

Dropout is a technique for addressing the problem of overfitting in deep neural networks with a large number of parameters [22]. 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 incoming and outgoing connections. 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. Dropout is applied during training or validation and switched off during the test phase.

2.3.3 Softmax Layer

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 $y_i \in [0, 1]$, and $\sum_i y_i = 1$. Formally, the softmax function is given by

$$\text{softmax}(\vec{z})_i = \frac{\exp(z_i)}{\sum_k \exp(z_k)}\,, \tag{2.9}$$

where $\vec{z}$ is the unnormalized log-probability defined by $z_i = \log \vec{P}(y = i|\vec{x})$, 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

$$\log P(y = i|\vec{x}) = \log \text{softmax}(\vec{z})_i = z_i - \log \sum_k \exp\left(z_k\right). \tag{2.10}$$

The second term in Eq. 2.10 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 Deep Learning Frameworks

In this section, we are going to have a brief overview of some deep learning frameworks and tools that we have heavily used during this project. First, we are going to introduce some basic descriptions of the TensorFlow computational graph. TF-Slim and its advantages will then be presented together with some tools which allowed us to implement deep learning models in HPC infrastructures. Lastly, we will say a few

words about Neon, a recent deep learning framework developed by Nervana-Systems and Intel, that we used in the final part of this thesis.

2.4.1 TensorFlow and the Computational Graph

TensorFlow [27] 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 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 *ops* (short for operations), and represent mathematical operations, while the graph edges represent the tensors. 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++.

TensorFlow programs are usually structured into a construction phase, that assembles a *computational 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 computations. 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` statement to specify which CPU or GPU to use for operations. On the main infrastructure we used through this work, described in Sec. 1.2.1, two GPUs are present, so we explicitly set the usage for both of them, whenever appropriate.

It is now useful to show how a convolution operation is concretely coded in TensorFlow. As described in Sec. 2.3.1, a convolution needs an input image, a kernel (or filter) and a stride vector, which controls how the kernel convolves around the input volume. Everything can be coded using `tf.nn.conv2d` op as follows

```
conv = tf.nn.conv2d(input, kernel, strides, padding=padType,
                            data_format=data_format)
```

where

- the *input*, if we consider the first layer of a CNN, can be a real image with 3 channels (RGB). The input shape is then

  `[batch_size, in_channel = 3, im_height, im_width ]`,

  where we used the NCHW format (data are presented in images, channels, height and width; see Sec. 4.3.1)

- the *kernel* must be initialized with a `tf.Variable` and have shape

  `[ker_height, ker_width, in_channels, out_channels]`

- the *stride* is a vector that specify how the sliding kernel moves for each dimension of the input. A stride such as `[1,1,1,1]` states that for each image and channel, the kernel must move by one pixel in both height and width dimensions, once the data format is fixed.

- *padding* is used to handle the output size of a convolutional or pooling layer. In *zero-padding* setting, called `"SAME"` in TensorFlow, the input volume is padded with zeros around the border and the output size is (for square images)

  `outputsize=np.ceil(np.float(inputsize)/np.float(stride)).`

  While in `"VALID"` padding scheme, which we do not add any zero padding to the input, the size of the output would be (for square images)

  `outputsize=np.ceil(np.float(inputsize - filtersize+1)/np.float(stride))`

2.4.2 TF-Slim Image Classification Model Library

TF-slim is a lightweight high-level API of TensorFlow for defining, training and evaluating complex models [23, 24]. In particular, TF-Slim allows the user to define models much more compactly by eliminating boilerplate code. This is accomplished through the use of *argument scoping* and numerous high level layers and variables. These tools increase readability and maintainability, reduce the likelihood of an error from copy-and-pasting hyperparameter values and simplifies hyperparameter tuning. Moreover, TF-Slim makes it easier to extend complex models, and to warm start training algorithms by using pieces of pre-existing model checkpoints.

Just to give an example, if we want to build a convolutional layer in native TensorFlow this could be rather laborious, as we can notice in the following code

```
input = ...
with tf.name_scope('conv1_1') as scope:
  kernel = tf.Variable(tf.truncated_normal([3, 3, 64, 128]),
         name='weights')
  biases = tf.Variable(tf.constant(0.0, shape=[128]),
```

```
            trainable=True, name='biases')
    conv = tf.nn.conv2d(input, kernel, [1, 1, 1, 1], padding='SAME')
    bias = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(bias, name=scope)
```

where first some `tf.Variable` are needed to define the kernel and the bias. Secondly, the convolution operation `tf.nn.conv2d` is invoked, added to the bias and evaluated in a ReLU activation function, i.e. `tf.nn.relu`. To alleviate the need to duplicate this code repeatedly for each convolutional layer, TF-Slim provides a number of convenient operations defined at the more abstract level of neural network layers. For example, using TF-Slim we can write the same code shown above as follows

```
input = ...
net = slim.conv2d(input, 128, [3, 3], scope='conv1_1')
```

where `[3,3]` are the dimensions of the filter. In addition to some scope mechanisms in TensorFlow, TF-Slim adds a new scoping mechanism called `arg_scope`. This new scope allows a user to specify one or more operations and a set of arguments which will be passed to each of the operations defined in the `arg_scope`. This functionality is best illustrated by example. Consider the following code snippet

```
net = slim.conv2d(inputs, 64, [11, 11], 4, padding='SAME',
                  weights_initializer=tf.truncated_normal_initializer(
                                                    stddev=0.01),
                  weights_regularizer=slim.l2_regularizer(0.0005),
                                                    scope='conv1')
net = slim.conv2d(net, 128, [11, 11], padding='VALID',
                  weights_initializer=tf.truncated_normal_initializer(
                                                    stddev=0.01),
                  weights_regularizer=slim.l2_regularizer(0.0005),
                                                    scope='conv2')
net = slim.conv2d(net, 256, [11, 11], padding='SAME',
                  weights_initializer=tf.truncated_normal_initializer(
                                                    stddev=0.01),
                  weights_regularizer=slim.l2_regularizer(0.0005),
                                                    scope='conv3')
```

It should be clear that these three convolution layers share many of the same hyperparameters. Two have the same padding, all three have the same `weights_initializer` and `weight_regularizer`. This code is hard to read and contains a lot of repeated values that should be factored out. By using an `arg_scope`, we can both ensure that each layer uses the same values and simplify the code

```
with slim.arg_scope([slim.conv2d], padding='SAME',
      weights_initializer=tf.truncated_normal_initializer(stddev=0.01)
      weights_regularizer=slim.l2_regularizer(0.0005)):

    net = slim.conv2d(inputs, 64, [11, 11],scope='conv1')
    net = slim.conv2d(net, 128, [11, 11],padding='VALID',scope='conv2')
    net = slim.conv2d(net, 256, [11, 11],scope='conv3')
```

In all the results we are going to present is Chapter 3 and 4, we used TF-Slim library both for training our networks from scratch and for transfer learning.

2.4.3 TensorFlow in HPC: Docker, TFrecords, GPUs

For the deep learning computations needed for this project, we have been provided with access to `C3HPC` cloud computing environment nodes. The operative system installed on a node of `C3HPC` is `CentOS7`, but the only Linux distribution TensorFlow runs on is 64-bit Ubuntu. Another technical issue was that our SEM dataset, hosted on a `LUSTRE` parallel file system, is huge and cannot fit into our GPUs memory. To solve these problems, the following tools were used to interconnect TensorFlow with our computing resources: Docker Containers and TFRecords data.

Docker Containers

Since TensorFlow provides a Docker image on the website, the best option has been to install a Docker container on the nodes, link everything with proper CUDA libraries and work inside it. *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. 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 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. For this reason, performance of an application inside a container is generally better compared to the same application running within a virtual machine.

Reading and Importing Data: TFRecords

There are three ways to read the training dataset in TensorFlow. The first is to read it directly from a CSV file. A second and third ones are to read it as single elements from disk or as groups of elements from disk. The single data read mode is not suitable for large datasets that cannot fit into the GPU memory; the best way to feed TensorFlow when dealing with a huge amount of elements is using the so-called *TFRecords*, a file format that packs elements along with the categories they belong to. Convert datasets of images to TFRecords can be done with a simple script that accepts as input the path to a folder containing the data subdivided in sub-folders, one for each category. The resulting TFRecords are slightly larger than the sum of the original image sizes, due to the addition of the metadata such as the category

they belong to. Once the TFRecords are available, they can be loaded and placed in the GPU devices memory, to reduce I/O to disk and speed up training processes.

2.4.4 Nervana-Neon and its Inception-v3 Implementation

Neon is a deep learning framework created by Nervana Systems. After Nervana joined Intel, they have been working together to bring superior performance to Intel CPU platforms using Intel Math Kernel Library (MKL). Intel MKL provides CPU optimized implementations for widely used primitives like convolution, pooling, activation functions and normalization. These MKL primitives exploit the full vectorization and parallelization capabilities of Intel architecture in contrast to existing vanilla implementations.

In the final part of this thesis project, in collaboration with Intel, we had the opportunity to benchmark some Neon codes and to compare them with equivalent TensorFlow implementations. Our preliminary results are presented in Sec. 4.3.3 of Chapter 4. Furthermore, for future tests we implemented a Neon version of Inception-v3, as shown in Sec. 2.5.2.

2.5 Into Different CNN Architectures: AlexNet, Inception, DenseNets

In this section, we are going deeper in the understanding of the architecture of some CNNs with the purpose of better explaining some details using TensorFlow language. As we previously saw, a CNN consists of an input and an output layer, as well as multiple hidden layers (for instance convolutional layers, pooling layers, fully connected layers).

With the aim to outperform in some relevant competition in image classification, for example the *ImageNet Large Scale Visual Recognition Challenge* (ILSVR) [28], deep learning experts design many clever combinations of the fundamental layers mentioned above. Especially, we are going to describe the main three models studied in this project: AlexNet, Inception-v3 and DenseNet. It is worth mentioning that, as one of the results of this thesis project, the DenseNet architecture was coded almost from scratch in TensorFlow and then adapted in TF-Slim.

2.5.1 AlexNet: the father of modern CNN

AlexNet [11] is the name of the first CNN which competed and won the ILSVR in 2012. The network achieved a top-5 error of 15.4 (*Top* 5 *error* is the rate at which, given an image, the model does not output the correct label with its top 5 predictions). The next best entry achieved an error of 26.2%, which was an astounding

improvement that pretty much shocked the computer vision community. From that moment on, CNNs became household names in the competition. AlexNet uses a relatively simple layout, compared to modern architectures, and was the first model to implement dropout. It contains only 8 layers, first 5 are convolutional layers followed by fully connected layers. In this project, we used AlexNet in Chapter 3 for a comparison with more advanced CNNs architectures and in Chapter 4 for benchmarking purposes.

2.5.2 Inception-v3

Inception-v3 [12, 13] is the 2015 iteration of Google 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 [34, 35].

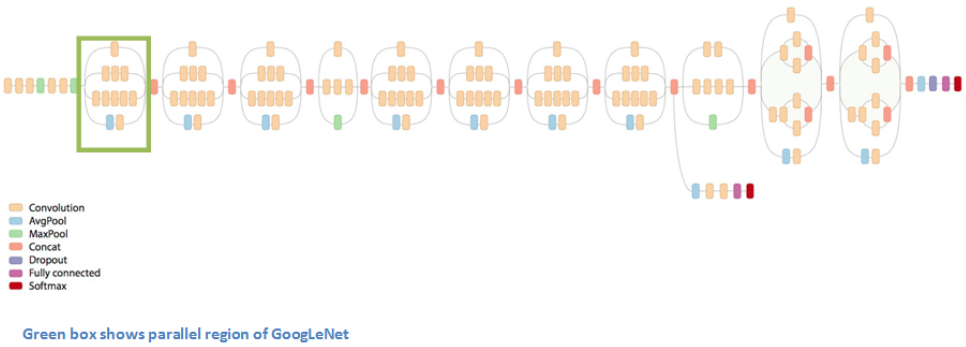

Figure 2.1: Inception-v3 architecture, which is composed by 11 inception modules plus some traditional convolutional layers at the beginning and some fully connected layers at the end.

A schematic picture of such an architecture is shown in Fig. 2.1. The model is a multi-layer architecture consisting of alternating convolutions and non-linearities, using ReLU activation functions in all the stages. These layers are followed by a fully connected layer leading to a softmax classifier. The relevant feature to notice about this model is that not everything is happening sequentially, as in AlexNet.

Some pieces of the network, called *inception modules*, are happening in parallel. Basically, at each layer of a traditional CNN, you have to make a choice of whether to have a pooling or a convolution, a convolution with kernel size $k \times k$ or $k' \times k'$. What an Inception module allows you to do is to consider multiple branches, perform all of these operations in parallel and concatenate the results at the end of the module using `tf.concat(axis= 3, values=[branch_0, branch_1, branch_2,...])`. However, the output ends up with an extremely large depth channel. The authors addressed this is by adding 1×1 convolution operations before the 3×3 and 5×5 layers. The 1×1 convolutions provide a method of dimensionality reduction and solves the

problem. For instance, we show below the 4th inception module (for clearness we hide some arguments in the ellipsis)

```
with tf.variable_scope(end_point):
      with tf.variable_scope('Branch_0'):
          branch_0 = slim.conv2d(net, depth(384), [3, 3], ...)
      with tf.variable_scope('Branch_1'):
          branch_1 = slim.conv2d(net, depth(64), [1, 1], ...)
          branch_1 = slim.conv2d(branch_1, depth(96), [3, 3], ...)
          branch_1 = slim.conv2d(branch_1, depth(96), [3, 3], ...)
      with tf.variable_scope('Branch_2'):
          branch_2 = slim.max_pool2d(net, [3, 3], ...)
        net = tf.concat(axis=3, values=[branch_0, branch_1, branch_2])
end_points[end_point] = net
```

During this project, we widely used Inception-v3 architecture in TF-Slim for the classification of our SEM images as it will be presented in Chapters 3 and 4.

In addition, as a result of this thesis, we coded from scratch Inception-v3 with Neon deep learning framework. This was done for future tests, comparisons, and research. For an immediate comparison, we show below the same Inception module ported in Neon

```
branch0 = Sequential(layers = [Conv((3, 3, 384), ...])
branch1 = Sequential(layers = [Conv((1, 1, 64), ...), Conv((3, 3, 96),
                                    ...), Conv((3, 3, 96), ...)])
branch2 = Sequential(layers = [Pooling(3,  op = "max", ...)])
layers.append(MergeBroadcast(layers = [branch0, branch1, branch2],
                                    merge = "depth"))
```

We can immediately notice the readability of Neon implementation with respect to TF-Slim even though both are built on a higher abstraction level w.r.t. TensorFlow.

2.5.3 Densely Connected Convolutional Neural Networks

Densely Connected Convolutional Neural Networks [14, 15], in short called DenseNets, are network architectures with *dense blocks*, i.e blocks where each layer is directly connected to every other layer in a feed-forward fashion. For each layer, the feature-maps of all preceding layers are concatenated and used as inputs, and its own feature-maps are used as inputs into all subsequent layers.

Differently from the general Eq. 2.1, in a DenseNet the l^{th} layer receives the feature-maps of all preceding layers

$$\mathbf{x}_l = H_l([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}]), \tag{2.11}$$

where $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}]$ refers to the concatenation of the feature-maps produced in layers $0, \ldots, l-1$. Whereas traditional convolutional networks with L layers have L connections (one between each layer and its subsequent layer), DenseNets have

$L(L+1)/2$ direct connections. Between dense blocks, a *transition block* is always inserted. Its structure is in general composed by a 1×1 convolutional layers followed by a pooling layer.

DenseNets have several compelling advantages, for instance they

- strengthen features/information propagation and encourage feature reuse;
- substantially reduce the number of parameters, as shown in Table 2.1;
- alleviate the vanishing-gradient problem.

This connectivity pattern yields state-of-the-art accuracies on CIFAR10/100 and SVHN. On the large scale ILSVRC 2012 dataset, DenseNets achieve a similar accuracy as ResNets, but using less than half the amount of parameters and roughly half the number of FLOPs.

An important hyperparameter in DenseNets is the *growth rate* k. If each function H_l produces k feature-maps as output, it follows that the l^{th} layer has $k \times (l-1) + k_0$ input feature-maps, where k_0 is the number of channels in the input image. In DenseNets with many layers, as DenseNet-121 or higher, to prevent the network from growing too wide and to improve the parameter efficiency, we limit k to a small integer, typically between 12 and 48. In [15], with the aim of reducing memory and time consumption, it is suggested to reduce the depth of the network and leverage the growth rate. Driven by this advice, as one of the result of this thesis project, we coded in TF-Slim a few of this shallow and wide DenseNets. In particular, let us enter in more details of our implementation of DenseNet-28(56). This model has growth rate $k = 56$ and three dense blocks of size $[2, 6, 4]$, two transition blocks plus an initial convolutional layer and final fully connected one. In total 28 layers are presents, as the name and a simple computation suggest (consider that each dense block has two convolutional layers).

Architecture	Num of Parameters
`DNet-121 (k=32)`	$\sim 6.9M$
`DNet-91 (k=32)`	$\sim 4M$
`DNet-40 (k=48)`	$\sim 2.7M$
`DNet-28 (k=56)`	$\sim 2M$
`Inception-v3`	$\sim 27.1M$
`AlexNet`	$\sim 50.3M$

Table 2.1: Number of parameters (weights and biases) of some of the models we used. Notice that DenseNet architecture substantially reduce the number of parameters w.r.t. the other conventional CNNs.

In this thesis project, we heavily used DenseNet architectures and touched their advantages. In particular, DenseNet-121 was employed in training and transfer learning

showing up extremely good performances in terms of accuracy and timing to converge. Furthermore, DenseNet-91, 40 and 28 were tested and the latter in particular outperformed in training on SEM dataset, as it will be presented in Chapter 3.

2.6 Training from Scratch and Transfer Learning

In what follows, we present the two deep learning techniques we employed by means of DNNs. First we introduce *training from scratch*, the most intuitive approach when dealing with a DNN which needs to learn some task from a dataset. Lastly, we illustrate *transfer learning*, which consist of retraining your DNN starting from a checkpoint of some pre-trained model, which has already learned from some huge dataset.

2.6.1 Training From Scratch

If you have at you disposal a large dataset, the most common way to start a supervised learning process of a DNN model is training it from scratch. The network in the initial step has a clean, random initialized set of parameters: weights and biases. The learning procedure is composed by many training steps, each of them consists of a feedforward flow of one batch of images followed by a back-propagation. Step by step these parameters are optimized according to minimize a loss function using, stochastic gradient descend (SGD). In Chapter 3, we will analyze in more details this process. For the moment, let us introduce some quantities that are customary in deep learning. For convenience, we define them in line with TF-Slim

- *training step*: once fixed the batch size, one training step corresponds to one batch of images processed by the algorithm. In a training, we have as much back-propagations as the number of steps.
- *epoch*: a DNN is said to have been trained up to an epoch, if it has been fed with a number of images equal to the dataset size. In terms of training steps, an epoch corresponds to

$$\texttt{num\_steps\_epoch} = \frac{\texttt{dataset\_size}}{\texttt{batch\_size}}$$

When training a model, it is often recommended to lower the learning rate as the training progresses. In TF-Slim, we used the `tf.train.exponential_decay` operation, which applies an exponential decay function to a provided initial learning rate `lear_rate`. The function returns the decayed learning rate, which is computed by means of

$$\texttt{decayed\_lear\_rate} = \texttt{lear\_rate} \times (0.94)^{(\texttt{global\_step}/\texttt{decay\_steps})}, \quad (2.12)$$

where the `global_step` represents the total number of steps done in a training at a fixed time, while

$$\texttt{decay\_steps} = \texttt{num\_step\_epoch} \times \texttt{num\_epoch\_per\_decay} \tag{2.13}$$

2.6.2 Transfer Learning

Many DNNs share a curious phenomenon: on the first layer they learn features similar to Gabor filters and color blobs. These first-layer features appear to be general and not specific to a particular dataset or task. They are applicable to many datasets and tasks. Features switch from general to specific by the last layer of the network. For example, in a network with a 10-dimensional softmax output layer that has been succesfully trained toward a supervised classification objective, each output unit will be specific to a particular class.

In transfer learning [18], initially a *base network* is trained on a *base dataset* and *task* and then the learned features are transferred to a second *target network* retrained on a *target dataset* and *task*. When target dataset is significantly smaller than the base dataset, transfer learning can be a powerful tool to enable training a large network without overfitting. Transfer learning allows you to transfer knowledge from one model to another. For example, you could transfer image recognition knowledge from a cat recognition app to a radiology diagnosis. Implementing transfer learning involves retraining the last few layers of the network used for a similar application domain with much more data. The idea is that hidden units earlier in the network have a much broader application which is usually not specific to the exact task that you are using the network for. In summary, transfer learning works when both tasks have the same input features and when the task you are trying to learn from has much more data than the task you are trying to train. In Sec. 3.2 of Chapter 3, where we will operatively apply transfer learning, we will go forward in the presentation of some details and techniques, such as *feature extraction* and *fine tuning*.

Chapter 3

Scientific Results

In this Chapter, we present the scientific results of this thesis project. In particular, for scientific results we refer to the extensive comparison performed among different DNN architectures and different deep learning techniques, applied to the SEM dataset. A measure of how good is a specific DNN architecture or a technique is the accuracy in the classification task computed on the test set, which is composed by images that the algorithm has never seen before. Together with the accuracy, we also consider the time is needed by these DNNs to converge to a stable value of the loss function and accuracy itself. In particular, we are going to discuss training from scratch in Sec. 3.1, comparing very different models; then in Sec. 3.1.1 we study the training behaviour of some DenseNets. Lastly, in Sec. 3.2 we present an analysis of transfer learning (feature extraction and fine tuning), considering ImageNet as a base dataset. All the computations shown in this chapter were done, unless explicitly written, on `C3HPC` cluster equipped with two `Tesla K20` Nvidia GPUs loading the Nvidia CUDA Deep Neural Network library (`cuDNN`). The discussion about technical results, as scalability and benchmarks on different computing resources, together with the way TensorFlow works on multiple GPUs will be done in Chapter 4. It is worth specifying that all the trainings we present in our plots have been selected to be the best among a set of trainings with slightly different hyper-parameters (learning rate, weight decay and others).

3.1 Training From Scratch on SEM dataset: comparison between different architectures

As mentioned in Sec. 2.6, the most natural and standard way to start a supervised learning procedure of a DNN model is training it from scratch. The network in the initial step has a clean, random initialized set of weights and biases (parameters). The learning procedure, which consist of many training steps each of them composed by a feedforward flow of one batch of images followed by a back-propagation, will optimize these parameters in order to minimize a loss function (in our case cross-

entropy) using, in our case, stochastic gradient descend (SGD). The initialization mentioned above is in general different layer by layer, depending also on the layer nature and activation function. The solution (and convergence) of a non-convex optimization algorithm (as SGD) depends on the initial values of the parameters and therefore the problem of the initialization of a neural network is non trivial and has been well studied in the past [9, 10]. In our case, we decided to initialize our models with standard methods such as Gaussian or uniform random initialization with fairly arbitrarily set variances.

In the field of deep learning applied to computer vision the first CNN that outperformed and won the *ImageNet ILSVRC 2012* was AlexNet [11], described in Sec. 2.5.1. Therefore, as a first attempt, we trained this simple model on our SEM dataset and reached 73% of accuracy. However, this results is quite poor and some improvements concerning the complexity of the CNN architecture were necessary. As a next step, we trained the version v3 and v4 of the more complex Inception architecture [12, 13], implemented in the winning *ImageNet ILSVRC 2014* submission and described in Sec. 2.5.2. In addition to the above standard architectures, we also explored some versions of the more recent Densely Connected CNN (DenseNet) [14], discussed in Sec. 2.5.3. In Chapter 4, we will understand that GPU memory is often a bottleneck that prevents employing large batch sizes in complicated architectures. For the moment, let us say that in order to avoid OOM (out of memory) issues on our `Tesla K-20` devices ($\sim$ 5 GB) and with the aim to fairly use the same batch size for all the above different models, we initially choose it to be 16. We remind that the batch size (sometimes called mini-batch) corresponds to the number of training images that the DNN will process during training before the update of its parameters. Inception-v4 and DenseNet-121 with batch size greater than 16 run OOM due to their complexity (for instance they required high number of floating point operations (FLOPs))[1]. In the main panel of Fig. 3.1, the accuracy computed on the test dataset during the 470 epochs of training with batch size 16 is shown. In terms of test accuracy, the more recent architectures Inception-v3, Inception-v4 and DenseNet-121 reach a visibly higher value than the simpler AlexNet after 470 epochs. Inception-v3 is performing better than Inception-v4, at least on SEM dataset. In Table 3.1, we report the test accuracy of the models[2] together with the time (in hours) required to reach 470 epochs of training. This number of epochs is enough for Inception-v3 loss function (and accuracy) to reach a stable minimal (and maximum) value. On the other side, Inception-v4 at 470 epochs does not seem to have converged yet. Besides the fact that Inception-v4 is less precise in the classification, it is also the slowest architecture to converge. In particular, it requires twice the time of Inception-v3 to reach 94% accuracy on SEM dataset. DenseNet-121 instead is comparable or higher than Inception-v3 in terms of accuracy. Moreover, it seems to learn faster in terms of number of epochs (the same can be said when looking at the loss function). The accuracy is already above 90% after 30 epochs. After these considerations, we decided to rule out Inception-v4 for what concerns training from scratch. Moreover,

[1] AlexNet has twice the number of parameters of Inception-v3 but its simple architecture requires far less FLOPs for each forward and backward propagation.

[2] This accuracy is the result of an average among the last ten evaluations (around the last 90 minutes of training), the error corresponds to the standard deviation.

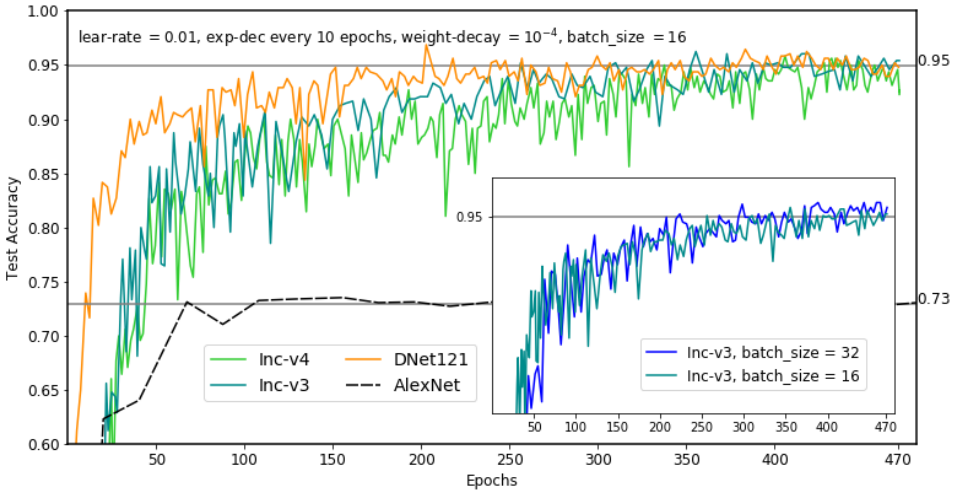

Figure 3.1: (main) Training from scratch Inception-v3 (lighter blue), Inception-v4 (green), DenseNet-121 (orange) and for comparison AlexNet (black) on SEM dataset. All these models were trained with learning rate 0.01 exponentially decreasing every 10 epochs, weight decay of 10^{-4} and batch size 16, except for AlexNet whose initial learning rate was set to 0.007. (inset) Inception-v3 trained with different batch sizes. The larger the batch size the better as long as we do not reach out of memory issues.

we decided to modify the DenseNet-121 architecture with the idea to decrease the number of layers and thus the complexity of the model in order to train it with a larger batch size. In the inset plot of Fig. 3.1, we show how accuracy increases when setting higher batch size for the Inception-v3 model. It is useful to remind here that an epoch corresponds to a number of images processed which is equal to the size of the training dataset. Therefore, after one epoch of training a network process the same number of images independently of the batch size choosen. In the last line of Table 3.1, the accuracy and the time needed to train Inception-v3 with batch size 32 up to 470 epochs are shown. Looking in more detail at the third column of Table 3.1, if we double the batch size, the time needed to complete 470 epochs is reduced by 10%. At first sight this might sound weird, but it can be explained by the following considerations. A training step consists in a feedforward plus a backward propagation of one batch of images. In Chapter 4, we will see that the backward propagation (i.e the parameter updating) takes more time with respect to the forward process. Therefore, at 470 epochs the training with batch size 32 has done half training steps (and consequently half parameters updating) than the one trained with batch size 16. Everything is in accordance with a general rule of thumb that we are going to list below (third item). After these first results, we can collect some considerations

- At a fixed batch size Inception-v3 is performing better than Inception-v4 in terms of both accuracy and time needed to complete the training;
- DenseNet-121 and Inception-v3 are comparable in accuracy, but DenseNet-121 is converging faster;
- Once a model has been chosen, the larger the batch size the better. However,

Architecture	Test Accuracy [%]	Time to 470 eps (hours)	Stable at (hours)
`Inc-v3, bs=16`	95.3 ± 0.2	76.7	~ 76
`Inc-v4, bs=16`	94.0 ± 0.1	160.7	> 160
`DNet-121, bs=16`	95.4 ± 0.6	58.5	~ 47
`Inc-v3, bs=32`	96.4 ± 0.2	69.8	~ 70

Table 3.1: Comparison between training results of different models, using batch size 16. The training was performed until 470 epochs were reached. In the fourth column we report the time in hours when the test accuracy becomes stable. In the last line, we report the same data for Inception-v3 trained with batch size 32.

we need to trade off the batch size and OOM issues. This is a well known fact in deep learning community.

With the aim of leverage the batch size of our training from 16 to 32, we decided to code a less complex version of DenseNet-121 which does not run OOM on our devices. The result is DenseNet-28, which has growth rate $k = 56$ and three dense blocks of size $[2, 6, 4]$ (a detailed description in Sec. 2.5.3), two transition blocks plus an initial convolutional layer and a final fully connected one (in total 28 layers, considering that each dense block has 2 convolutional layers). In the main panel of Fig. 3.2, we show the results of training Inception-v3 and DenseNet-28 up to 760 epochs. As reported in Table 3.2, Inception-v3 slightly exceed DenseNet-28 in terms of accuracy but it is more than twice slower, as can be appreciated looking at the inset of the same figure. As described in Sec. 2.5.3, DenseNet-28 has $2M$ parameters to train against $27M$ of Inception-v3 (DenseNet-121 with growth rate $k = 32$ has 6.8M parameters). A simple TensorFlow script was built in order to count the exact number and type of parameters given a model checkpoint.

Architecture	Test Accuracy [%]	Time to 760 eps (hours)	Stable at (hours)
`Inc-v3, bs=32`	96.6 ± 0.2	110.9	~ 70
`DNet-28, bs=32`	96.2 ± 0.4	41.5	~ 33

Table 3.2: Performances of Inception-v3 (blue) and DenseNet-28 (orange). While the former reaches a slightly higher accuracy, the latter is more than two time faster.

3.1.1 Comparing Different DenseNets: 91, 40 and 28 layers

In order to understand which combination of dense and transition blocks in the DenseNet architecture was more suitable for our SEM dataset, we performed several studies and benchmarks. In [15], the authors suggest to use a wide and shallow DenseNet with the purpose of reducing memory and time consumption. To obtain a wide DenseNet, we set the depth (number of layers) to be smaller ($L = 91, 40, 28$) and the growth rate of feature maps to be larger ($k = 32, 48, 56$). In Fig. 3.3, we

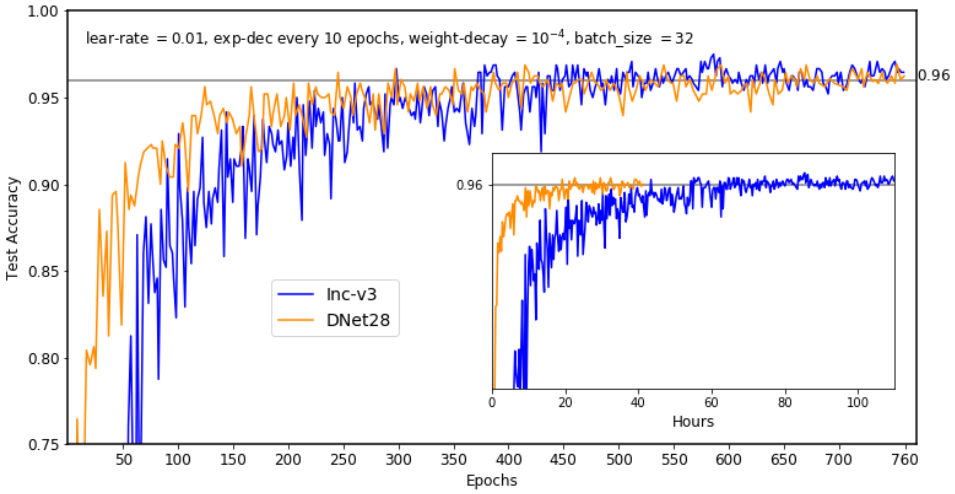

Figure 3.2: (main) Training from scratch Inception-v3 and DenseNet-28 on SEM dataset up to 760 epochs. All these three models were trained with an initial learning rate of 0.01, exponentially decreasing every 10 epochs, weight decay of 10^{-4} and batch size 32. (inset) Test accuracy against training hours needed by the two models.

show the training progress up to 970 epochs on SEM dataset of DenseNet-91, 40 and 28 respectively. In this case, we run our computations on four `Nvidia GTX-1070` GPUs (available on `C3HPC` cluster for a short period) using batch size equal to 56. Despite the fact that DenseNet-28 is the network with fewest layers, its accuracy is comparable to the others. Probably this is due to the higher growth rate ($k = 56$) set for this model. Considering the time to complete 970 epochs, DenseNet-28 is the fastest being that it requires less FLOPs. In Table 3.3, precise results are listed and from them it emerges that DenseNet-28 is the more suitable architecture for our SEM dataset, at least when dealing with training from scratch.

Architecture	Test Accuracy [%]	Time to 970 eps (hours)	Parameters
`DNet-91 (k=32)`	95.3 ± 0.3	36.6	$\sim 4M$
`DNet-40 (k=48)`	95.5 ± 0.6	29.1	$\sim 2.7M$
`DNet-28 (k=56)`	95.7 ± 0.2	24.8	$\sim 2M$

Table 3.3: Accuracy and time required to complete training up to 970 epochs are listed. DenseNet-28 reaches the highest accuracy in the shortest time. All the trainings were done on four `Nvidia GTX-1070` GPUs on `C3HPC` cluster. For each model, the learning rate was initially set to 0.01 and then let exponentially decay every 3 epochs.

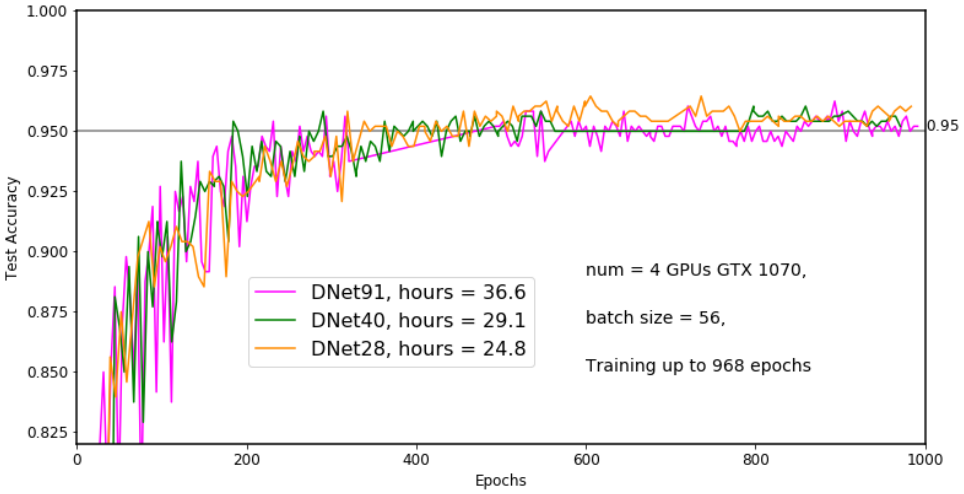

Figure 3.3: Training from scratch of three different DenseNet architectures: DenseNet-91 with growth rate 32, DenseNet-40 with growth rate 48, DenseNet-28 with growth rate 56. Each training was done using four `Nvidia GTX-1070` GPUs on `C3HPC` cluster.

3.2 Transfer Learning from ImageNet to SEM dataset

As anticipated in Sec.. 2.6.2, transfer learning is a technique which is becoming very popular in deep learning. It focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. In this section we apply transfer learning on our SEM dataset. This means that once a deep learning architecture (say Inception-v3) is chosen as a *target network* to be trained on a *target dataset*, we can apply transfer learning starting from a pre-trained *checkpoint*, i.e. an equivalent neural network architecture, called *base network*, whose parameters are not randomly initialized but have been already trained on an huge *base dataset* to solve a different but similar problem.

In our specific case, we are going to test transfer learning on our target SEM dataset starting from some checkpoints pre-trained on the base dataset ImageNet (ILSVRC-2012-CLS), a large visual database designed for use in visual object recognition software research. As explained in Sec. 2.6.2, transfer learning is in general applied when your dataset is too small to train a huge neural network architecture and you want to avoid overfitting. The strengths of this approach are that it is faster than a training from scratch and, in its simplest version, can be easily performed with modest computing resources, for instance without GPUs. One of the weaknesses is that you are not free to use whatever architectures but you are constrained to adapt your choice to the available pre-trained checkpoints present in the literature.

In this project, transfer learning was widely studied using ImageNet pre-trained checkpoints of Inception-v3, Inception-v4, ResNet-Inception-v2 and DenseNet-121. Similarly to the case of training from scratch, Inception-v3 and in particular DenseNet-121 demonstrated better performances (accuracy and timing) on our target SEM

dataset. We were not able to benchmark transfer learning on other DenseNets, such as DenseNet-28, since any ImageNet pre-trained checkpoint of this architecture is present yet. The only available pre-trained DenseNet models on ImageNet are: version 121, 169, 161, 201 [16]. However, their implementation is for Caffe (another common deep learning framework) and a TensorFlow conversion was performed on the 121 layers version, the only one that was easily tractable according to our resources and batch size requirements. In greater detail, we applied the following transfer learning methods:

Feature Extraction: start with a pre-trained checkpoint, reset and random initialize only the parameters which belong to the last layer. Then, retrain just them. In TF-Slim and considering as an example Inception-v3, the following are the relevant flags for these two operations, respectively

`-checkpoint_exclude_scopes` = `InceptionV3/Logits,InceptionV3/AuxLogits`,

`-trainable_scopes` = `InceptionV3/Logits,InceptionV3/AuxLogits`,

where `Logits` and `AuxLogits` identify the last layer.

(Complete) Fine Tuning: start with a pre-trained checkpoint, reset and random initialize only parameters which belong to the last layer. Then retrain the network allowing back-propagation through all the layers. In TF-Slim, this is achieved just removing the second flag above since by default all the scopes (i.e layers) are retrained.

In the main plot of Fig. 3.4, test accuracy evaluations during fine tuning on SEM dataset of Inception-v3 (blue line) and DenseNet-121 (orange line) pre-trained on ImageNet are shown. It should be noted that, differently from the previous training plots, here we decided to put on the x-axes the number of hours needed to complete the computations up to some epochs, in order to emphasize differences. Both Inception-v3 and DenseNet-121 were fine tuned up to 214 epochs, but the latter took significantly less time to reach a stable higher accuracy of 97.3% in the classification of test images. The results are reported in Table 3.4. ln addition to fine tuning, also feature extraction was performed on both models (magenta and red lines), as shown in main Fig. 3.4 (look at the inset plot for a zoom). In both cases, it is extremely fast but on the other side it cannot reach an accuracy higher than 90%. Again, DenseNet-121 appears to learn faster than Inception-v3. Feature extraction is the simplest transfer learning technique. Differently from fine tuning (or training from scratch) we expected limited performances in respect of accuracy. After all, only the last layer (i.e the classifier) is re-trained and this process requires a small number of epochs. In the inset panel of Fig. 3.4, feature extraction was applied also to Inception-v4 and Inception-Resnet-v2. However, neither of them outperforms feature extraction applied on DenseNet-121. Similar results about feature extraction were obtained in [6], the starting point of this thesis project.

It is also possible to combine feature extraction and fine tuning. If we start from a pre-trained checkpoint, the common practice is to first apply feature extraction on the target dataset (in our case SEM dataset) and then fine tune the entire model on that dataset. In this case, the fine tuning starts with last layer weights already

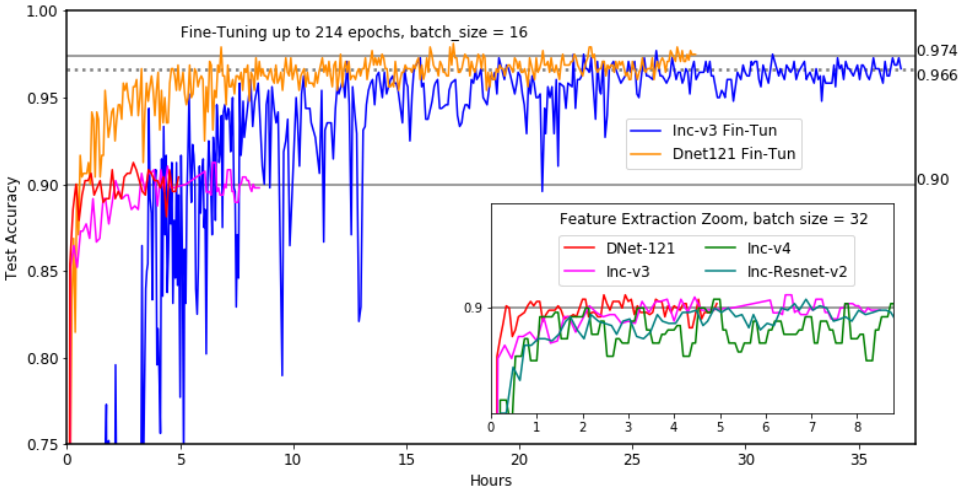

Figure 3.4: Test accuracy results of fine tuning of Inception-v3 (blue) and DenseNet-121 (orange) pre-trained on ImageNet. The latter model exceeds the former one both in terms of accuracy and computing time. In the inset plot, feature extraction results are shown.

Architecture	Test Accuracy [%]	Time to 214 eps (hours)	Stable at (hours)
`Inc-v3 F-Tun`	96.7 ± 0.4	37.0	~ 32
`DNet-121 F-Tun`	97.3 ± 0.3	27.8	~ 20
`Inc-v3 F-Ex`	90.0 ± 0.5	–	~ 4
`Inc-v4 F-Ex`	89.2 ± 0.3	–	~ 4
`Inc-Res-v2 F-Ex`	89.2 ± 0.3	–	~ 4
`DNet-121 F-Ex`	89.9 ± 0.2	–	~ 1

Table 3.4: For fine tuning, the test accuracy values of Inception-v3 and DenseNet-121 were averaged considering the last 90 minutes of training. In a day of training on two `Tesla K20` GPUs, the fine tuning of DenseNet-121 from an ImageNet checkpoint reaches 97.3% of test accuracy. Feature extraction exhibit limited performances in terms of accuracy, but is really fast to converge.

trained and it may take less time to converge. However, when the number of classes is not huge, as in our case (10 classes), we noticed that there are not relevant gains in performances when applying this practice. Starting with fine tuning was already a good choice. On the other side, one may apply feature extraction after fine tuning just to stabilize the final classification and sometimes gain a small fraction of accuracy.

Chapter 4

Technical Results

In the first part of this Chapter we present the benchmarks and the speedups obtained on trainings with TensorFlow model implementations on multiple and different GPU cards (Graphics Processing Unit). In the second part of the Chapter, we discuss some benchmarks that have been done on AlexNet architecture on different computational infrastructures and different deep learning frameworks, such as TensorFlow and Neon (Nervana Systems, Intel) [26]. It is a well known fact that GPUs are perfectly suited to deep learning since the type of calculations they were designed to process happens to be the same as those encountered in deep learning. Images, videos, and other graphics are represented as matrices, so that when you perform any operation, such as a zoom in effect or a camera rotation, all you are doing is applying some mathematical transformation to a matrix. GPUs, compared to central processing units (CPUs), are more specialized at performing matrix operations and several other types of advanced mathematical transformations. In general, this makes deep learning algorithms run several times faster on a GPU compared to a CPU. Learning times can often be reduced from days to mere hours. However, GPUs memory is often a bottleneck that prevents to employ large batch sizes in complicated architectures. Also for this reason, recent hardware developments pursued by Intel are pushing towards leveraging CPUs power in deep learning operations. In the final part of this thesis project, we had the opportunity to interact with Intel and perform some tests on Neon, the deep learning framework developed by Nervana-Systems and Intel, which is optimized for Intel CPUs. In the last sections of this Chapter, we report our preliminary results obtained so far, which are promising and pave the way to future investigations.

4.1 TensorFlow on Multiple GPUs

Our available computational resources contain multiple GPUs for scientific computation. Therefore, TensorFlow can leverage this environment to run the training operation concurrently across multiple cards. Training a model in a parallel distributed fashion requires coordinating training processes. For what follows we call

model clone a copy of a model in a GPU[1]. There are two main approaches when training a model on multiple GPUs: *asynchronous* and *synchronous* training. Naively employing asynchronous updates of model parameters lead to sub-optimal training performance because an individual model clone might be trained on an old copy of the model parameters. Conversely, employing fully synchronous updates will be as slow as the slowest model clone. However, on the grounds that each GPU will have similar speed we decided to employ the second approach, i.e. our training procedure employs synchronous stochastic gradient descent (SGD) across multiple GPUs. The rule of thumb is that synchronous data parallelism (i.e training) is performed on multiple GPUs which are on the same node, while the asynchronous parallelism is used in computations distributed across multiple nodes. Moreover, in a recent paper [17], synchronous gradient updates have demonstrated to reach higher accuracy in a shorter amount of time.

The overhead caused by employing TensorFlow in a Docker container prevented us from exploiting the high-throughput infiniband which connects different nodes. Therefore, we limited our computations to GPUs hosted on a single node. We placed an individual model clone on each GPU and updated model parameters synchronously by waiting for all GPUs to finish processing a batch of data. This setup requires that

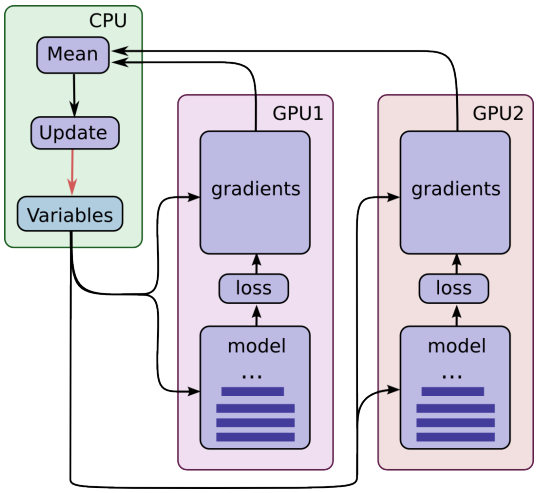

Figure 4.1: Sketch of synchronous training on two GPUs. One step of training means that the CPU sends a model clone to each GPU which processes a batch of images and computes gradients. When both GPUs finish their computations, they send their result back to the CPU which then averages gradients and updates the model for the next training step.

all GPUs share the model parameters. A well-known fact is that transferring data to and from GPUs is quite slow. For this reason, all model parameters are stored and updated with the gradients averaged across all model clones on the CPU. A fresh set of model parameters is then transferred to each GPU before a new batch of data is

[1]not to be confused with the number of replicas that specifies how many worker machines are used for training. In this thesis, all the learning processes are done using GPUs located in a single node/machine and not on GPUs distributed on multiple nodes/machines. Copies of a model on different machines are called *model replicas* in TensorFlow language.

processed, as sketched in Fig. 4.1.

4.2 Performances on Multiple GPUs

The knowledge of the mechanism explained above about synchronous learning is needed to benchmark deep learning models while using multiple GPUs. In particular, when dealing with TF-Slim, one learning step corresponds to the following pipeline. The model clone, together with a batch of images, is given to each GPU (if we set batch size equal to 32 and we start our training with two GPUs, that means an *effective* batch size of 64). Each GPU computes its gradients after the supervised analysis and sends them to the CPU. The latter averages the results gained from the two GPUs and updates the model, which is then sent again to the GPUs for the next step. While training, TF-Slim reports the time required by a single step (`sec/step`), i.e the time needed to perform a feedforward and back-propagation of a batch of images. The number of images processed can be obtained by

$$\texttt{num\_images} = \frac{\texttt{batch\_size}}{\texttt{step\_time}} \times \texttt{num\_gpu} \tag{4.1}$$

The speedup analysis reported below was done separately using up to two Nvidia `Tesla-K20` and up to four Nvidia `GTX-1070` installed on different nodes of `C3HPC` cluster. Moreover, some tests were also done on Galileo (CINECA), which nodes are equipped with two Nvidia `Tesla-K80`, or equivalently to four Nvidia `Tesla-K40`. As a technical note, it is worth mentioning that all these results were obtained loading the Nvidia CUDA Deep Neural Network library (`cuDNN`), which is a GPU-accelerated library of primitives for deep neural networks. `cuDNN` provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. In Fig. 4.2, Inception-v3 and DenseNet-28 were trained for 20 hours with one and two `Tesla-K20` GPUs. In Table 4.1, the results in terms of `imags/sec` are reported together with `Tesla-K40` and `GTX-1070` tests. This metric is often adopted to have a direct and fair comparison of different networks in terms of FLOPs, independently of the batch size. Looking at the Inception-v3, the rate of images processed scales almost linearly with the number of Tesla GPUs, while is not scaling perfectly for `GTX-1070`. For a reason that requires further investigations, DenseNet-28 seems to instead scale reasonably well also with `GTX-1070`. In Fig. 4.3, data from Table 4.1 are shown.

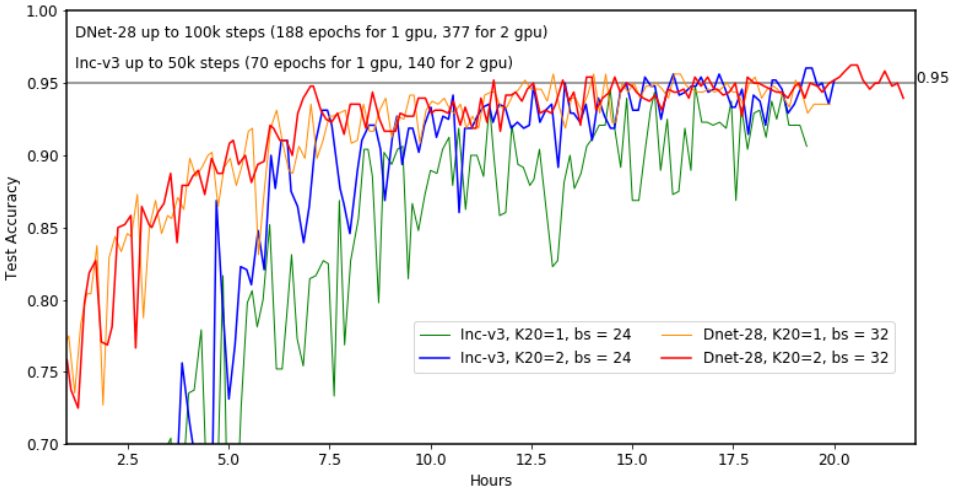

Figure 4.2: Training up to 20 hours Inception-v3 (1 GPU green, 2 GPUs blue) and DenseNet-28 (1 GPU orange, 2 GPUs red) with one and two `Tesla-K20` GPUs on `C3HPC`.

Inception-v3 (imgs/sec)	1	2	4
`Tesla K20`	17.2	33.3	–
`Tesla K40`	23.0	42.1	78.5
`GTX1070`	40.9	56.0	66.4
DenseNet-28 (imgs/sec)	**1**	**2**	**4**
`Tesla K20`	45.5	81.9	–
`GTX1070`	53.5	100.3	183.7

Table 4.1: Number of images processed per second using different GPUs and architectures.

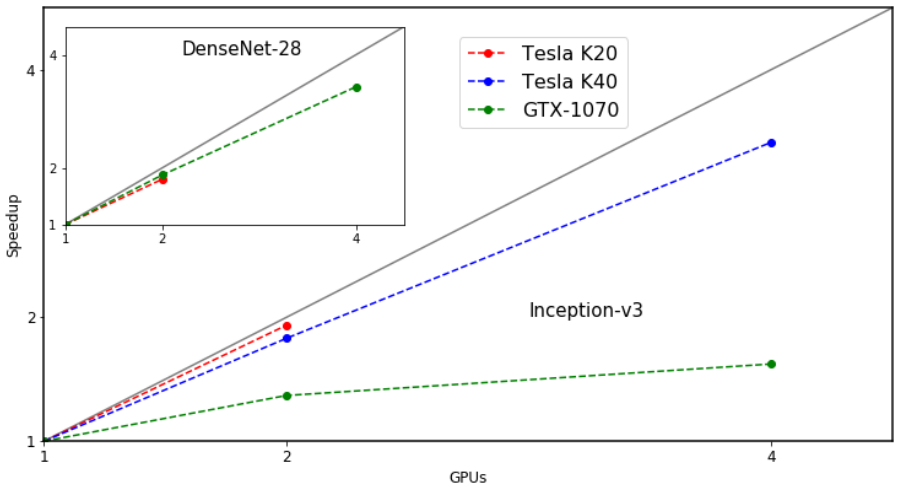

Figure 4.3: (main) Speedup of Inception-v3 on `Tesla-K20`, `Tesla-K40` and `GTX-1070`. (inset) Speedup of DenseNet-28 on `Tesla-K20` and `GTX-1070`.

4.3 Benchmarking Convolutional Neural Networks

The primary purpose of this section is to benchmark relevant operations occurring in deep learning on different hardware platforms, in particular different GPUs and CPUs. Although fundamental computations behind deep learning are well understood, the way they are used in practice can be surprisingly diverse. For example, a matrix multiplication may be compute-bound, bandwidth-bound, or occupancy-bound, based on the size of the matrices being multiplied and the kernel implementation. Because every deep learning model uses these operations with different parameters, the optimization space for hardware and software targeting deep learning is large and underspecified. Big companies, for instance Intel and Nvidia, are working in order to build increasingly fast and optimized processors targeted at deep learning.

4.3.1 Type of Operations

There are plenty of operations when dealing with a CNN, but the most expensive in terms of FLOPs are

- **Dense Matrix Multiplication**: they exist in almost all DNNs nowadays. They are used to implement fully connected layers and consist of GEMM (General Matrix to Matrix Multiplication) operation $A \times B = C$.
- **Convolutions**: they make up the vast majority of FLOPs in networks that operate on images (CNN) and form important parts of networks such as speech and natural language modeling (RNN), thus making them perhaps the single most important layer from a performance perspective. Convolutions have four or five dimensional inputs and outputs giving rise to a large number of possible orderings for these dimensions. One ordering is the NCHW format (channel first), i.e. data is presented in image, number of features (channels), height and width. This format is optimal for `cuDNN` (when using Nvidia GPUs). On the other side, when using TensorFlow on CPU, the default format is NHWC (channel last), i.e. data is presented in image, height, width and number of features (channels). We are going to deal with both of them.

In what follows, we are not going to measure those specific operations but a whole forward and backward (where back-propagation is in action) step.

4.3.2 TensorFlow Performances on GPUs

The following benchmarks were performed with the help of `convnet-benchmarks` [25] on Nvidia `Tesla K20` and `GTX1070` on `C3HPC` cluster, and `Tesla K40` on Galileo (CINECA). For this purpose, we choose the simplest CNN architecture, AlexNet.

TensorFlow networks are benchmarked with Nvidia's `cuDNN 6.1` library, considering both forward and forward-backward flows of random images of size $[128, 3, 227, 227]$ (NCHW format). In particular, what we are going to measure are the forward and forward-backward propagation of a batch of images, as reported in Table 4.2. We code a simple script to execute the TensorFlow benchmark

```
$: python bench_script.py --batch_size 128 --num_batches 50 --data_format NCHW
```

AlexNet, `batch_size`=128	Forward (sec/batch)	Forward-Backward (sec/batch)
`Tesla K20`	0.077 ± 0.011	0.230 ± 0.033
`Tesla K40`	0.062 ± 0.021	0.177 ± 0.059
`GTX1070`	0.039 ± 0.006	0.108 ± 0.016

Table 4.2: Time measures, averaged on 50 trials, of forward and forward-backward propagation of a batch of 128 images.

Looking at Table 4.2, we can conclude that `GTX1070` GPU proves to be the fastest among the computing resources available. This was also clear from Table 4.1, where we reported the number of images per second processed through Inception-v3 and DenseNet-28. The point that `Tesla-K40` is slower that `GTX1070` might be surprising, keeping also in consideration that `Tesla-K40` has 2880 CUDA cores, while `GTX1070` only 1920. The reason behind this difference, can be explained considering the single precision (32bit) peak-performance of these GPUs: 4.3 TFLOPs (`Tesla-K40`, based on Pascal architecture) and 6.5 TFLOPs (`GTX1070`, based on Maxwell architecture). `Tesla K40`, which is an evolution of `K20`, is slightly faster than its previous version in particular if we look at the forward-backward timing. In column 2 of Table 4.2, we reported the time needed for the only forward step since this gives direct insight on the inference time. Independently of the computing device, it is worth mentioning that forward-backward flow of images is three times slower that the simple forward propagation, since the back-propagation is more demanding in terms of FLOPs.

4.3.3 Nervana-Neon and TensorFlow Performances on CPUs

In the following two sections, we present some tests done with TensorFlow and Neon (described in Sec. 2.4.4) on Intel CPUs. First, we discuss the results obtained using `Intel Core i7` $-$ `6500U` CPU @ 2.50GHz on our laptop. Lastly, we perform similar tests on CINECA clusters, all provided with Intel processors.

TensorFlow and Neon on our Laptop

TensorFlow implementation of AlexNet architecture in TensorFlow was benchmarked on four `Intel Core i7` $-$ `6500U` CPU @ 2.50GHz (our laptop), taking care about shifting to NHWC format . The command used was

```
$: python bench_script.py --batch_size 128 --num_batches 50 --data_format NHWC
```

where the flag `num_batches` indicates the number of batches to run and average. In the first line of Table 4.3, the timings required for a forward and forward-backward propagations are reported. TensorFlow performances on four CPUs are at least two order of magnitude slower that similar computation performed on one GPU, as expected and reported in Table 4.3. The same test was performed on another deep learning framework: Neon.

To draw a fair comparison, we benchmarked Neon implementation of AlexNet with the same architecture and input data as the TensorFlow CPU version discussed above. The command used was

```
$: python examples/convnet-benchmarks/alexnet.py -b mkl
```

where the flag -b switch on MKL backend. In Table 4.3, the results are reported and Neon demonstrated to be more than twice faster on the `Intel Core i7` $-$ `6500U` @ 2.50GHz CPU.

AlexNet, `batch_size`=128	Forward (sec/batch)	Forward-Backward (sec/batch)
`TFlow, Intel CPU (4)`	5.1 ± 1.7	15.6 ± 5.2
`Neon, Intel CPU (4)`	1.9 ± 0.5	6.1 ± 1.3

Table 4.3: Measures of forward and forward-backward propagations (`batch_size` $= 128$) performed on `Intel Core i7` $-$ `6500U` CPU @ 2.50GHz (our laptop).

Neon and TensorFlow on CINECA clusters

Similar benchmarks, together with scalability tests, on Neon (version 2.1.0) and TensorFlow (version 1.3.1) were performed on the following CINECA clusters

- Marconi-A2, where each node is equipped with 68 cores `Intel Xeon Phi`7250(KnightLandings) @ 1.4 GHz;
- Galileo, where each node is equipped with 16 cores `Intel Haswell` @ 2.40 GHz.

In the main panel of Fig. 4.4, we plot the speedups up to 64 cores on Marconi-A2 of forward and forward-backward propagations of a batch of 128 images. These results clearly show how the Neon framework has been optimized for Intel CPUs.

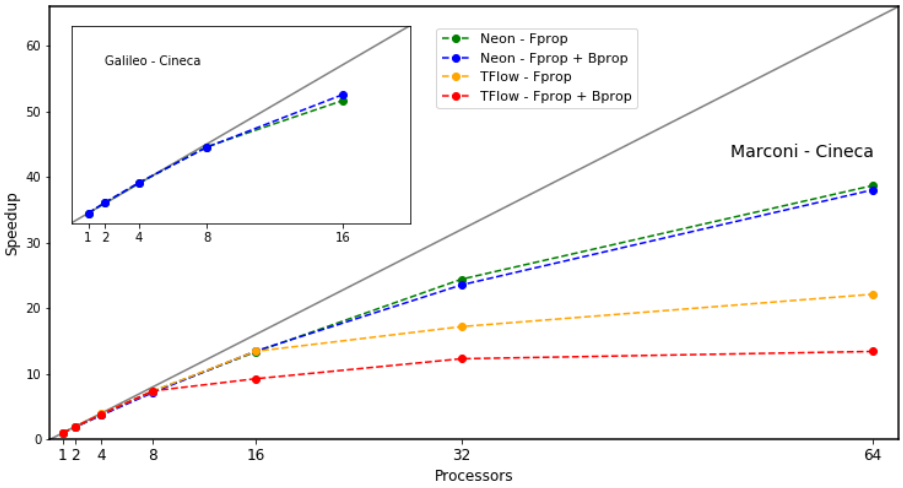

Figure 4.4: (main) Speedup on Marconi cluster of Neon (green and blu) and TensorFlow (orange and red). Batch of 128 images. (inset) Speedup of Neon on Galileo cluster.

Chapter 5

Conclusions

In this Chapter, we summarize the procedures followed in this thesis project and discuss the main results obtained.

We first performed an extensive comparison among different DNN architectures (AlexNet, Inception-v3, Inception-v4, DenseNets) and different deep learning techniques (training from scratch, feature extraction, fine tuning) applied to the SEM dataset. From the results obtained (see Chapter 3), we can conclude that

- higher batch sizes give better accuracies;
- at fixed batch size, DenseNet architecture is the best choice both targeting accuracy and time to solution (95.4% in $\sim$ 47 hours to convergence for Dnet-121 and 96.2% in $\sim$ 33 hours for Dnet-28);
- DenseNets outperforms the other architectures both when trained from scratch and when transfer learning is applied (97.3% after $\sim$ 20 hours);
- at fixed DNN architecture, feature extraction is the fastest method but gives the worst accuracy (90% in $\sim$ 2 hours) since only the last layer is retrained;
- at fixed DNN architecture, fine tuning from a large dataset (i.e ImageNet) is faster and gives higher accuracy than training from scratch on a relatively small dataset;
- if a large dataset is available, can be worth training from scratch a network to have a more dedicated checkpoint. In this case, DenseNet-28 would be a good choice to balance accuracy and time to solution.

As a second step, we benchmarked the TF-Slim implementation of the above models on multiple and different GPU cards. As reported in Chapter 4, we evinced that

- training a model on multiple GPUs allows to increase the effective batch size;

- due to their Pascal architecture, GTX1070 outperforms Tesla cards on training tasks, even though Teslas scales better.

Finally, we compared the performances on CPUs of AlexNet implemented both in TensorFlow and Neon. We can state that

- on our laptop, equipped with Intel Core i7 − 6500U CPU @ 2.50GHz, Neon resulted more than two times faster;
- on CINECA MARCONI-A2 cluster, equipped with 68 cores Intel Xeon Phi7250(KnightLandings) @ 1.4 GHz, Neon speedup scales better.

We additionally highlight the following standalone results achieved:

- a DenseNet implementation was coded in TF-Slim:
- the Inception-v3 model was ported from TF-Slim to Neon for future investigations. It will be added to the architectures available in the growing Neon repository.

Bibliography

[1] NFFA-EUROPE homepage. `http://www.nffa.eu/about/`

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

[3] RDA homepage. `www.rd-alliance.org`

[4] SEM homepage. `http://www.nffa.eu/offer/characterisation/installation-4/sem/`

[5] CNR-IOM homepage. `www.iom.cnr.it`

[6] M. H. Modarres, R. Aversa, S. Cozzini, R. Ciancio, A. Leto, G. P. Brandino, *Neural Network for Nanoscience Scanning Electron Microscope Image Recognition*, Scientific Reports, 2017.

[7] eXact Lab srl homepage. `www.exact-lab.it`

[8] c3hpc homepage. `www.c3hpc.it`

[9] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in International conference on artificial intelligence and statistics, 2010, pp. 249?256.

[10] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv:1502.03167, Feb. 2015.

[11] A. Krizhevsky, I. Sutskever, G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, Advances in neural information processing systems, 1097-1105.

[12] C. Szegedy et al., *Going deeper with convolutions*, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

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

[14] G. Huang, Z. Liu, L. van der Maaten and K. Weinberger, *Densely Connected Convolutional Networks*, arXiv:1608.06993v4

[15] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, K. Q. Weinberger *Memory-Efficient Implementations of DenseNets*, arXiv:1707.06990v1

[16] DenseNet Checkpoints. `https://github.com/shicai/DenseNet-Caffe`

[17] J. Chen, X. Pan, R. Monga, S. Bengio, R. Jozefowicz, *Revisiting Distributed Synchronous SGD*, arXiv:1604.00981v3

[18] J. Yosinski, J. Clune, Y. Bengio, H. Lipson, *How transferable are features in deep neural networks?*, Advances in Neural Information Processing Systems, 2014.

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

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

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

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

[23] TF-Slim homepage. `https://research.googleblog.com/2016/08/tf-slim-high-level-libra`

[24] TF-Slim Github page. `https://github.com/tensorflow/models/tree/master/research/slim`

[25] Convnet-Benchmarks Github page. `https://github.com/soumith/convnet-benchmarks`

[26] Neon Github page. `https://github.com/NervanaSystems/neon`

[27] TensorFlow homepage. `www.tensorflow.org`

[28] ILSVR homepage. `http://www.image-net.org/challenges/LSVRC`

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

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

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

[32] Docker homepage. `www.docker.com`

[33] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016

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

[35] A. Canziani, E. Culurciello, A. Paszke, *An Analysis of Deep Neural Network Models for Practical Applications*,