# Scalable Training of Deep Learning Models on Cloud Infrastructures

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

**Author:** Javier Jorge Cano

**Advisors:** Dr. J. Damián Segrelles
Dr. Germán Moltó

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Máster Universitario en Computación Paralela y Distribuida
*Master's Thesis*

September 2018

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

— Leslie Lamport

# Abstract

In this Master's thesis we focus on deploying and provisioning in an unattended and distributed way a set of nodes that conduct the training of a deep learning model. To this end, the deep learning framework TensorFlow will be used, as well as the Infrastructure Manager that allows us to deploy complex infrastructures programmatically. The final infrastructure should consider: data handling, model training using this data, and the persistence of the trained model. For this purpose, it will be used public Cloud platforms such as Amazon Web Services (AWS) and General-Purpose Computing on Graphics Processing Units (GPGPU).

# Contents


**List of Figures** IX

**List of Tables** XI

**1. Introduction** 1
- 1.1. Motivation . . . 1
- 1.2. Objectives . . . 2
- 1.3. State-of-the-art . . . 3
  - 1.3.1. Machine Learning . . . 4
  - 1.3.2. Deep Learning Frameworks . . . 9
  - 1.3.3. Cloud providers . . . 12
  - 1.3.4. Infrastructure as code . . . 13

**2. Proposal** 17
- 2.1. System's Architecture . . . 17
- 2.2. Deploying with Infrastructure Manager . . . 20
  - 2.2.1. Software provisioning and configuring with Ansible . . . 23
- 2.3. Recipe's definition . . . 25
  - 2.3.1. Resources' definition . . . 25
  - 2.3.2. Operations' definition . . . 27
- 2.4. Distributed TensorFlow training . . . 35
  - 2.4.1. Distributed TensorFlow before Estimator API . . . 35
  - 2.4.2. Distributed TensorFlow after Estimator API . . . 37

**3. Experimentation** 39
- 3.1. Setup . . . 39
  - 3.1.1. Physical resources . . . 39
  - 3.1.2. Cloud Provider . . . 40
  - 3.1.3. Dataset . . . 42

3.1.4. TensorFlow Code . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.2. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.2.1. Budget . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.3. Research problem: Age recognition . . . . . . . . . . . . . . . . . . . . . 47
3.4. Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**4. Conclusions** **51**
4.1. Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

**Bibliography** **53**

# List of Figures

1.1. Neural network representation. . . . . . . . . . . . . . . . . . . . . . . . . 6
1.2. Examples of the distribution of nodes. . . . . . . . . . . . . . . . . . . . . 8
1.3. Distributed training diagram. . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1. Distributed training scheme . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.2. Step 1: Getting the data and uploading it to the HDFS storage. . . . . . 20
2.3. Step 2: Training starts . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.4. Step 3: Upload the model . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.5. Infrastructure Manager's Architecture . . . . . . . . . . . . . . . . . . . . 22

3.1. Images from CIFAR-10 with their categories. . . . . . . . . . . . . . . . . 42
3.2. Inception-Resnet-V1 architecture . . . . . . . . . . . . . . . . . . . . . . . 43
3.3. TensorBoard screenshot . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.4. Images from the provided dataset. . . . . . . . . . . . . . . . . . . . . . . 48

# List of Tables


3.1. Recommended GPU-based instances for Deep Learning Base AMI. . . . 40
3.2. CPU-based instances selected. . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3. Comparison of both training schemes: synchronous and asynchronous. . 44
3.4. Single node results with different hardware. . . . . . . . . . . . . . . . . 45
3.5. Multiple node results with different hardware. . . . . . . . . . . . . . . . 45
3.6. Prices owning a GPU vs in a pay-per-use regime. . . . . . . . . . . . . 47
3.7. Single physical node and multiple node results on Age dataset, with different hardware. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

# Chapter 1

# Introduction

## 1.1. Motivation

Nowadays, Computer Science is experiencing a revolution with the Artificial Intelligence's (AI) subfield known as Machine Learning (ML). This is compound of a set of techniques and methods that try to process and analyse data in order to get valuable information of the past, present or future.

However, this is not something completely new, since the previous description could fit with several statistical techniques, which have been used for decades, to perform the same transformation from data to information. Therefore, what is the difference these days? Why did ML become the way to go for the industry? The answer could be found in the Computer Scientists' contributions that allowed ML to become a product that really works and can be useful for the big audience.

Regarding these contributions, some of them are algorithmic techniques that support the theoretical foundations, dedicated tools to ease the implementation, and specialized hardware to make them competitive. These proposals and approaches, with the vast amount of data currently available almost everywhere have facilitated the development of products that make our lives easier.

Some companies have open-sourced many of their algorithms and methods as frameworks for the community. These frameworks enable a quicker and easier evaluation of the proposals from academia, and some of them are also focused on the deployment in production environments. Some examples of popular toolkits are TensorFlow [1], PyTorch [50] or MXNet [43], to name a few.

For these reasons, and after demonstrating from academia that ML actually works for many tasks, big and not-so-big companies are really interested in leveraging this scenario. There are thousands of examples in our everyday life, just without leaving

our smartphone. In your device, you can find speech recognition to transcript messages, face identification to unlock the device, adapted recommendations in our favourite app, image classification to organize our pictures, and many more, and this is just the beginning.

Considering the interest of the academia and industry to use ML in their pipelines, it is of vital importance to provide better tools, easier and faster ones, to deploy their algorithms and techniques as soon as possible in production. To this end, more and more research groups and companies are using virtualization to develop their ML solutions these days, as long as some of these techniques are computationally demanding and require expensive and dedicated hardware.

Regarding this virtualization, Cloud providers such as Amazon or Google are aware of these needs of the industry, and they are doing their best efforts to provide the whole ML pipeline in their platforms, such as Google's Cloud AI [20] or Amazon SageMaker [6], with virtualized infrastructures or hardware that meets the requirements for these kinds of systems in a pay-per-use regime. However, there is still a lot of work to do in terms of automate the workflow or make it flexible and independent from a particular Cloud provider.

With respect to this flexibility, we can use another layer of abstraction above these providers to overcome the details of each one and define instead the customized computational environment we require, avoiding the interaction with any provider's interface. Using this abstraction, known as "Infrastructure As Code" (IaC) [64], we are able to define, in a Cloud-agnostic way, the virtual infrastructure as well as the process of managing and provisioning the computational resources.

Therefore, the IaC paradigm can ease the development of ML products, providing an ease-to-use interface between the non-expert Cloud user and all the supported Cloud providers, as well as flexibility, making the evaluation of a different infrastructure as simple as a modification of a line of code. This paves the way for a whole automated ML pipeline, from the deployment and provision of the required infrastructure to its execution in production.

## 1.2. Objectives

Taking the aforementioned considerations into account, in this final project we aim to deploy a customized computational environment to perform the training of a machine learning model using an "Infrastructure As Code" solution, to provide software

that comprises the creation, provision and configuration of all the resources involved in the process in an unattended way.

This automation enables a lot of flexibility and ease the training of several models in parallel, launching different architectures. Regarding the expenses, this approach makes the training of the state-of-the-art machine learning models affordable to everyone, as they can pay-per-use in any public Cloud provider to use one or more expensive GPUs as much as they can afford, without any initial important investment. This is very crucial, as there are a lot of implementations of models that are open-source, but the required infrastructure to train them is not accessible to all.

This general objective is composed of the following specific objectives:

- Deploy a transient cluster of nodes to perform the training of deep neural networks using TensorFlow.
- Create, provision and configure the cluster in an unattended way, using the Infrastructure Manager.
- Use the Amazon Web Services public Cloud infrastructure to provision computational resources on a pay-as-you-go basis.
- Use specialized computing hardware such as GPUs to achieve efficient training in the aforementioned public Cloud.
- Evaluate the proposal with a real research problem.

## 1.3. State-of-the-art

In this section we will contextualize the task and the components of the problem, starting with a brief description of machine learning and neural networks, the most successful technique in this field, and the main tools that can be used for developing these kinds of algorithms and models.

Before that, we will show how these models can be trained in parallel, where we will focus on a distributed solution that can benefit from using more than one computational resource.

Considering this, the best solution is to use a Cloud provider to be able to deploy several nodes to distribute the training, as it is done in one of the last breakthroughs from Google, AlphaGO [55]. We will comment on the main public providers and the functionality that they offer. The last part of this section will show tools that work on

top of these providers offering the abstraction of “Infrastructure As Code” that was previously introduced.

### 1.3.1. Machine Learning

As we want to deploy machine learning models, in this section we will give a brief introduction to these techniques. This helps to understand the requirements of these methods and concepts such as kinds of tasks and the model taxonomy, as well as the distributed training that will be introduced later.

Among the numerous techniques that artificial intelligence comprises, machine learning stood out as one of the most successful approaches over the last years, since it is widely used in many contexts and tasks with outstanding results. Notorious examples of this success are personal assistants such as Apple’s Siri [12] or Microsoft’s Cortana [21], Facebook’s face recognition [58] or Google’s translation systems [65], to name a few.

The main goal of these kinds of algorithms is to improve some measure of performance on a particular task [42], based on certain provided knowledge, such as data or user’s input. In general, this could be understood as if the algorithm was “learning” from the experience, in analogy with living beings. While the term “learn” is usually used with these types of techniques, what is actually happening is that there is an optimization problem that the algorithm tries to solve iteratively, adjusting its parameters according to the evidence that is provided in some form. Therefore, ML is more related to function fitting rather than our biological processes of learning.

Regarding the different tasks that involve this new field, we can provide the following taxonomy:

- Supervised: Information of any kind, associated with the example, is provided during the process to discover a mapping from the input (example) to the output (class, label, category, etc.). For example: Classifying handwritten digits where the image of a handwritten digit (input) is provided, as well as the actual value that it represents (output), that is a label between 0 and 9.

- Unsupervised: Just the input is provided, that is the examples without any additional information. The purpose is to discover hidden information or knowledge that naturally comes out from data. For example: Grouping viewer’s preferences (input) to discover target audiences, a technique known as clustering.

During this project we are focused on supervised tasks, as they are the most popular and led to the most interesting advances in this field.

Among these tasks, we can find:

- Classification: Inputs should be mapped to a discrete set of values, i.e: two or more classes, as in the example of digits that has been introduced previously.
- Regression: Inputs should be mapped to one or more continuous values.

In the present project we are going to deal with supervised classification tasks. We are focused on accelerating the training of models that fits in these kinds of tasks, parallelizing them as it will be shown later.

Regarding the ML algorithms that can be used to solve these tasks, many approaches have been proposed. Some popular techniques are: decision trees [16], Support Vector Machines (SVM) [62] and Neural Networks [52] among others.

Currently, Neural Networks (NN) are dominating the field of classification tasks, specially after the disruptive results obtained in 2012 in the most challenging image classification task at that moment [38], defeating the SVM approaches that were leading those contests previously.

We have chosen this model to implement our approach, taking into account their good results and the fact that this technique can get the most out of parallelization, distributing the whole model itself as it will be shown later. In the following section we will give a brief introduction to these types of models.

### Neural Networks and Deep Learning

Neural networks can be depicted as a set of nodes (or neurons) and connections among them. These nodes are grouped in layers, and they are connected with the nodes in the previous layer with edges. These edges represent non-linear transformations of the values in the nodes, and then the whole network represents a set of non-linear transformations over the input that can be seen as an unique non-linear function. Therefore, a NN represents a function as follows:

$$\mathbf{y} = f(\mathbf{x}; w)$$

Where we have an input $\mathbf{x}$ and the output $\mathbf{y}$ as the result of the function parametrized by $w$. This input is provided to the input layer and the output is obtained in the output layer. The data is usually represented as feature vectors, i.e: grey levels of the pixels of a picture or the height and weight of a person, while the output usually

represents a vector of probabilities. The set of parameters $w$ is a set of values assigned to each connection, usually called weights. Figure 1.1 shows a representation of a neural network, where we have the input layer with five neurons, two layers in between usually known as hidden layers with three neurons each, and finally the output layer with two neurons.

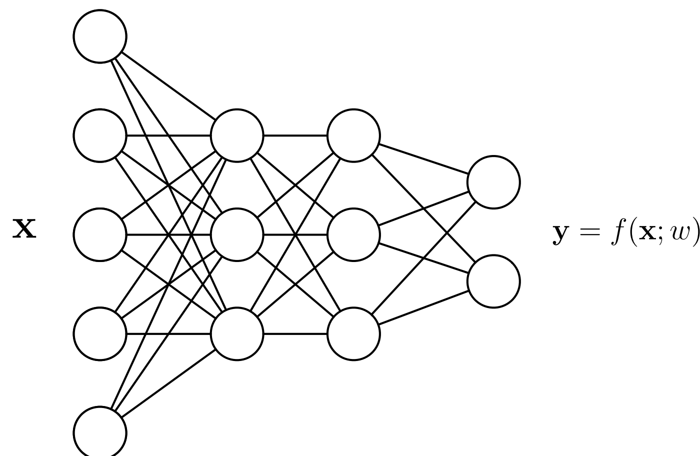



**Figure 1.1.** Neural network representation.

The main goal of this model is to get the weights that optimize some criteria related to a function. Without entering into the theoretical details, as an example of this, we could think about a function that giving an instance, provides the label associated with this instance, in the form of probability represented in the output, i.e.: an image of a digit and the probability of belonging to a class between 0 and 9. The optimization arises when we want to minimize the classification error or the divergence between the real distribution and the estimated one.

The procedure goes as follows: first, it is computed the output of the function with the provided input and the current weights. According to this output, it is obtained a divergence between the actual output and the expected one. Proportionally to this divergence, it is also calculated the modification of these weights (known as the gradients of the function) in order to reduce this divergence in the next iteration. Finally, the weights are modified using a proportion of these gradients. This algorithm is known as backpropagation [52] and it is widely used to adjust the weights of the network iteratively. This process is commonly known as training the model.

There is another stage where we actually want to use the model to make predictions, that is giving an input providing the predicted output, but this project deals just with the training stage and we will not go into the details of this second phase.

Backpropagation is an efficient algorithm, but as we require the use of a lot of data to train the model, in some cases it requires several days or weeks to be trained, and this could be worse if we consider more complex models. Examples of these kinds of models are Convolutional Neural Networks [39], usually used with images, or Recurrent Neural Networks [30] that are meant to model sequential dependencies that can be found in natural language tasks.

These models where the structure of the network is composed of many layers and complex units are known as deep models, comprising the so-famous deep learning. The key behind this is that the model itself is able to learn an internal representation of the problem, filtering the features that are more important using projections through numerous layers, and then with this meaningful representation, performing the desired task, i.e: classifying an image.

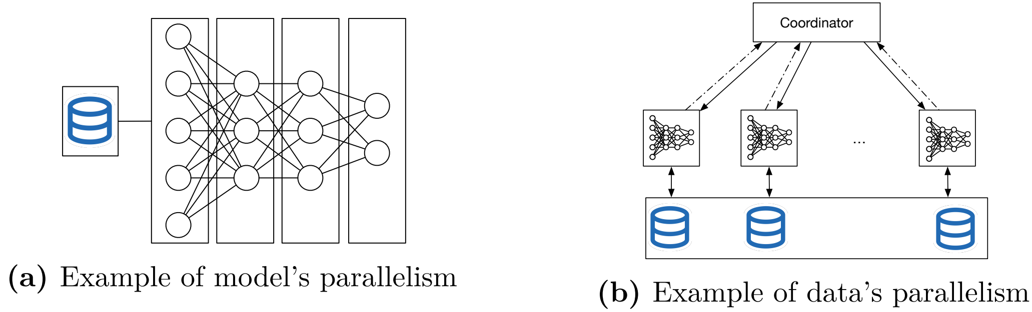
Considering the required number of instances and the models' complexity, several approaches based on parallelization techniques were proposed in literature. They are divided in two groups: based on model's parallelism or based on data's parallelism. In the first group the parallelization comes from dividing the same model among different nodes and then running the backpropagation algorithm as a pipeline. This kind of parallelization implies a lot of interaction among the nodes and it only makes sense when the model is so big that it does not fit in a single node.

The second group of techniques tries to get the most out of the fact that the amount of data that we use to train these models are huge, so we can split our dataset among different nodes that will run the same model, with the same set of parameters, over a different set of examples. There is interaction among nodes when the parameters are updated, but this is simpler and faster than in the first group. This approach is the most popular as it is the most straightforward and easy to implement, and there are many frameworks that provide this functionality. Figure 1.2 illustrates two examples of nodes' distribution for these approaches.

For the aforementioned reasons, during this project we have used neural networks as a ML model, and a parallelization strategy based on data parallelism.

Going briefly into the details of this second approach, we are going to introduce the specific scheme that we want to implement, among different tools for training the model using data parallelism.

As we mentioned before, the aim is to adjust the weights progressively using backpropagation, where we perform a forward pass through the network, that is, obtaining the output with the input of the function and the current weights. Then, we get the error and gradients, that are the corrections that we should apply to the




**(a)** Example of model's parallelism

**(b)** Example of data's parallelism

**Figure 1.2.** Examples of the distribution of nodes for each kind of parallelization's technique. Every box represents a different node.

current weights to improve the performance measure that we selected. Finally, we apply a proportion of these changes in a backward pass through the network.

The idea is to perform the forward pass and the gradient computation individually at each node, calling them worker nodes, and then another node acting as a coordinator gathers all these gradients, averages them, and then sends back to the nodes to perform another iteration. This is the approach presented in [23] and it is illustrated in Figure 1.3, where $w$ is the current set of parameters, $\Delta w$ is the set of gradients and $\eta$ is the proportion of change that we want to apply to obtain the new set of parameter $w'$ that will be provided to the worker nodes. This scheme could work either asynchronously or synchronously. In our case, data is provided by a data layer making it available for every node, represented with the box underneath.

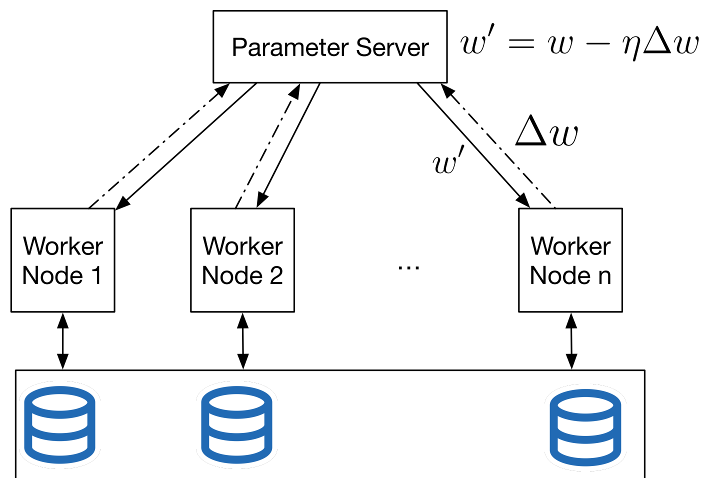



**Figure 1.3.** Distributed training diagram.

### 1.3.2. Deep Learning Frameworks

In this section we are going to introduce different frameworks where it could be developed, trained and tested deep neural networks models in a single or distributed environment. As this is the most popular model nowadays, it is important to include it in our solution through a framework that provides distributed training as one of its features.

Currently, neural networks and deep learning frameworks are proliferating in an untraceable way, since the field is moving very quickly. We selected those that are currently being used or supported by big companies, considering that this field is just a few years old in terms of production-ready development tools. We have also considered those frameworks that offered parallelization using more than a single node. Other criteria such as activity in the development, parallelization functions and the kind of license were also taken into account.

#### Tensorflow

This open-source framework was launched by Google in 2015 [1] and it is one of the most widely used toolkits to develop and train neural networks. It is used intensively in the industry and the academia, and it is constantly maintained and supported by Google, one of the big players in the deep learning community, so it can be seen in its official repository [47]. It is written in C++ and Python and supports the use of Graphics Processing Units (GPU) from NVIDIA [45] to speed up computations using the CUDA library [45]. TensorFlow (TF) provides an API for Python, C++, Java, Go and R, and several implementations of ready-to-use models used in the academia and industry.

TF computations are expressed as a computational graph, where each vertex represents an operation and edges represent the result of these operations. After modelling a neural network like that, the forward pass to obtain the output is reduced to a graph traversal algorithm applying operations. The structures that TF uses are multidimensional arrays, that are referred to as tensors. In order to get the gradient of the function, TF uses internally automatic differentiation techniques that automate the computation of the derivatives.

Considering the functionality, TF allows users to easily realize and combine NN popular model, such as convolutional or recurrent neural networks.

Regarding parallel execution in a distributed environment, as this framework was thought as toolkit for large-scale environments, it supports this functionality since the first releases.

TF is available for Linux, Windows, macOS and Android.

### The Microsoft Cognitive Toolkit

Previously known as CNTK [41], Microsoft developed this open-source toolkit and released it in 2016. Written in C++, it provides a similar abstraction based on computational graph as TF. It offers an API for Python and C++ and several out-of-the-box state-of-the-art models.

The toolkit provides functions to develop the most used NN such as convolutional or recurrent neural networks. As TF, it also uses automatic differentiation and parallelization across multiple GPUs and servers.

Regarding the platforms, it is available on Windows and Linux.

### Keras

Keras, launched in 2015 [36] an originally developed by François Chollet, is a meta-framework that could use Tensorflow or Microsoft Cognitive Toolkit as a back-end. It aims to ease and enable fast experimentation with the development of these models using building blocks that represent high-level structures such as layers or even complete neural networks.

Written in Python, it provides the same functionality as the back-end that it is being used. Depending on this as well, it can be executed on Windows, Linux or macOS.

### Caffe

CAFFE (Convolutional Architecture for Fast Feature Embedding) is a toolkit developed at Berkeley Vision and Learning Center at University of California, Berkeley. It was developed initially in 2013 [18] by Yangqing Jia during his Ph.D. program and it is one of the oldest toolkit for creating, training, evaluating and deploying neural networks.

Unlike TF and Microsoft's toolkit, models and optimizations are defined as plaintext schemas instead of code. However, you can implement your own layers and architectures programatically. It uses automatic differentiation to compute the derivatives during the backward pass.

Written in C++, it supports parallelization in a single node using GPU or in a distributed setup. It offers interfaces to Python, Matlab and C++. It is also multiplatform, being available for Windows, Linux or macOS. There is a second iteration of this toolkit called Caffe2 [19], open-sourced by Facebook.

### PyTorch

Based on Torch [60], PyTorch [50] is younger than the rest of toolkits in this list, being released in 2016, but with a great adoption among machine learning practicioners. Developed and supported by Facebook AI Research (FAIR), it aims to be as close as possible to other popular tools among data scientists as Numpy [44] or Scipy [53]. This is possible thanks to the integrated development in Python, introducing a minimal overhead.

In addition to the use of automatic differentiation and the graph abstraction, it provides dynamic graphs that can change during their execution, providing more flexibility than its competitors. PyTorch is available for Linux, Windows and macOS.

Regarding parallelization, it is optimized for GPU memory usage and provides single and multi-node parallelization functionality as well.

### MXNet

The last toolkit in this list comes from the Apache Software Foundation and it was launched in 2015 [43]. It provides similar functionality to other toolkits: multiple NN models, automatic differentiation, and scalability to multiple GPUs and multiple machines. MXNet is available for Linux, Windows and macOS.

What differentiates MXNet from other toolkits is its early adoption by the public Cloud providers such as Amazon Web Services (AWS) [9] or Microsoft Azure [40]. In fact, AWS supports strongly this framework, including it in the machine learning functionality that is provided in the platform such as AWS Greengrass [4] for the Internet-of-Things (IoT) projects or AWS Lambda [5] as its serverless computing platform.

### Framework selection

What this brief review shows is that, nowadays, there is a uniform set of functions that are provided by almost all toolkits, such as parallelization on single or multiple nodes, flexibility to code using different interfaces, automatic differentiation, similar

abstractions regarding the structure of the network, and "model zoos" to deploy an out-of-the-box model in production quickly.

Then, the decision of choosing one or the other is usually based on the expertise and skills of the team with a specific tool, such as Python or C++. Besides, there is a lot of effort put into standardizing NN models in order to interoperate among different toolkits, with the project Open Neural Networks Exchange Format (ONNX) [48], making the choice of a framework not so critical.

For this project, we have decided to use TensorFlow, considering the previous experience on this toolkit and former related developments in the research group. Once the toolkit is chosen, and recalling that we are interested in deploying a Cloud infrastructure to perform the training, we will show different public providers that we can use to accomplish this.

### 1.3.3. Cloud providers

We have gathered in the following sections the most important Cloud providers where we can deploy our infrastructure. We want to deploy and provision different nodes that communicate each other in order to train a neural network model in a distributed way, using TensorFlow to do that, on top of these platforms. Before this, it is important to remark the different levels of service that these providers could offer:

- Software as A Service (SaaS): Provides software based on a subscription, as on-demand, through a provider, without the complexity of configuring the platform to run it.

- Platform as A Service (PaaS): Provides a platform that allows the developer to develop and run applications on top of it, avoiding the complexity of configuring the infrastructure, such as virtual machines or networks.

- Infrastructure as A Service (IaaS): Provides virtual resources and high level APIs to not configure or interact with any physical device or location.

Considering this taxonomy, we have considered the following platforms.

#### Amazon Web Services

Amazon Web Services (AWS) [9] is the branch of Amazon that offers public on-demand Cloud computing resources. The project started in 2002, although it was launched officially in 2006, when Amazon realized that the retail computing

infrastructure that it had at that time was under-exploited during low season, and it started selling access to their virtual servers.

AWS is leading the market of Cloud providers since then, and it offers a wide variety of functionality of any kind, from a mere virtual machine to a complete autoscaling and multizone infrastructure with thousands of nodes around the globe.

AWS offers IaaS and PaaS products in order to develop SaaS solutions, that are not provided natively. This platform also has storage solutions such as S3 buckets [7] or DynamoDB [2] as a database service.

#### Microsoft Azure

The Cloud computing service created by Microsoft [40] and launched in 2010 was thought as a PaaS, using the .NET framework. Nowadays, it provides also IaaS, and several Windows-based instances, as well as virtual machines running Linux, and other tools from Microsoft such as Internet Information Service (IIS). It provides serverless functionality like AWS with Azure Functions [15].

It includes SQL services such as Cosmos DB [14] as well as storage solutions like Azure Blob Storage [13].

#### Google Cloud

Google Cloud [27] is the Cloud computing infrastructure developed by Google, shared with Google's internal resources. Google provides it since 2008, and as well as AWS and Azure, provides IaaS and PaaS, and serverless computing.

Regarding the storage, it offers the Google Cloud Storage [29] service to persist the resources, and Google Cloud Bigtable [28] as well, as an example of database service.

In a similar way to what happened with frameworks from the previous section, the services offered in public Cloud providers are converging, providing almost the same functionality in any case, leaving other characteristics, such as usability or pricing, the responsability to tip the balance against one option or the other.

### 1.3.4. Infrastructure as code

Nowadays there are solutions to deploy and configure infrastructures in a Cloud-agnostic way, in order to provide a deployment that can be interchangeable among platforms. This is a new paradigm inside Cloud computing, where tools are still in an early stage of development, and they are in constant evolution. This is the kind

of software that we need to develop our solution, as we want to create, configure and execute our training mainly without human interaction and allowing the user to change architecture's parameters, such as number or kind of nodes, using configuration files in a comfortable way.

Infrastructure as code (IaC) is a paradigm where we want to define infrastructures to be deployed, configured and executed programmatically, benefiting the replicability, automation and management of doing it in this way, in contrast to the traditional process with a lot of human effort. This eases the path to a whole automatized pipeline of software deployment and distribution, from the infrastructure to the very last line of code, everything could be coded. Our applications' architecture could be defined once and it would be reproducible, and easy to modify and maintain with this kind of software.

In the context of deep learning, it provides flexibility to test different approaches with an infrastructure adapted to the requirements, from a sanity check with a bunch of generic nodes to thousands of specialized virtual machines with dedicated hardware in the final delivery, just introducing few changes in the infrastructure's definition. It also allows you to execute your experiment in a controlled scenario, with a particular context or software's version.

**Infrastructure Manager**

The Infrastructure Manager (IM) [35, 17] is an open-source platform to deploy on-demand customized virtual computing infrastructures. It allows the user to deploy complex infrastructures choosing among multiple Cloud providers, such as AWS, Azure or Google Cloud. It was developed at *Universitat Politècnica de València* by the "*Grid y Computación de Altas Prestaciones*" group (GRyCAP) [32] and it is actively maintained.

Once the resources are deployed, the configuration is carried out thanks to the integration with Ansible [10], an open-source tool that automates software provisioning and configuration management, among other functionalities. An interesting feature is the use of roles that leads to a more complex setup and functions, providing collections of variables, tasks, files, templates, and modules. This function eases the version control and collaboration in the development, as all the roles have a similar structure. Ansible's roles endorse the idea of one-to-one correspondence between a role and the function of the node that is being configured, that is, after the role is completed, the node should have all that it needs to perform its task.

Regarding interoperability, IM allows you to indicate just your requirements, such as number of CPUs or the amount of memory, and it will contact with the "Virtual Machine image Repository and Catalog"(VRMC) [63] to get a list with the most suitable virtual machine images in the selected provider.

IM can be used through different interfaces: Command Line Interface (CLI), a Web Graphic User Interface (GUI), XML-RPC service API and REST API. IM works with the client-server architecture, so the deployments can be done either synchronously or asynchronously.

Currently, there are other similar tools such as Terraform [59], another popular open-source software to deploy, configure and manage infrastructures as well. Considering that this project comprises a collaboration with the GRyCAP group, we have selected IM as the software to deploy and configure our proposal.

Also, the IM is the orchestration tool adopted by the European Grid Infrastructure (EGI) [26] in order to deploy Virtual Machines through the VMOps Dashboard on the EGI Federated Cloud that spans across Europe. This paves the way for easy adoption of the approach described in this work by the European scientific community.

After introducing the problem and the elements that we have chosen to provide a solution, in the following section, our approach to deploy and configure a TensorFlow cluster to perform the training of a neural network in a distributed way will be described.

# Chapter 2

# Proposal

The main objective of this project is to automatize the deployment, configuration and execution of a distributed deep learning training based on neural networks, on a public cloud provider, using resources that we cannot afford otherwise, such as several nodes connected using one or more expensive GPUs.

Regarding the infrastructure and configuration, we will use the Infrastructure Manager (IM) to create, provision and configure every node of the proposed architecture. We will use Ansible roles, through the IM, to prepare nodes to perform the training with the toolkit TensorFlow (TF).

Firstly, we have identified the architecture of the system, defining the different nodes and roles that will be present in the distributed training. The second section of this chapter will describe the Infrastructure Manager functionality and how we have used it to deploy the desired architecture. We will detail one of the key components of this tool, Ansible, that configures, installs and launches tasks during the process. After that, we will talk about TensorFlow and the modifications that we should perform in our code to train our model in a distributed way. Additionally, we will comment our approach to deploy and provision data to the nodes. Finally, we will describe how to persist the model to perform the inference, both in between training iterations or when it is completed.

## 2.1. System's Architecture

First of all, we should identify and define the nodes that are involve in the architecture. We want to train a deep neural network in a distributed way, getting the most out of data parallelism, considering that the training process could be split according to different shards of data. This escalates very easily, and this is how some of the big

companies such as Google achieved the amazing breakthroughs these last years, such as AlphaGO [55]. Among different approaches for doing this, we have followed the scheme from [23], that is reproduced in Figure 2.1.

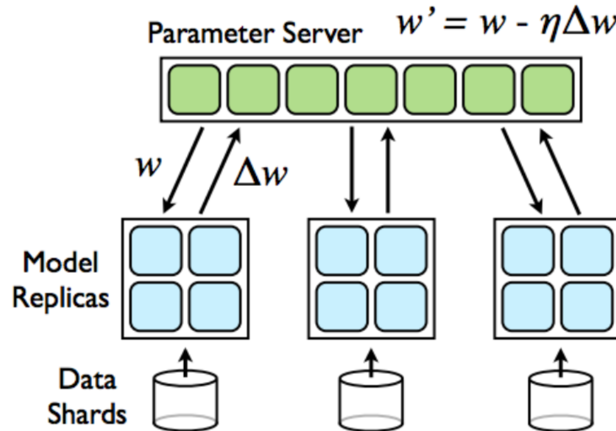



**Figure 2.1.** Distributed training scheme: Parameter server and model replicas, in a master-slave setup [23].

Considering this, we require the following nodes:

- Parameter server (PS): Node(s) that initializes parameters, distributes them to the workers, gathers the gradients and then applies them, and scatters them back to the workers.
- Model replicas: Node(s) that stores the graph and that carries out the Backpropagation's forward and backward steps, and finally sends back the updates to the PS. We will refer to these nodes as workers, and there will be a master among them in charge of persisting the model.
- Data shards: Nodes that provide data for MRs.

This system uses a client-server approach, and as a brief reminder, we will describe the role of each one in detail.

Workers, as clients, perform the most computationally intensive part, that is performing expensive linear algebra operations such as matrix-matrix multiplications that are involve in backpropagation. The key is that the nodes are configured with a GPU that accelerates these kinds of operations. Once these steps are completed, workers have the modifications that should be performed in the weights, according

to their fragment of data. These are the gradients ($\Delta w$) that will be sent to the PS. Then, the node waits until receiving the updated weights ($w'$).

The PS plays the server's role, gathering the gradients ($\Delta w$) from workers, averaging them and then applying the update to the current weights of the model ($w$) in order to get the new ones ($w'$). These will be returned to the workers as the model's new parameters. The main advantage is that this operation is much, much faster than performing a backpropagation's iteration, and it should not be a bottleneck.

This scheme could work either synchronously or asynchronously, meaning that the worker could wait for the rest of the workers to finish their iteration to move to the next one (training synchronously with others) or the node could continue with the next batch of data when it completes the forward-backward step (training asynchronously).

Regarding the data shards, we have proposed the use of a distributed file system among nodes, in order to share data and model checkpoints as well. In particular, we have chosen Hadoop Distributed File System (HDFS) [54], a distributed and scalable file system that is focused on storing massive amounts of data. It has been used mainly on the Java's framework Hadoop [33] to implement the MapReduce paradigm [24].

It requires a cluster structure to distribute data with nodes developing different tasks, such as a `namenode` that stores the cluster's information or `datanodes` that store the distributed files. We have decided to deploy both clusters, the one in charge of data storage and the other one in charge of processing data, in the same resources and nodes, coexisting with each other.

Regarding the training, each node will run a TensorFlow script that implements the aforementioned training's scheme. We will comment on this in its specific section lately.

As a summary, Figure 2.2, 2.3 and 2.4 show the steps that we have proposed to prepare our architecture and perform the training.

In the following section, we will describe how we can use the Infrastructure Manager to deploy this architecture.

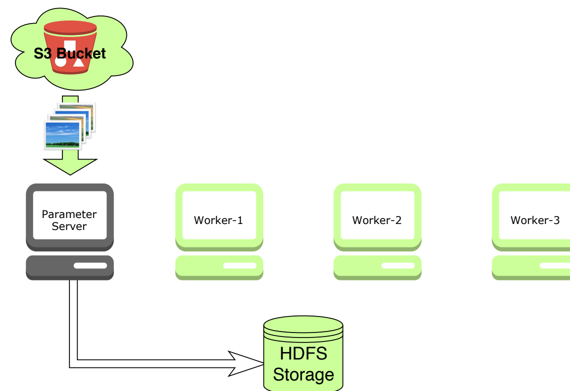



**Figure 2.2.** Step 1: Getting the data and uploading it to the HDFS storage.

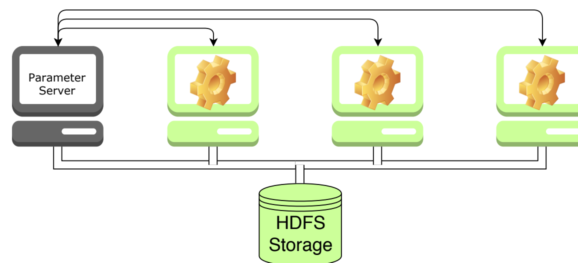



**Figure 2.3.** Step 2: Training starts, the parameter server and worker nodes collaborate to train the model, using TensorFlow and the HDFS file system.

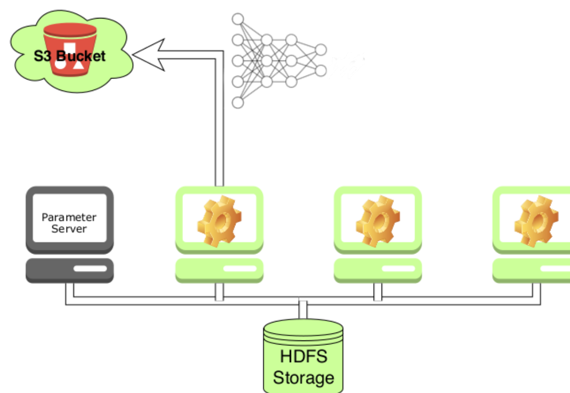



**Figure 2.4.** Step 3: Once training is completed, a worker stores the model in a persistent storage, such as an S3 bucket.

## 2.2. Deploying with Infrastructure Manager

As stated earlier, the Infrastructure Manager can deploy complex and customized virtual infrastructures on multiple back-ends. Moreover, it automates the Virtual Machine Image (VMI) selection, deployment, configuration, software installation,

monitoring and update of virtual infrastructures. Regarding DevOps[1] capabilities, IM provides them thanks to Ansible.

As configuration files, it uses "Resource and Application Description Language" (RADL), a high-level language to define virtual infrastructures and VM requirements. The general structure of these files is as follows:

```
1 network <network_id> (<features>)
2 system <system_id> (<features>)
3 configure <configure_id> (<Ansible recipes>)
4 contextualize [max_time] (
5  system <system_id> configure <configure_id> [step <num>]
6 ...)
7 deploy <system_id> <num> [<cloud_id>]
```

Considering the interaction with the software, IM provides several interfaces to interact with. For this project we have selected the Command Line Interface (CLI), in order to use it requiring no graphic user interface.

Figure 2.5 shows the structure of this tool. IM is a client-server based software, and there are illustrated the ways of interacting with the server side, such as the Web and CLI interface, from the client side, as well as a REST API to interact with other services. Through one of them, the RADL file is sent to the IM server.

Once the RADL file is put in the server, the Cloud Selector manages the interaction with cloud providers, and it launches the Master VM that will orchestrate the deployment and configuration. The module called Configuration Manager configures this master node using Ansible, providing the contextualization and configuration files and installing the Contextualization Agent. This module on the Master VM will deploy, configure and provision the actual resources, in our case, the parameter server and the workers.

After defining the RADL file, the next step is to launch the infrastructure. The CLI interacts with the IM server, providing an authentication file during the interaction, with commands as follows:

- Launch the infrastructure with IM (auth_file.dat contains the cloud-provider credentials):

```
im_client.py -a auth_file.dat create infrastructure.radl
```

[1]Stands for Development and Operations

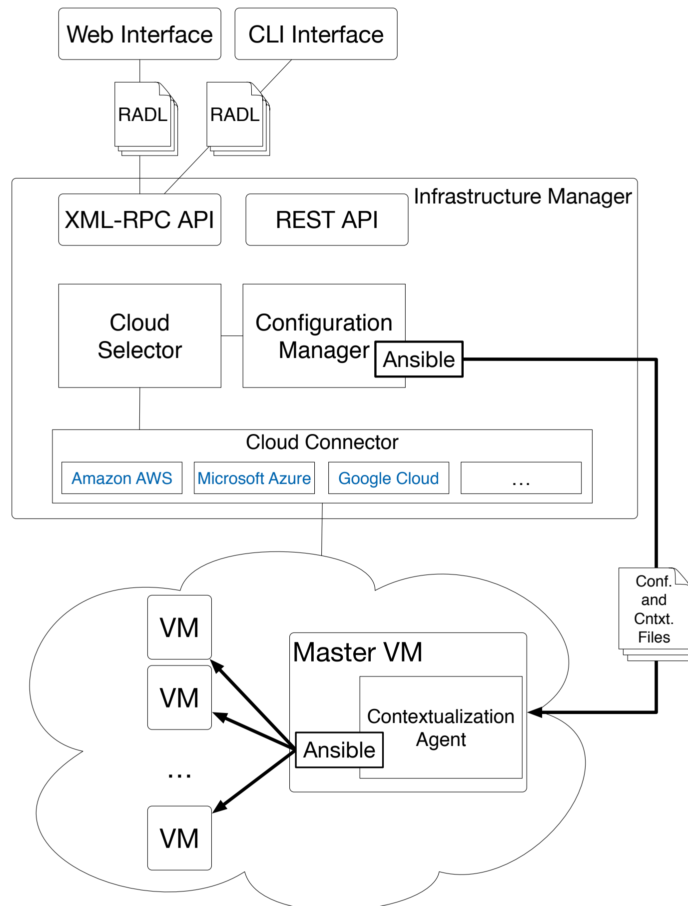



**Figure 2.5.** Infrastructure Manager's Architecture.

- Check the status of the deployment (with the infrastructure-id that you got in the creation):

```
im_client.py -a auth_file.dat getstate <inf-id>
```

- Check the status of the contextualization:

```
im_client.py -a auth_file.dat getcontmsg <inf-id>
```

- List your infrastructures:

```
im_client.py -a auth_file.dat list
```

- Destroy the infrastructure:

```
im_client.py -a auth_file.dat destroy <inf-id>
```

- And other commands, such as `addresource`, `removeresource`, `start`, `stop`, ...

During the deployment of the virtual resources with the IM, we need to provision and configure the required software and resources. Ansible, a DevOps software to perform these steps, is used in the IM in order to do that. We are going to provide a brief introduction to it and how it is used within IM in the following section.

### 2.2.1. Software provisioning and configuring with Ansible

Ansible is a configuration management software that allows us to control and configure machines, providing modules to automate tasks such as provisioning, configuration management, and application deployment. We have used Ansible to perform different functions, such as installing packages, cloning repositories, setting the environment, copying files and running scripts, to name a few.

Ansible's key idea is to provide flexibility to work in a regime where nodes come and go, being destroyed or created on-demand, and we need some mechanism to configure and replace or replicate them in an unattended way.

This tool is based on a *push* approach, as it does not require any agent to work with the resources but a SSH connection with privileges. On the other hand, there are other tools that use the *pull* alternative, installing an agent in the remote node. Puppet [49] is an example of these kinds of tools.

Ansible is oriented to an imperative paradigm and it uses YAML as a format for its configuration files. It is shown an example of Ansible 's piece of code underneath:

```
1  - hosts: node01
2    become: true
3    tasks:
4         - name: ensure apache is at the latest version
5           apt: pkg=apache2 state=latest update_cache=yes
6         - name: ensure apache is running
7           service: name=apache2 state=started
8         - name: Changing index.html
9           copy: dest=/var/www/index.html content="<h1>Hello world!</h1>"
```

Although it is self-descriptive enough, we will make some comments on this fragment. This piece of code ensures that Apache2 [11] is at the latest version, updating otherwise, and then it starts the Apache2 server. Finally, it changes the content of the *index.html* file with the string that is shown. All these steps are performed as the privileged `root` user, due to the `become:yes` instruction.

This will provide the following output in the CLI client, indicating that everything was fine:

```
TASK [ensure apache is at the latest version] **************************
ok: [node01]
TASK [ensure apache is running] ****************************************
ok: [node01]
TASK [copy] ************************************************************
ok: [node01]
```

These and many more are the kind of tasks that we could include in our infrastructure due to the integration with the IM. Regarding this, we have a specific section within the configure directive that allows us to introduce Ansible's operations. However, the most interesting part is the use of another important feature that Ansible provides, known as Ansible roles.

A role is an Ansible's abstraction to encapsulate operations and tasks to perform with a machine in order to be ready to act as some role, such as a HTTP server. Considering this example, it requires a server-side program, like Apache2, to provide the HTTP code, and a configuration file that defines how this works. A role, in the Ansible's context, is a set of files and folders with a specific structure, that contains Ansible code. These directories are organized as follows:

- `defaults`: Folder containing files defining default values, i.e.: number of nodes of each kind, version, etc. In the web-server example, we can provide the default version for Apache2.

- `meta`: Meta information such as authors, description, license, etc.

- `tasks`: It includes files that defines tasks and operations. For example, the updating of the package manager's cache.

- `handlers`: Event catchers that enable reactive deployment, i.e.: launching a task when another has finished.

- `templates`: Files potentially parameterized with Jinja2[2], i.e.: to provide values in the definition, filling them with this during the deployment and copying the resulting files in the VM. For example, the web-server's configuration file that could be adapted depending on the cloud provider.

In conclusion, an Ansible role prepares the node to perform a specific role. In our case, there are two main roles: the parameter server and the worker.

## 2.3. Recipe's definition

### 2.3.1. Resources' definition

In this section, we will introduce the infrastructure's specifics with the help of the RADL configuration file and Ansible roles. We will leave the details related to TensorFlow for the following section, focusing on how we can prepare the environment to execute the training script with this framework. The complete version of the configuration files and code are available on GitHub[3].

First of all, we need to indicate the network parameters. These lines are usually present at the top of the file:

```
1  network publica (
2          outbound = 'yes' and
3          outports = '2222/tcp-2222/tcp,6006/tcp-6006/tcp ' and
4          provider_id = 'vpc-id.subnet-id ')
5
6  network privada(
7          outbound = 'no' and
8          outports = '9000/tcp-9000/tcp,...' and
9          provider_id = 'vpc-id.subnet-id '
10 )
```

We are defining a network called "publica" accesible from any part of the Internet by setting the outbound parameter to 'yes', and the set of ports that we will need to access from outside with SSH and to communicate with other nodes. We are indicating in the `provider_id` the virtual private cloud (VPC) [8] identifier. VPC creates a logically isolated section of the AWS Cloud, and it should exist in the platform beforehand.

[2] http://jinja.pocoo.org/
[3] https://github.com/JJorgeDSIC/Master-Thesis-Scalable-Distributed-Deep-Learning/

Besides, a subnet to connect each node is also required. The same is applied to the network "privada", but in this case we are limiting the access from outside.

After defining networking, we should indicate which kind of nodes we want to deploy. In order to do that, IM includes a section to define "systems", these will be the nodes to compose our architecture. In the following piece of code there is shown, as an example, the required parameters for the parameter server that we are deploying:

```
1  system ps (
2   net_interface.0.connection = 'publica' and
3   net_interface.0.dns_name = 'ps-#N#' and
4   net_interface.1.connection = 'privada' and
5   net_interface.1.dns_name = 'ps-#-priv' and
6    instance_type = 't2.small' and
7   disk.0.image.url = 'aws://zone/ami-id' and
8   disk.0.os.credentials.username='ubuntu' and
9   disk.0.applications contains (
10   name='ansible.modules.git+https://github.com/path/to/role|role_name') and
11  disk.0.os.credentials.public_key = 'user-keypair' and
12  disk.0.os.credentials.private_key = '
13   -----BEGIN RSA PRIVATE KEY-----...-----END RSA PRIVATE KEY-----')
```

Although the order is arbitrary, we have grouped the network parameters at the top, and the parameters associated with the hardware's node at the bottom. We will use this as a general example, as the rest of the nodes follow the same structure. This fragment is related to the technical characteristics of the nodes, and it is defined:

- `net_interface.0.connection`: Connection of the first interface (indexed by 0) to the network "publica", that we have defined previously, with the parameter `connection`.

- `net_interface.0.dns_name`: Node's name in this DNS domain, it will be injected in the nodes in the /etc/hosts file, and it could be parameterized using the special command that replaces `#N#` with the number of the index that this node will have in the deployment, following the order that we will indicate in following sections. In this case: ps-0, ps-1, etc.

- `instance_type`: The instance that will be deployed. The kind of instance is related to different hardware characteristics, such as number of cores, memory, number of GPUs, etc. In this example, `t2.small` is an instance with 1 core and 1 GiB of memory.

- `disk.0.image.url`: Disk image, we have chosen images based on `Ubuntu 16.04` for this project.
- `disk.0.os.credentials.username`: The user that will interact with the virtual machine to install and configure software using Ansible.
- `disk.0.applications contains`: The required Ansible roles for the deployment, pointing to the repository's URL or to the Ansible Galaxy[4] URL. We could include more than one role using more commands like this.
- `disk.0.os.credentials.public_key, disk.0.os.credentials.private_key`: Public and private key to access without the password to the nodes using SSH or other remote tools. If these commands are not provided, they will be created and delivered during deployment.

### 2.3.2. Operations' definition

From this point onwards, we should define the operations and tasks that will be done using Ansible, in the `configure` sections. We can use Ansible code directly or we can apply an Ansible role from the ones that we have indicated in the `system` section for the node.

We will show how we can invoke Ansible both with Ansible's commands directly or using roles, in order to show the two options that it provides. The main structure of these kinds of sections is the following:

```
1  configure example (
2  @begin
3  ---
4  - vars:
5   #Here the place to define variables that will be used on this section.
6    tasks:
7   #Here Ansible tasks that use modules and functions.
8    roles:
9   #Here the roles with their parameters.
10
11 @end
12 )
```

[4] Ansible Galaxy is Ansible's official hub for sharing Ansible content - https://galaxy.ansible.com

### HDFS cluster

To deploy our HDFS system, we have used, forked and adapted the Hadoop role provided by the GRyCAP group on their GitHub[5]. This role defines several variables and task, that we have adapted to our requirements.

```
1  configure hadoop_step1_master (
2  @begin
3  ---
4  - vars:
5    tasks:
6    - name: Ensuring 'PasswordAuthentication no' is not set
7      lineinfile: dest=/etc/ssh/sshd_config regexp="PasswordAuthentication no" \
8                  line="PasswordAuthentication no" state=absent
9    - name: Ensuring 'PasswordAuthentication yes' is set
10     lineinfile: dest=/etc/ssh/sshd_config regexp="PasswordAuthentication yes" \
11                 line="PasswordAuthentication yes" state=present
12
13   - name: Restarting SSH service
14      service: name=ssh state=restarted
15
16   - name: Install openjdk-8-jdk
17     apt: pkg=openjdk-8-jdk state=latest update_cache=yes
18   - name: Preparing the environment
19     lineinfile: dest=/etc/bash.bashrc line="export HADOOP_HOME=/opt/hadoop" \
20      create=yes
21     #...more environment variables definition...
22
23   roles:
24   - { role: 'hadoop',
25           hadoop_version: '2.9.0',
26           hadoop_master: 'ps-0-priv',
27           hadoop_type_of_node: 'master' }
28 @end
29 )
```

We focus on the role part, as the rest of the section is self-explanatory enough. In this case we have the first parameter that indicates the name of the role, differentiating among the roles that we could have indicated in the system section. Then, the rest of the parameters are role dependant, in this case, to indicate the version, who is the master node and then which kind of node is the current one. We have something

[5]https://github.com/JJorgeDSIC/ansible-role-hadoop-1

similar in the rest of the nodes, but pointing that they are acting as slaves concerning the Hadoop cluster. The section for the workers is omitted as it is pretty similar to this but for these last changes. After this, there are two more sections where more environment variables are set.

### Data handling

Once this step is completed, the Hadoop cluster is completely deployed, and then we can proceed with data handling, as well as configuring and installing TensorFlow. We will reserve for later the details related to TF scripts and functionality, as we are focusing now on the deployment, configuration and provision of the architecture. Proceed then with obtaining data and putting it into the HDFS file system.

We have selected as an example the dataset CIFAR-10 [37], which we will comment in more detail lately when we talk about the experimentation. This dataset is already uploaded to an S3[6] bucket, the AWS storage service. On the following recipe's fragment it is performed both the downloading and the uploading of this data into HDFS.

```
1  configure getting_and_distributing_data (
2  @begin
3  ---
4  - vars:
5     access_key: AKIAIS....
6     secret_key: Bcx0up...
7    roles:
8    tasks:
9    - name: Creating data directory
10     file:
11      path: /tmp/cifar-10-data
12      state: directory
13      mode: 0777
14
15   - name: Downloading data (s3) - training
16     s3:
17      aws_access_key: {{ access_key }}
18      aws_secret_key: {{ secret_key }}
19      bucket: bucket_name
20      object: /cifar-10-data/train.tfrecords
21      dest: /tmp/cifar-10-data/train.tfrecords
22      mode: get
```

[6] Stands for Simple Cloud Storage Service - https://aws.amazon.com/s3/

```
23    - name: Downloading data (s3) - validation
24          s3:
25           aws_access_key: {{ access_key }}
26           aws_secret_key: {{ secret_key }}
27           bucket: bucket_name
28           object: /cifar-10-data/validation.tfrecords
29           dest: /tmp/cifar-10-data/validation.tfrecords
30           mode: get
31
32    - name: Uploading data to HDFS
33          command: /opt/hadoop/bin/hadoop fs -put /tmp/cifar-10-data/ /
34
35    - name: Creating model directory on HDFS
36          command: /opt/hadoop/bin/hadoop fs -mkdir /cifar-10-model
37  @end
38  )
```

First, AWS credentials are provided in `vars` section in order to grant access to the S3 bucket where the data is allocated. Before this, we created a folder to store the data locally, and then we can download data to the local storage. We should do this for two different data sets: training and validation. Finally, we put into the HDFS system the data with the Hadoop command that is shown in the code, and we create a directory to store the model as well, in order to be available for everyone.

### Running the TensorFlow role

We are going to use again Ansible roles to launch the training script. We can use it with the following structure:

```
1   configure distributed_tensorflow_ps (
2   @begin
3   ---
4   - roles:
5     {
6           #Role's name, indicated in the system definition.
7           role: 'distributed_tf',
8           #Node's type
9           node_type: 'ps',
10          #Number of GPUs available in the node
11          num_gpus: 1,
12          #Number of ps nodes
13          num_ps_nodes: 1,
```

```
14          #Number of worker nodes
15          num_worker_nodes: 0,
16          #PS location
17          ps_nodes_list: 'ps-0:2222',
18          # Workers' master node
19          master_node: 'master-1:2222',
20          #Comma-separated hosts' list
21          worker_nodes_list: 'worker-2:2222,worker-3:2222',
22          #Whether launch or not tensorboard
23          launch_tensorboard: true,
24          #Whether launch or not the training
25          launch_training: true
26    }
27  @end
28  )
```

This set of parameters is for the parameter server, but it is similar to the workers, with some changes accordingly to their function, such as the `node_type` or whether a GPU is used or not.

The last part of the node is to define how many nodes to deploy of each kind, with the keyword `deploy`, and to establish the order for the `configure` sections to be executed in the `contextualize` section. This is shown underneath.

```
1   deploy ps 1
2   deploy worker 2
3
4   contextualize (
5     system ps configure hadoop_step1_master step 1
6     system worker configure hadoop_step1_slave step 2
7
8     system ps configure hadoop_step2_common step 3
9     system worker configure hadoop_step2_common step 4
10
11    system ps configure hadoop_step3_common step 5
12    system worker configure hadoop_step3_common step 6
13
14    system ps configure getting_and_distributing_data step 7
15
16    system ps configure distributed_tensorflow_ps step 8
17    system worker configure distributed_tensorflow_worker step 9
18
19  )
```

In the next section, we will comment a bit on the role that we used to prepare, install and execute the distributed TensorFlow script.

### Distributed TensorFlow Role

This role involves four parts: first, some environment preparation with paths and additional variables. Second, the conditional installation of a TF version or the other, depending on whether the node has a GPU or not. Third, cloning the actual training scripts and running them, conditioned on the role of the node in the cluster. Finally, we have included the code to upload the model to an S3 bucket to persist it.

The following fragment of the role shows the variable and path definition, indicating to the node how many nodes there are of the different kinds. We have used the library CUDA-9 [22] from NVIDIA, so this should be set accordingly in the `LD_LIBRARY_PATH`.

```
1  - name: Defining some env. variables (PSNODES)
2    lineinfile: dest=/etc/environment line=PSNODES={{ num_ps_nodes }} create=yes
3
4  - name: Defining some env. variables (MNODES)
5    lineinfile: dest=/etc/environment line=MNODES={{ num_master_nodes }} create=yes
6
7  - name: Defining some env. variables (WNODES)
8    lineinfile: dest=/etc/environment line=WNODES={{ num_worker_nodes }} create=yes
9
10 - name: Pointing LD_LIBRARY_PATH paths to 9.0
11   lineinfile: dest=/etc/environment \
12    line="LD_LIBRARY_PATH=...:/usr/local/cuda-9.0/lib64:/:..:" create=yes
```

Then, the conditional installation of TensorFlow, enabling the use of nodes with or without GPU.

```
1  - name: Install Tensorflow
2    pip:
3          name: tensorflow
4          executable: pip3
5    when: not gpu
6
7  - name: Install Tensorflow (GPU)
8    pip:
9          name: tensorflow-gpu
10         executable: pip3
11   when: gpu
12
13
```

```
14 - name: Cloning repository with TF code
15   git:
16    repo: 'https://github.com/JJorgeDSIC/DistributedTensorFlowCodeForIM.git'
17    dest: '{{ clone_path }}'
18    clone: yes
```

In the following step we launch Tensorboard (a TF tool to visualize the training process) and run the training scripts. The parameters shown will be commented in the TF section. Some of these actions are conditioned on variables that we could defined during deployment, providing a lot of flexibility. For example, whether launch the Tensorboard panel or not.

```
1  - name: Launch tensorboard
2    command: ./launch_tensorboard.sh {{ model_path }}
3    when: launch_tensorboard
4    args:
5          chdir: '{{ clone_path }}/{{ work_dir }}'
6  - name: Training (WORKER)
7    command: ./launch_training_dist_async.sh
8                  1 \ # Number of GPUs available
9                  {{ data_path }} \
10                 {{ model_path }} \
11                 {{ train_steps }} \
12                 worker \ # Role in the architecture
13                 {{ ps_nodes_list }} \
14                 {{ master_node }} \
15                 {{ worker_nodes_list }} \
16 #(task Training(WORKER) continuous here)
17   when: node_type == "worker" and launch_training
18   async: 432000
19   poll: 0
20   register: training_finished
21   args:
22         chdir: '{{ clone_path }}/{{ work_dir }}'
23 #... the same for the other roles
24
25 - name: 'Check on async task'
26   async_status:
27         jid: "{{ training_finished.ansible_job_id }}"
28   register: job_result
29   when: node_type == "worker"
30   until: job_result.finished
31   retries: 300
```

It is worth commenting in more detail on the functionality that is shown in the previous fragment, as it is very useful. We can define asynchronous tasks with the command `async` and a maximum time of execution in seconds. It comes with the command `poll`, that indicates (in seconds) how frequently you would like to poll for status, with 0 you are indicating that it is executed without any polling, letting this to the `register` instruction. The command `register` is a trigger to call other tasks, in this case, notifying when the training has finished.

This event will be caught by the following task, that checks on this event, with the command `async_status` and waiting for the job with the job id (command `jid`) to finish, indicating how many retries with `retries`. When this task has been completed, it raises the `job_result` event, that signals that the training is completed. Then, we proceed with saving the model in an S3 bucket, using the `s3` module that Ansible provides. As we mentioned before, this is performed by the special worker defined as master.

```
1  - name: 'Copy model locally'
2    command: /opt/hadoop/bin/hadoop fs -get /cifar-10-model/ /tmp/cifar-10-model
3    register: upload_model
4    when: job_result is succeeded and node_type == "master"
5
6  - name: 'Uploading model'
7    s3:
8          aws_access_key: '{{ access_key }}'
9          aws_secret_key: '{{ secret_key }}'
10         bucket: bucket_name
11         object: /test-cifar-10-model
12         src: /tmp/cifar-10-model
13         mode: put
14   when: upload_model is succeeded and node_type == "master"
```

At this point, we have finished with the required IM and Ansible files to deploy our architecture. As a summary, we have shown the RADL file that the IM needs to deploy the infrastructure, where we:

- Indicate the nodes' characteristics that we need with parameters such as number of cores, memory and so on.
- Declare the operations that we should perform to install, configure and run the required software with `configure` blocks and Ansible roles.
- Define the architecture's number and kind of nodes, as well as the order of the operations to perform upon them.

In the next section, we will show how we have adapted the distributed training scripts in order to use them in our pipeline.

## 2.4. Distributed TensorFlow training

We have used code from the examples provided by Google in the TF repository [47], adapting them to our needs. We have used two different approaches during the development of the project that we will show in this section. We will do a brief overview of these proposals.

Before this, to contextualize a bit, we will comment on the idea behind the distributed TF training. TF distributed training can follow the scheme that was proposed in the system's architecture section, where one or more nodes are acting as parameter servers and the rest act like workers (or model replicas, following the nomenclature of the original paper). In order to do this, it proposes writing a script to run these two roles.

These two versions are posed because of the change in the TF API when it moved from 1.0 to 1.1 version, when the Estimator, a high-level TF API that greatly simplifies machine learning common tasks, appeared. As there are out there several systems that use r.1.0, we will provide code to use both approaches. Although we have described in the previous section the functionality with the second one, the code and configuration files are on GitHub[7] and they should be easy to understand after checking that section.

### 2.4.1. Distributed TensorFlow before Estimator API

In this first approach, the idea is to define a cluster specification in the code where we indicate where and which are the cluster's nodes. This script should have in certain point something similar to this:

```
1 cluster = tf.train.ClusterSpec({
2         'ps':['ps-0:2222'],
3         'worker':['worker-1:2222','worker-2:2222']
4 })
5
6 server = tf.train.Server(cluster,job_name=JOB_NAME,task_index=TASK_INDEX)
```

These instructions are parameterized, in order to assign the correct job name and task index to every node. Regarding these parameters, job name refers to the role that

[7]https://github.com/JJorgeDSIC/Master-Thesis-Scalable-Distributed-Deep-Learning

the node has in the cluster, while task index indicates which node in the nodes' list is currently running the TF script.

Then, we should run the code with a conditional like this, that indicates the code to execute depending on your role.

```
if JOB_NAME == 'ps': #checks if parameter server
        server.join()
else:
        is_chief = (FLAGS.task_index == 0) #checks if this is the chief node
        # Model's definition and regular TF code...
        sess = tf.train.MonitoredTrainingSession(\
                master=server.target, \
                is_chief=is_chief)
        # Session running and regular training execution
```

It is important to remark that the rest of the code remains the same, so the changes that we should introduce are minimal to run a distributed training. We are avoiding here the data handling and the model managing, that are concerned with TF details that are out of the scope of this project.

In order to include this scheme, we have parameterized the code as follows: First, indicating the nodes' addresses and tasks for each one. This is introduced in the code when the infrastructure is created thanks to Ansible roles and Jinja2, that gets the files in the `templates` folder and replaces the values accordingly to the variables provided. For example:

```
# Variables
parameter_servers =  ['ps-0:2222']
workers = ['worker-1:2222', 'worker-2:2222']
job_name = ''ps''
task_index = 0
```

And then, in the file in `templates`:

```
parameter_servers =  {{ parameter_servers }} format=ps-%d:2222, wantlist=True) }}
workers = {{ workers }}
TASK_INDEX = {{ task_index }}
JOB_NAME = {{ job_name }}
```

We can indicate just the number of nodes, and then Jinja2's lists do the rest:

```
#This will generate a sequence like this: 'worker-0:2222', 'worker-1:2222'
workers =  {{ lookup('sequence', \
                'start=0 count={{ num_worker_nodes }} format=worker-%d:2222',\
                 wantlist=True) }}
```

With this ends the first way of performing a distributed training. In the following section we will see a different approach that is even easier and more comfortable.

### 2.4.2. Distributed TensorFlow after Estimator API

With the appearing of the Estimator API the way of programming with TF changed a bit. The idea is to provide high-level functions that encapsulate the different parts of the machine learning pipeline: model, input, training, validation and evaluation. Providing common headers for these steps they can manage training and inference better, as the whole pipeline is decomposed into isolated functions.

Examples are:

- `input_fn(is_training, data_dir, batch_size, num_epochs)`: Provides the input pipeline to feed the model. It should provide the input in a form and size that the model expects.

- `model_fn(features, labels, mode, params)`: Header to define the model. Takes the input from `input_fn` and encapsulates the forward and backward steps, as well as other functions such as logs and so.

After defining this, we can include a main section as follows.

```
1  estimator = tf.estimator.Estimator(model_fn=model_fn, \
2          params={'batch_size': batch_size,})
3
4  train_input = lambda: input_fn(True, data_dir, batch_size, epochs_per_eval)
5
6  eval_input = lambda: input_fn(False, eval_data_dir, batch_size)
7
8  train_spec = tf.estimator.TrainSpec(train_input, max_steps=100000)
9
10 eval_spec = tf.estimator.EvalSpec(eval_input,steps=1000)
11
12 tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

With this, we have the whole pipeline being managed by TF, without worrying about running iterations, evaluation steps, logs or saving the model manually. And another great advantage, we have a program that could run in a single node with CPU or GPU, with multiple GPUs or in a distributed way, just indicating these variations as an environment variable called `TF_CONFIG`. As an example, if we want to run this

script with a parameter server and three worker nodes, we should define TF_CONFIG and run the script as follows, if we were executing the script on the PS node:

```
export TF_CONFIG='{
        "environment": "cloud",
        "model_dir": "/tmp/model",
        "cluster":
                {
                        "ps": ["ps-0:2222"],
                        "master": ["worker-1:2222"],
                        "worker": ["worker-2:2222", "worker-3:2222"]
                },
         "task": {"type": "ps", "index": 0} }'

python3 example.py
```

And TF deals with all the distributed training for us. We have used this approach eventually, as it is the newest and easiest way of performing a distributed training. In order to use it, we have included a shell script that configures this variable accordingly to the parameters provided during the definition of the RADL. This is what is done in the script aforementioned in the role. During the deployment, the IM calls this script and sets TF_CONFIG to launch the training with the adequate parameters.

Regarding communications, TF uses gRPC [31], a high performance, open-source universal Remote Procedure Call (RPC) framework developed at Google, to perform calls among devices in different nodes and/or internally, if there are more than one GPU per node.

After this chapter we have covered the identification of the requirements, the definition of the problem, the solution that has been posed and the different pieces to deploy a fully functional TF cluster. Regarding these pieces, we have shown how we can use the IM to define, deploy, configure and install, thanks to its combination with Ansible, the required infrastructure according to the needs.

Moreover, we have illustrated how Ansible can be used to perform many tasks, that can be grouped and organized in roles. This structure helps to replicate the node's configuration, provision and managing in order to ease the transient node life cycle.

Finally, we have indicated what changes should be made in order to execute distributed TensorFlow code in two different ways.

In the next chapter we will use the configuration files, codes and scripts that we have developed in order to evaluate our approach.

# Chapter 3

# Experimentation

In this chapter we are going to study the proposal in terms of flexibility and timing. For doing this, we have selected a public dataset and a public Cloud provider to deploy our infrastructure and perform the evaluation.

First, we consider executing the training using single nodes, in physical and virtual machines, to get the baseline that we can achieve with the resources at reach. Second, we study the use of a Cloud provider to deploy VM with and without specialized hardware, to compare the results to both single and multiple node configuration.

## 3.1. Setup

### 3.1.1. Physical resources

To perform the local experimentation, we have used the following software and hardware in a physical node:

- OS: Ubuntu 16.04.
- GPU Libraries: NVIDIA CUDA 9 - CUDNN 7.
- Framework: TensorFlow r1.10.
- CPU: Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz (4 cores - 2 thread/core - ~$290.00).
- Memory: 128 GB (8x16GB) - DDR4.
- GPU: GeForce GTX 1080Ti (11GB, 3583 cores @ 1.6Ghz, compute capability: 6.1 - ~$700).

- Storage: 2x HD 3TB + SSD 256GB.

### 3.1.2. Cloud Provider

We have selected Amazon Web Services as the public cloud provider to use during the evaluation, considering the previous experience and work of the author, and the wide functionality that this platform provides. We have carried out an analysis of the kind of instances and pricing, in terms of choosing affordable equipment.

With the purpose of reducing costs, we have selected an Amazon Machine Image (AMI) to accelerate the deployment and avoid some time-consuming tasks such as update packages when we use a fresh Ubuntu 16.04 VM.

About this AMI, it is called Deep Learning Base AMI (Ubuntu) Version 10.0 AMI[1], and it is available in most Amazon EC2 regions. Regarding the type of instance, it can be used with several instance types, from a small CPU-only instance to the latest high-powered multi-GPU instances. Concerning libraries and software, it comes with NVIDIA CUDA 9 and NVIDIA cuDNN 7.

In the Table 3.1 are summarized the different options that are recommended in the documentation[2] for this image:

| Model | GPUs | vCPU | Mem (GiB) | GPU Mem (GiB) | Price (hourly) |
|---|---|---|---|---|---|
| p2.xlarge | 1 | 4 | 61 | 12 | $0.90 |
| p2.8xlarge | 8 | 32 | 488 | 96 | $7.20 |
| p2.16xlarge | 16 | 64 | 732 | 192 | $14.40 |
| p3.2xlarge | 1 | 8 | 61 | 16 | $3.06 |
| p3.8xlarge | 4 | 32 | 244 | 64 | $12.24 |
| p3.16xlarge | 8 | 64 | 488 | 128 | $24.48 |

**Table 3.1.** Recommended GPU-based instances for Deep Learning Base AMI.

The information of these families are summarized below[3]:

- About `p2` (general-purpose GPU compute applications):
  - High frequency Intel Xeon E5-2686 v4 (Broadwell) processors.
  - High-performance NVIDIA K80 GPUs, each with 2,496 parallel processing cores and 12GiB of GPU memory.

[1] https://aws.amazon.com/marketplace/pp/B077GCZ4GR
[2] https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html
[3] https://aws.amazon.com/ec2/instance-types/

- Supports GPUDirect for peer-to-peer GPU communications.
- Provides Enhanced Networking using Elastic Network Adapter (ENA) with up to 25 Gbps of aggregate network bandwidth within a Placement Group.
- EBS-optimized by default at no additional cost.

- About `p3` (latest generation of general purpose GPU instances):
  - Up to 8 NVIDIA Tesla V100 GPUs, each pairing 5,120 CUDA Cores and 640 Tensor Cores.
  - High frequency Intel Xeon E5-2686 v4 (Broadwell) processors.
  - Supports NVLink for peer-to-peer GPU communication.
  - Provide Enhanced Networking using Elastic Network Adapter with up to 25 Gbps of aggregate network bandwidth within a Placement Group.

Other recommended families are the `g2` (old generation) and `g3`, focused on graphics-intensive applications. In order to compare prices, a `g2.2xlarge` will cost $0.65 hourly while `g3.4xlarge` would be $1.14 hourly. However, these are not designed for deep learning, but to be used with computer graphics.

Regarding the GPUs that `pX`'s family uses:

- Tesla K(epler)80[4]: Kepler Architecture (3.X capabilities), 4992 CUDA cores, 24GB GPU-mem, dual GPU (in `p2.xlarge` you are sharing the GPU). Price: ~$3,500.00.
- Tesla V(olta)100[5]: Volta Architecture (7.X capabilities), 5120 CUDA cores, 640 Tensor Cores[6] and 16 GB HBM2. Price: ~$8,000.00.

Regarding the CPU-based that we use when it is required, we have chosen instances `t2.micro` and `t2.small`, whose characteristics are summarized in Table 3.2.

Concerning software, we have used the same versions as the aforementioned physical equipment in all the VM.

[4]https://www.nvidia.com/en-us/data-center/tesla-k80/
[5]https://www.nvidia.com/en-us/data-center/tesla-v100/
[6]Dedicated hardware to perform matrix multiplication, performing 64 floating-point fused-multiply-add (FMA) operations per clock

| Model | vCPU | Mem (GiB) | Price (hourly) |
|---|---|---|---|
| t2.micro | 1 | 1 | $0.0116 |
| t2.small | 1 | 2 | $0.0230 |

**Table 3.2.** CPU-based instances selected.

### 3.1.3. Dataset

We have used the dataset CIFAR-10 [37], that comprises 60k 32x32 colour images, with 10 different categories (6k images per category). The official partitions are: 50k for training, 10k for evaluation. This means that we will train the model with 50k and we will evaluate its accuracy predicting the correct category among the 10 possible with the remaining 10k. Figure 3.1 shows some examples from this dataset with their categories.

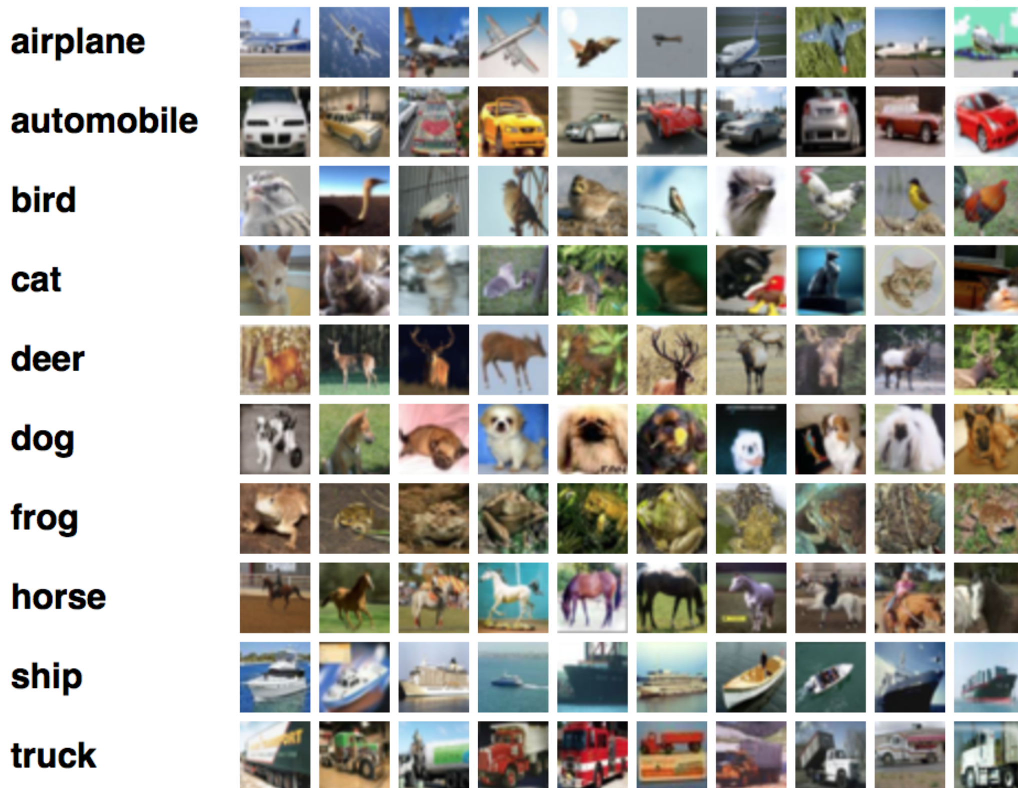



**Figure 3.1.** Images from CIFAR-10 with their categories [37].

We have selected this dataset since it is handy and complicated enough to require a non-trivial neural network model. As this is a dataset of images, we can use a

convolutional neural network where GPUs, with their specialized hardware to process graphics, can bright.

### 3.1.4. TensorFlow Code

We have chosen an implementation provided in the TF official repository[7]. We have executed it without any change, thanks to the use of the Estimator API that was introduced in the previous chapter, in order to illustrate that the code that follows this scheme can be trained inside this infrastructure with no changes. The model that this code uses is the convolutional neural network model known as Inception-Resnet-V1 [34]. This model has 44 layers, millions of parameters, and it is based on one of the first huge deep learning models called Inception [57]. Figure 3.2 shows the whole architecture, decomposed in different blocks.

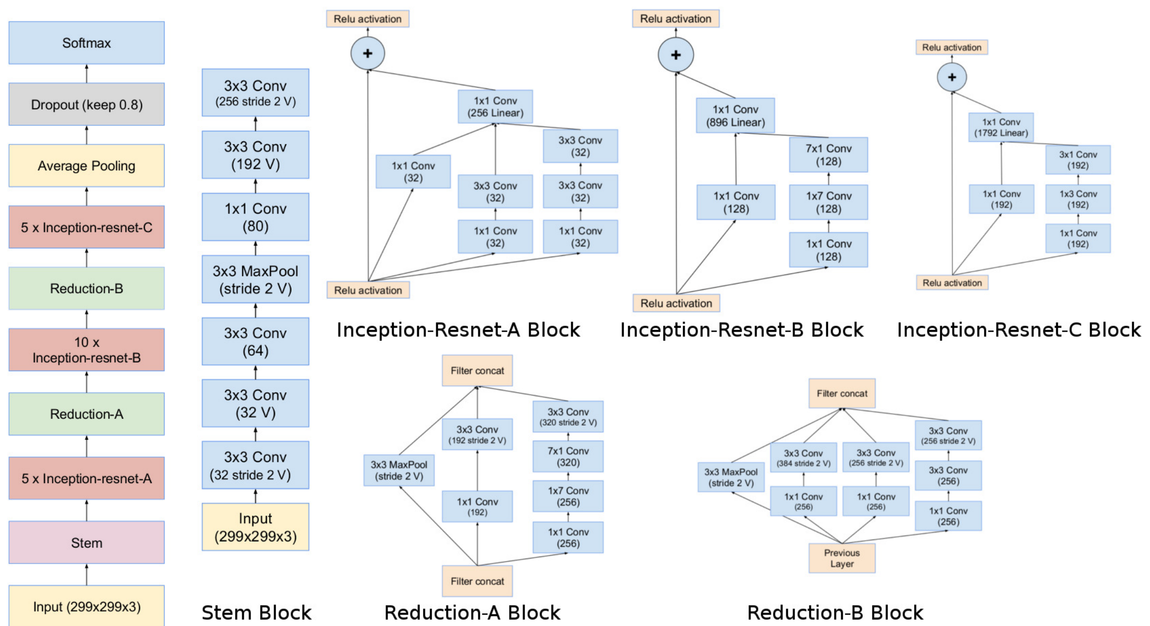



**Figure 3.2.** Inception-Resnet-V1 architecture, on the left it is shown the complete diagram and the blocks' details are on the right [56].

Considering whether asynchronous or synchronous training, we have decided to evaluate both firstly in order to choose one of them. Regarding TF version, we have used the version r1.10, currently the latest one.

[7]https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator

## 3.2. Results

First, we have performed an evaluation to decide whether to conduct an asynchronous or synchronous training. Using a configuration with three GPU nodes, one parameter server and three worker nodes. We have carried out both trainings and we show the results in Table 3.3, evaluating several iterations of the process.

| Training | G.step/sec | Avg.ex/sec |
|---|---|---|
| Synchronous | 3,26±1,14 | 485±67 |
| Asynchronous | 10,04±5,59 | 2755±765 |

**Table 3.3.** Comparison of both training schemes: synchronous and asynchronous.

After getting better results with the asynchronous training, we have chosen this training's scheme for the following experiments.

We have performed 60k steps globally over the training partition, with batches of 128 images to carry out a fair comparison among different architectures. With this training on a physical machine with one GPU we have obtained a classification accuracy of 92,27%. Therefore, we will use this value to compare the accuracy and other timing measures during 60k or the equivalent in a distributed configuration, that is, 30-30 if there are two worker nodes or 20-20-20 if there are three.

Regarding single node experiment, we execute the training with different hardware's configuration. We refer to the physical machines with the prefix `phys`, and `virt` to the virtual ones. We will denote the use of CPU or GPU adding the suffix `_cpu` or `_gpu` respectively.

Regarding measuring, we have evaluated:

- Global step per second: That is the number of batches processed per second. A batch has a size of 128 samples.
- Average examples per second: Number of instances that the model can process per second.
- Total time: How long the training takes to complete. This is not always plausible considering that CPU based nodes can take a considerable amount of time.
- Accuracy: In order to compare the whole training cycle, we ensure that we get a similar accuracy.

We executed the experimentation, repeating the evaluation five times and taking averages and deviations, and Table 3.4 shows the results for the first set of machines.

| Machine | G.step/sec | Avg.ex/sec | T.time (min) | Acc. (%) |
|---|---|---|---|---|
| phys_cpu | 0,44±0,02 | 54,94±2,78 | 2200 (est.) | - |
| virt_cpu | 0,18±0,02 | 25,12±0,92 | 4860 (est.) | - |
| phys_gpu | 22±1 | 2804±22 | 45 | 92,2±0,54 |
| virt_gpu | 7±0,56 | 850±15 | 142 (est.) | - |

**Table 3.4.** Single node results with different hardware.

We noticed the obvious leverage that GPU enjoys with this kind of data. These problems are extremely heavy for CPUs, discouraging any possible training with this hardware. It is important to note the remarkable difference between the physical GPU and the virtual one. We want to evaluate if we can get better results using parallelization.

The second set of experiments involve the use of a distributed configuration. For doing this, we have deployed the infrastructure in AWS and run the experiments, taking the same measures than before. Table 3.5 summarizes the results. We divided accordingly the number of steps among the workers to add up to 60k. Values shown are related to the architecture as a whole.

| Machine | G.step/sec | Avg.ex/sec | T.time (min) | Acc. (%) |
|---|---|---|---|---|
| virt_cpu + virt_gpu x 2 (1 PS - 2 WN) | 7,68±2,95 | 1259±395 | ~92 | 90,23±0,73 |
| virt_gpu x 3 (1 PS - 2 WN) | 13,04±0,48 | 1687±80 | ~41 | 90,87±0,21 |
| virt_cpu + virt_gpu x 3 (1 PS - 3 WN) | 10,04±5,59 | 2755±965 | ~43 | 90,78±0,91 |

**Table 3.5.** Multiple node results with different hardware.

There are some results that show very high variance, this was the side-effect of using a lower-hardware as a PS, such as `t2.small`. During the experiments with this kind of node, the instances per second were decreasing gradually along iterations until they reach lower values than in the `virt_gpu` single training. In contrast to what can be thought, using this fourth node as PS has implied a bottleneck in the system. Figure 3.3 illustrates this fact, that we discovered after suspecting and discarding that the HDFS latency to access to the data or the model were not the cause.

It seems that at certain point, the resources of the CPU-based PS were exhausted, and then the performance got worse. Changing the type of instance solved the problem,

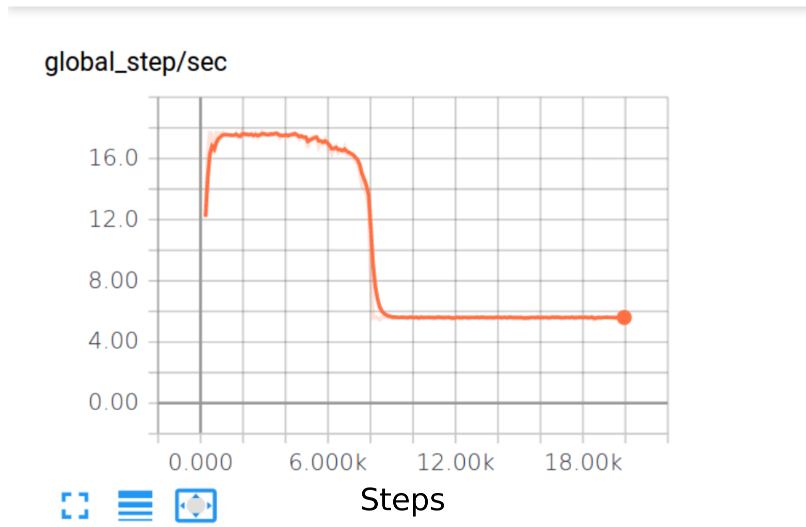



**Figure 3.3.** TensorBoard screenshot, showing the bottleneck in the middle of the training due to the use of low CPU-based instance.

as can be seen in the improvement in the results using `virt_gpu x 3`, providing a stable global step/sec rate, at the expense of losing a GPU node to do the hard work.

Regarding flexibility, we have performed the previous experiments again with the distributed configuration, using our deployment system with the IM and Ansible, changing parameters easily and interacting just with the RADL file, that define the architecture. Considering this, we have found that the best combination is the use of three nodes (a parameter server and two worker nodes) that are GPU-based.

### 3.2.1. Budget

It is worth considering the economic aspect of these experiments, as one reason to put the training on the Cloud is that high-level GPUs are very expensive. Considering this, Table 3.6 shows how much the system costs during the training process in comparison to the fixed cost of buying a high-end GPU.

Finally, the configuration with the three GPUs in parallel achieved a good performance according to the expenses, as the prices' table shows. If we cannot afford a high-end GPU, we can go for these virtual substitutes. Therefore, we can deploy and test our programs with a low-cost architecture, such as these `p2.xlarge` instances, and when we want to train the final model, we just should change some configuration RADL parameters in order to use the latest high-powered multi-GPU instances to speed up our training. There are plenty of room to improve the numbers that there are shown in this project using better instances.

| Machine | Price |
|---|---|
| Tesla K80 | $3,000 (owner) |
| Tesla V100 | $8,000 (owner) |
| Pascal GP102 (GTX 1080 Ti) | $800 (owner) |
| virt_cpu + virt_gpu x 2 (1 PS - 2 WN) | $2,73 (91 m x 60 sec x 2 VM x $0.00025 secondly) |
| virt_gpu x 3 (1 PS - 2 WN) | $1,84 (41 m x 60 sec x 3 VM x $0.00025 secondly) |
| virt_cpu + virt_gpu x 3 (1 PS - 3 WN) | $1,93 (43 m x 60 sec x 3 VM x $0.00025 secondly) |

**Table 3.6.** Prices owning a GPU vs in a pay-per-use regime.

## 3.3. Research problem: Age recognition

We have collaborated during this project with the "*Centro de Computação Gráfica*"[8] from Portugal, in order to evaluate our solution with an actual research problem.

For this collaboration, we have used the dataset that they provided, composed of pictures of faces, labelled with kid, young, adult or elder, aiming to apply this in a security surveillance system. This group want to move their experiments to the Cloud, in order to reduce the time required for training their models. As this part of the project is still under development and it requires more iterations, we will show some measures related to the performance in terms of examples per second and batches per second, as a viability test. We want to evaluate the effort to perform a new task in the infrastructure, considering that the TF code is already implemented using the Estimator API.

Regarding the dataset, it is composed by around 65k RGB images with a resolution of 224x224. We are training the model Resnet again, to predict the label among the set $\{kid, young, adult, elder\}$. Some examples of these pictures are shown in Figure 3.4. It is composed of different sources, the subset that is shown was extracted from the website *Internet Movie Database* (IMDB)[9].

We could perform this training just considering some changes to adapt the parameters from one script to the other, as using the adequate paths for data, checkpoints, and so on. We could deal with the increment of the data easily, as we are using a distributed file system that masks the complexity of distributing the dataset, being

[8]http://www.ccg.pt
[9]https://www.imdb.com

**Figure 3.4.** Images from the provided dataset.

downloaded just by one node and then put in the HDFS system. As a negative part, there is required some time to download the data and put it into HDFS, up to around 10-15 minutes as it is almost 4.5GB of dataset. This could be improved using more advanced and expensive buckets from the S3 offer.

We have evaluated the `phys_gpu`, `virt_gpu x 3` and `virt_cpu + virt_gpu x 3` that provided the best result among virtualization systems. Table 3.7 shows these results.

| Machine | G.step/sec | Avg.ex/sec |
|---|---|---|
| phys_gpu | 18,53±1,82 | 2217±190 |
| virt_gpu x 3<br>(1 PS - 2 WN) | 10,06±0,78 | 1353±47 |
| virt_cpu +<br>virt_gpu x 3<br>(1 PS - 3 WN) | 12,45±2,78 | 1498±317 |

**Table 3.7.** Single physical node and multiple node results on Age dataset, with different hardware.

Again, it could be a convenient solution if you do not have any physical alternative to deploy this architecture, as the results show. It is left the evaluation of more than three GPU instances, but it seems that with one more node, i.e.: an additional GPU to act as a PS, the performance could equal the physical device.

Concerning the storage, we did not perceive any important latency caused by the HDFS system neither in CIFAR-10 nor the faces dataset. Once the dataset is uploaded,

we have performed experiments with and without distributing data and model and not noticing any remarkable difference. Maybe it could happen with massive data, as it requires a more complicated management. Due to time and budget restrictions, we could not evaluate this.

As regards the latency introduced by the deployment, although it is unavoidable, it will be absorbed by the training when it is concerned with a real task, that could imply hours or days of training. The advantages of performing all the steps without human intervention are an excellent payoff. It usually took between 15-20 minutes to deploy the configuration with a `t2.small` and three `p2.xlarge`.

## 3.4. Discussion

Increasing the throughput of the training of deep neural networks have become mandatory in order to solve the challenging tasks that we are facing nowadays. Parallelizing this training using GPUs seems the way to go, since the proposal in [38], where the process of sharing a model between GPUs in order to accelerate the training was done manually with a lot of human effort. This has changed and now it is very easy, as we shown in this project, to code the training using TensorFlow to share the hard work among different GPUs locally and in a distributed configuration. Regarding this, we have shown how to deploy the required infrastructure in a public Cloud provider to get the most out of the functionality that this framework provides.

There are other approaches related to the use of multi-GPUs beyond using different nodes or more than one GPU per node. For instance, GPU virtualization provides the interface to use a variable numbers of virtualizing GPUs, intercepting local calls and sending them to a GPU server farm to be processed, with thousands of GPUs working in parallel. An example of these kinds of proposals is shown in rCUDA [51]. However, these approaches are either experimental or more expensive than public Clouds, and they do not provide additional services required for these kinds of problems such as storage resources. We have proposed a whole system where data and training are considered and coexisting, reducing communication latencies.

The parallel training scheme that is shown here works well in several problems, as it is shown in [23], but there are more approaches, such as Elastic Averaging SGD [67]. This proposes a distributed optimization scheme designed to reduce communication overhead with the parameter server, modifying the function that it computes with an additional term that simulates an "elastic force" between current parameters and the workers'. Ideas in that direction are concerned with the function that the parameter

server computes in order to favour the convergence of the training. This project aimed to develop the architecture and the deployment, as well as implement a basic distributed training, so these kinds of evaluations regarding training convergence are out of the scope of this work.

We have decided to focus on data parallelism as our motivation was to provide the whole deep learning pipeline for problems that benefit this kind of scheme, i.e.: several instances that are distributed in a data cluster, but there are tasks where this is not enough. Among these tasks, we can consider machine translation where models, or the composition of them, are really huge. This is the problem that authors faced in [66], where they used model parallelism as well as data parallelism. For these kinds of models, there is no alternative for training them in a reasonable time.

Regarding infrastructure, nowadays there are alternatives to VM such as containers, that provide operating-system-level virtualization and they are lighter than VM. As they have less to emulate due to the level of abstraction that they provide, the start-up time could be reduced considerably. On the other hand, they require the use of a platform in the system to run them previously installed, the use of a management service such as Amazon EKS [3] and a limited storage capacity.

Although there are examples of distributed systems that perform inference with TF and containers [61], this is not as advanced in terms of performing the training, that normally requires more computational power and storage, and a tight connection with the hardware underneath, such as GPU drivers. Regarding this, NVIDIA provides `nvidia-docker` [46], an interface to interact with Docker [25], another tool to perform operating-system-level virtualization, in order to benefit from the underlying NVIDIA's hardware and the container's flexibility.

# Chapter 4

# Conclusions

During this project we have provided and evaluated a mechanism to deploy, configure and execute a distributed TensorFlow training in an unattended way. Using the Infrastructure Manager and Ansible, we can modify our architecture and train easily just by changing some parameters in a configuration file.

Regarding software, we have used the state-of-the-art tools in their latest version to provide a solution that benefits from the recent software improvements, as TensorFlow r1.10 and CUDA 9. As evaluation, we performed experiments with two different problems and datasets, CIFAR-10 and the Age recognition problem. We have shown that we can run TensorFlow scripts without any additional effort, if they are using the Estimator API, a high-level API that eases the development of deep learning models.

Concerning the results, we are close to the baseline with the physical GPU, even considering that it is two generations ahead of the virtual GPUs that we used. However, we provided a good performance according to the price of these instances, considering a good trade-off. With the present approach, anyone with a limited budget can perform deep learning.

## 4.1. Future work

There are some ideas that we would have liked to try, but due to the time and budget limitations we could not. We will leave as future improvements the following points:

- Using a massive dataset, as dealing with Terabytes of data during training could be a challenge. Maybe the latency to get the information could be a bottleneck

in this situation. We will try to get a bigger dataset to evaluate our approach in the future.

- Test the proposal with dozens of instances, as we were restricted to just three GPU nodes in order to control the budget.
- Use better instances, such as `p3` family, that includes more than one GPU per node, in order to evaluate in-node parallelization and between-nodes parallelization jointly.

# Bibliography


[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

[2] Amazon DynamoDB (2018). [online - Sept. 2018] https://aws.amazon.com/dynamodb/.

[3] Amazon Elastic Container Service for Kubernetes (2018). [online - Sept. 2018] https://aws.amazon.com/eks/.

[4] Amazon Greengrass (2018). [online - Sept. 2018] https://aws.amazon.com/greengrass/.

[5] Amazon Lambda (2018). [online - Sept. 2018] https://aws.amazon.com/lambda/.

[6] Amazon SageMaker (2018). [online - Sept. 2018] https://aws.amazon.com/sagemaker/.

[7] Amazon Simple Cloud Storage Service (2018). [online - Sept. 2018] https://aws.amazon.com/s3/.

[8] Amazon Virtual Private Cloud (2018). [online - Sept. 2018] https://aws.amazon.com/vpc/.

[9] Amazon Web Services (2018). [online - Sept. 2018] https://aws.amazon.com.

[10] Ansible (2018). [online - Sept. 2018] https://www.ansible.com/.

[11] Apache2 (2018). [online - Sept. 2018] https://httpd.apache.org.

[12] Apple Machine Learning Journal (2018). [online - Sept. 2018] https://machinelearning.apple.com.

[13] Azure Blob Storage (2018). [online - Sept. 2018] https://azure.microsoft.com/en-us/services/storage/blobs/.

[14] Azure CosmosDB (2018). [online - Sept. 2018] https://docs.microsoft.com/en-us/azure/cosmos-db/.

[15] Azure Functions (2018). [online - Sept. 2018] https://azure.microsoft.com/en-us/services/functions/.

[16] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). Classification and regression trees.

[17] Caballer, M., Blanquer, I., Moltó, G., and de Alfonso, C. (2015). Dynamic management of virtual infrastructures. *Journal of Grid Computing*, 13(1):53–70.

[18] Caffe (2018). [online - Sept. 2018] http://caffe.berkeleyvision.org.

[19] Caffe2 (2018). [online - Sept. 2018] https://caffe2.ai.

[20] Cloud AI (2018). [online - Sept. 2018] https://cloud.google.com/products/ai/.

[21] Cortana Research (2018). [online - Sept. 2018] https://www.microsoft.com/en-us/research/group/cortana-research/.

[22] CUDA Library (2018). [online - Sept. 2018] https://developer.nvidia.com/cuda-zone.

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

[24] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[25] Docker (2018). [online - Sept. 2018] https://www.docker.com/.

[26] European Grid Infrastructure (2018). [online - Sept. 2018] https://www.egi.eu.

[27] Google Cloud (2018). [online - Sept. 2018] https://cloud.google.com/.

[28] Google Cloud Bigtable (2018). [online - Sept. 2018] https://cloud.google.com/bigtable.

[29] Google Cloud Storage (2018). [online - Sept. 2018] https://cloud.google.com/storage/.

[30] Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM.

[31] gRPC (2018). [online - Sept. 2018] https://grpc.io.

[32] GRyCAP (2018). [online - Sept. 2018] http://www.grycap.upv.es/.

[33] Hadoop (2018). [online - Sept. 2018] http://hadoop.apache.org.

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

[35] Infrastructure Manager (2018). [online - Sept. 2018] http://www.grycap.upv.es/im/.

[36] Keras (2018). [online - Sept. 2018] https://keras.io.

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

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

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

[40] Microsoft Azure (2018). [online - Sept. 2018] https://azure.microsoft.com/.

[41] Microsoft Cognitive Toolkit (2018). [online - Sept. 2018] https://www.microsoft.com/en-us/cognitive-toolkit/.

[42] Mitchell, T. M. et al. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877.

[43] MXNet (2018). [online - Sept. 2018] https://mxnet.apache.orgg.

[44] Numpy (2018). [online - Sept. 2018] http://www.numpy.org.

[45] NVIDIA CUDA library (2018). [online - Sept. 2018] https://developer.nvidia.com/cuda-zone.

[46] NVIDIA Docker (2018). [online - Sept. 2018] https://github.com/NVIDIA/nvidia-docker.

[47] Official TensorFlow Repository (2018). [online - Sept. 2018] https://github.com/tensorflow.

[48] Open Neural Networks Exchange Format (2018). [online - Sept. 2018] https://onnx.ai.

[49] Puppet (2018). [online - Sept. 2018] https://puppet.com/solutions/devops.

[50] PyTorch (2018). [online - Sept. 2018] https://pytorch.org.

[51] Reaño, C., Silla, F., Shainer, G., and Schultz, S. (2015). Local and remote gpus perform similar with edr 100g infiniband. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, page 4. ACM.

[52] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

[53] Scipy (2018). [online - Sept. 2018] https://www.scipy.org.

[54] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee.

[55] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

[56] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12.

[57] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[58] Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708.

[59] Terraform (2018). [online - Sept. 2018] https://www.terraform.io.

[60] Torch (2018). [online - Sept. 2018] http://torch.ch.

[61] Tsai, P.-H., Hong, H.-J., Cheng, A.-C., and Hsu, C.-H. (2017). Distributed analytics in fog computing platforms using tensorflow and kubernetes. In *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific*, pages 145–150. IEEE.

[62] Vapnik, V. N. (1998). *Statistical learning theory*, volume 1. J. Wiley & Sons.

[63] Virtual Machine image Repository and Catalog (2018). [online - Sept. 2018] http://www.grycap.upv.es/vmrc/.

[64] Wittig, M. and Wittig, A. (2016). *Amazon web services in action*. Manning.

[65] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016a). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[66] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., ?ukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016b). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

[67] Zhang, S., Choromanska, A. E., and LeCun, Y. (2015). Deep learning with elastic averaging sgd. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 685–693. Curran Associates, Inc.